Equivalence Checking on ESL Utilizing A Priori Knowledge

Niels Thole*†, Heinz Riener†, Görschwin Fey*†

*Institute of Computer Science, University of Bremen, Bremen, Germany
{nthole, fey}@informatik.uni-bremen.de

†Institute of Space Systems, German Aerospace Center, Bremen, Germany
Heinz.Riener@dlr.de

***Abstract*—We propose EASY, an algorithm for functional equivalence checking of ESL descriptions written in a high-level programming language like C++. Given two ESL descriptions, in a PDR-like fashion EASY systematically refines a candidate invariant to characterize the reachable states of a miter of the descriptions until either the invariant becomes inductive or a counterexample has been found. The algorithm is flexible and allows to incorporate a priori knowledge about the design to speed up the verification process. We provide an implementation of EASY on top of a standard model checker and show in two case studies that EASY outperforms other state-of-the-art equivalence checking tools.**

I. Introduction

Electronic system level (ESL) design methodologies focus on the description of the functionality of an entire system on a high level of abstraction. In contrast to traditional *register-transfer level* (RTL) descriptions, an ESL description captures the behavior of a system while neglecting low-level details like hardware/software partitioning, timing, or power consumption. This allows designers to focus on behavioral characteristics of the system and enables functional verification and validation in early design phases. ESL descriptions are often formulated in general-purpose programming languages like Java or C++.

In this paper, we introduce, EASY[1], an algorithm for functional equivalence checking on ESL. EASY takes as input two ESL descriptions to be checked for functional equivalence, corresponding mappings between the initial states and operations of the two descriptions, and optionally a candidate invariant. In essence, the algorithm uses a *property-directed reachability* (PDR) [5] approach to systematically learn and improve an invariant that characterizes the reachable states of a miter of the two ESL descriptions, until either an inductive correctness proof succeeds or a counterexample has been found that disproves functional equivalence. On termination, the learned invariant serves as a certificate for functional equivalence, whereas a counterexample can be used for debugging the ESL descriptions. The correspondence mappings are necessary to match the two ESL descriptions when they are structurally different. Optionally, a candidate invariant can be provided to approximate the reachable states of the miter of the two ESL descriptions; underapproximation as well as overapproximations are supported. The candidate invariant is a simple way to incorporate knowledge a priori known by the designer into the verification process to speed up reasoning. If the provided candidate invariant indeed is inductive, the algorithm terminates quickly as equivalence can be shown easily. Otherwise, in an attempt to prove functional equivalence, EASY iteratively refines the candidate invariant utilizing counterexamples when functional equivalence checking fails. A counterexample is either spurious, i.e., unreachable from the initial states, then those states can safely be excluded from the candidate invariant, or a counterexample is real, such that a mismatch of the behavior of the two ESL descriptions has been revealed.

This paper makes the following contributions:

1) We describe a light-weight design and verification methodology for embedded systems on the ESL. The behavior of the system is described on a high abstraction level utilizing C++ as flexible modeling language. A system is described as a C++ class — the member variables describe the system's state, whereas the methods describe operations that manipulate the state. The C++ code serves as an executable, functional specification of an embedded system neglecting low-level design details.
2) We present, EASY, a state-of-the-art algorithm to prove or disprove functional equivalence of ESL descriptions that follows the described design methodology and especially allows to incorporate designer knowledge to speed up the reasoning process. On termination, the algorithm produces a certificate in terms of an inductive invariant if the two ESL descriptions are functionally equivalent or a counterexample if functional equivalence was disproved.
3) We provide an implementation of EASY that instruments the given C++ classes with a simple assertion checking scheme and uses CBMC [4] as model checker.
4) In two case studies, we evaluate the practical applicability of EASY and show that our equivalence checking algorithm outperforms our previous version NSMC [11].

The remainder of the paper is structured as follows: first, we present related work in Section II and preliminaries in Section III as well as the data structures used for our algorithm

[1] pronounced as the two letters E.C.

in Section IV. Then, the core algorithm of EASY is proposed in Section V. Lastly, we present two case studies in Section VI. Section VII concludes.

II. Related Work

ESL [2] design and verification was introduced as an emerging design methodology with the intend to allow for fast development, verification, and validation. The heart of ESL is a flexible modeling language. For this purpose often SystemC, HandleC, C++, or Java are used. In this work, we propose a light-weight modeling framework for hardware designs on the ESL. Instead of introducing a new language, the ESL descriptions are implemented in C++ assuming that syntax and semantics of C++ is familiar to many hardware and software engineers and, thus, serves as a suitable ESL modeling language. A hardware design is modeled as a C++ class, where methods correspond to operations and member variables define the system state. A mapping that defines corresponding operations of two ESL descriptions, respectively, has to be provided by a user to deal with structurally different designs.

Functional equivalence checking plays an important role in EDA. Its immediate application is verifying two hardware designs before and after changes, e.g., when optimizations or logic synthesis were applied. Today, RTL-to-RTL equivalence checking is typically available in commercial EDA tools. For ESL-to-RTL equivalence checking, several solutions, e.g., [3], [8], [10], were suggested in academia. Bounded model checking [3] was used to show equivalence of a C program and a Verilog design without focusing on timing. A cycle-accurate data-flow graph [8] that combines an RTL and an ESL description into a miter was suggested for equivalence checking. Functional equivalence checking can be used on the miter utilizing reachability analysis or induction. Moreover, Leung et al. [9] propose a translation validation technique for C to Verilog. As an optimization, equivalence-point detection [6], [1], [7] has been proposed.

PDR [5] (also known as IC3) has recently been proposed as an effective unbounded model checking technique for reactive hardware designs. The algorithm systematically generates counterexamples to strengthen an abstraction of the reachable states, while in contrast to abstraction techniques the transition relation of the system is kept precise. EASY exploits this idea to generate an invariant; however, our algorithm is tailored to functional equivalence checking of ESL descriptions.

III. Preliminaries

In this section, we introduce a variant of *Mealy transducers* [12] to model hardware modules and characterize the functional equivalence of two Mealy transducers based on their input/output behavior.

Definition 1. A *Mealy transducer* $M = (S, S_0, X, Y, \phi, \psi)$ is a tuple, where S is a finite non-empty set of states, $S_0 \subseteq S$ is the finite subset of initial states, X is the input alphabet, Y is the output alphabet, $\phi : S \times X \to S$ is the transition function, and $\psi : S \times X \to Y$ is the output function. ■

For an input $x = x_0 x_1 \dots x_n \in X^*$, we say that $y = y_0 y_1 \dots y_n \in Y^*$ is an output of M if there exists a state sequence $s_0 s_1 \dots s_{n+1} \in S^*$, such that $\forall i \in \mathbb{N}_0, i \leq n : \phi(s_i, x_i) = s_{i+1} \wedge \psi(s_i, x_i) = y_i$, i.e.,

$$s_0 \to^{x_0/y_0} s_1 \to^{x_1,y_1} \dots \to^{x_n/y_n} s_{n+1},$$

where $s_0 \in S_0$ and $s_i \in S$ for $0 \leq i \leq n+1$. We write $y = M(x)$, where $M(x)$ is the output of M for the input x.

In contrast to the standard definition of [12], our definition of Mealy transducers does not define any accepting or final states, but assumes that all states are accepting.

IV. Data Structures

In this section, we will describe how the hardware modules on ESL are modeled and what data structures we use to support the equivalence check. In Section IV-A, we will show how the hardware modules given as C++ classes are described as Mealy Transducers. Section IV-B will introduce lockstep machines, which are used to combine two Mealy transducers into a single new Mealy transducer, and in Section IV-C we will show how logical formulas are used to reason over subsets of states.

A. Modeling Hardware Modules

We model hardware modules on system level as C++ classes. The member variables of a class define the state of the hardware module, whereas the public methods of the class with its arguments define terminating operations that can be executed to change the state and describe the inputs of the according Mealy transducer.

Example 1. For this example, consider two different implementations of a modulo-4 counter. Both use a member variable `counter` which stores the current state and is initialized to 0 as well as a method `countUp` to increase `counter`. The first counter, the modulo-counter, uses a modulo operator to stay within the counting range when counting up, whereas the latter, the if-counter, uses a conditional statement to reset the counter to 0 when the value 3 is increased.

Suppose that $\mathsf{Int}_k = \{0, 1, \dots, k-1\}$. The Mealy transducer $M_{\mathsf{mod}} = (\mathsf{Int}_{256}, \{0\}, \{\texttt{countUp}\}, \mathsf{Int}_4, \phi_{\mathsf{mod}}, \psi_{\mathsf{mod}})$ and $M_{\mathsf{if}} = (\mathsf{Int}_{256}, \{0\}, \{\texttt{countUp}\}, \mathsf{Int}_{256} \setminus \{4\}, \phi_{\mathsf{if}}, \psi_{\mathsf{if}})$ model the input/output behavior of the modulo- and the if-counter, respectively, where for $i \in \mathsf{Int}_{256}$

$$\psi_{\mathsf{mod}}(\texttt{countUp}, i) = (i+1)\%4 \text{ and}$$
$$\psi_{\mathsf{if}}(\texttt{countUp}, i) = \begin{cases} 0, & i = 3 \\ i+1, & \text{else.} \end{cases}$$

Finally, the next-state function and the output function are equal, i.e., $\psi_{\mathsf{mod}} = \phi_{\mathsf{mod}}$ and $\psi_{\mathsf{if}} = \phi_{\mathsf{if}}$. ■

The state spaces and transition functions of the two counter implementations are visualized in Fig. 1 (M_{mod} on the top and M_{if} on the bottom). Each node in the figure corresponds to a possible state and each edge from u to v indicates that state v is reached when the method `countUp` is executed in state u. The initial nodes are marked with an additional incoming edge. The output produced in each state is identical with the counter value in the reached state. Both implementations behave equivalently within the states reachable from their initial states.

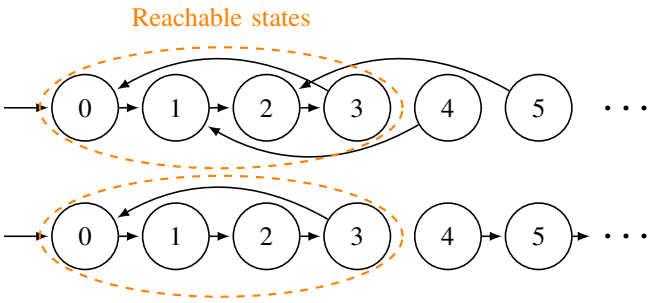


Fig. 1: Visualized state spaces and transition functions of M_{mod} (top) and M_{if} (bottom).

B. Lockstep Machine

We use a lockstep machine to describe the parallel execution of methods on the two considered hardware modules as we want to check if both modules behave in the same way when methods that should be equivalent are called.

Definition 2. Suppose that $M_1 = (S_1, S_{0_1}, X_1, Y_1, \phi_1, \psi_1)$ and $M_2 = (S_2, S_{0_2}, X_2, Y_2, \phi_2, \psi_2)$ are Mealy transducers. A *lockstep machine* $M_* = (S_* = S_1 \times S_2, S_{0_*} = S_{0_1} \times S_{0_2}, X_* = X_1 \times X_2, Y_* = Y_1 \times Y_2, \phi_*, \psi_*)$ is a tuple, where

$$\phi_* : S_* \times X_* \to S_*, (\begin{pmatrix} s' \\ s'' \end{pmatrix}, \begin{pmatrix} x' \\ x'' \end{pmatrix}) \mapsto \begin{pmatrix} \phi_1(s', x') \\ \phi_2(s'', x'') \end{pmatrix} \text{ and}$$

$$\psi_* : S_* \times X_* \to Y_*, (\begin{pmatrix} s' \\ s'' \end{pmatrix}, \begin{pmatrix} x' \\ x'' \end{pmatrix}) \mapsto \begin{pmatrix} \psi_1(s', x') \\ \psi_2(s'', x'') \end{pmatrix}.$$

■

Definition 3. Suppose that $M_* = (S_*, S_{0_*}, X_*, Y_*, \phi_*, \psi_*)$ is a lockstep machine and $(\delta \subseteq S_{0_*}, \Delta \subseteq X_*)$ is a pair called a *correspondence mapping*.

A state $s = (s', s'') \in S_*$ is called *safe* under Δ iff $\psi_1(s', x') = \psi_2(s'', x'')$ for all $(x', x'') \in \Delta$.

Moreover, M_* is called *equivalent* under (δ, Δ) iff for all finite sequences $x = x_1 x_2 \dots x_n \in \Delta^n$ of methods and all initial states $s_0 \in \delta$, M_* reaches a safe state s_n, where $s_i = \phi_*(s_{i-1}, x_i)$ for $1 \leq i \leq n$. ■

C. The Candidate Invariant and Learned Clauses

We use sets of clauses to describe sets of states. In a vector similar to PDR, sets of clauses overapproximate states reachable within a certain number of steps.

For better readability, we sometimes use a set of clauses to describe a formula. In this case, the described formula is a conjunction of all clauses within the set. In addition, sets of states and formulas are sometimes used interchangeably. When a formula is used in place of a set of states, the set contains exactly all states that fulfill the formula. When a set of states is used to describe a formula, the resulting formula is fulfilled for a state iff that state is within the set.

Definition 4. A *clause vector* $\overrightarrow{F} = F_0 F_1 \dots F_N$ with highest index N is a vector, where each F_i is a set of *clauses*, i.e., logic formulas.

A set F_i is called the i-th *frame* and is meant to overapproximate all states that are reachable within i steps, i.e., by executing i methods $m \in \Delta$ starting from an initial state $s_0 \in \delta$.

We define a formula $F_i\uparrow$ as a conjunction over all clauses of $F_i, F_{i+1}, \dots, F_N$, i.e., $F_i\uparrow \;=\; \bigwedge_{j \in \{i, i+1, \dots, N\}} \bigwedge_{c \in F_j} c$.

A clause vector is used to formulate a candidate invariant to prove the equivalence of the lockstep machine $M_* = (S_*, S_{0_*}, X_*, Y_*, \phi_*, \psi_*)$ under a correspondence mapping (δ, Δ). The set F_0 is a special case that describes the initial states and thus, only contains δ as a clause. While advancing $\overrightarrow{F}$, the clause vector always needs to fulfill two properties:

$$\forall i \in \{0, 1, \dots, N-1\} \forall s \in \{s \in S_* | F_i\uparrow (s)\} : s \text{ is safe} \quad (1)$$

$$\forall i \in \mathbb{N}_0 \forall f \in \Delta \forall s \in S_* : F_i\uparrow (s) \to F_{i+1}\uparrow (\phi_*(s, f)) \quad (2)$$

Property 2 describes, that for all $i \in \mathbb{N}_0$, when any method $f \in \Delta$ is called in a state that fulfills $F_i\uparrow$, the state that is reached after the execution of f must fulfill $F_{i+1}\uparrow$. States describe complete assignments of all member variables and a state fulfills a formula if that formula evaluates to $\mathtt{true}$ under that assignment. ■

Lemma 1. When a clause vector contains an empty set at position i, i.e., $F_i = \emptyset$, the according lockstep machine M_* is equivalent.

Proof: Since $F_i = \emptyset$, $F_i\uparrow = F_{i+1}\uparrow$. As all states within $F_i\uparrow$ can only reach states in $F_{i+1}\uparrow = F_i\uparrow$ by property 2, $F_i\uparrow$ is an invariant. By property 1, all states within $F_i\uparrow$ are safe and thus all reachable states are safe and M_* is equivalent. ■

We use a model checker for individual steps during our equivalence check. A call to the model checker uses a *pre-* and a *post-hypothesis*. These hypotheses are formulas over the variables of the checked models. A modelcheck with such a pair of hypotheses checks if all states that fulfill the pre-hypothesis are safe and reach a state that fulfills the post-hypothesis after the execution of any method $f \in \Delta$. When doing a modelcheck, we verify the existence of a counterexample for each method separately but consider all possible arguments of that method nondeterministically. A call to the model checker is given as $\mathsf{Check}(h_{\text{pre}}, h_{\text{post}}, M_1, M_2, \Delta)$.

The call creates a miter for our underlying model checker and returns a counterexample. If no counterexample exists, the checker returns $\perp$. As we usually cannot know if a counterexample describes a non-safe state or a reachable state that does not fulfill the post-hypothesis, we can use the post-hypothesis $\mathtt{true}$ to specifically check for non-safe states without utilizing another interface.

V. Equivalence Checking on ESL

The algorithm decides for two models, their correspondence mapping, and a candidate invariant. The candidate invariant can be $\mathtt{true}$ which would provide the algorithm no additional knowledge about the models. It can be given manually by the developer, who should have detailed knowledge, or can be generated by third-party tools.

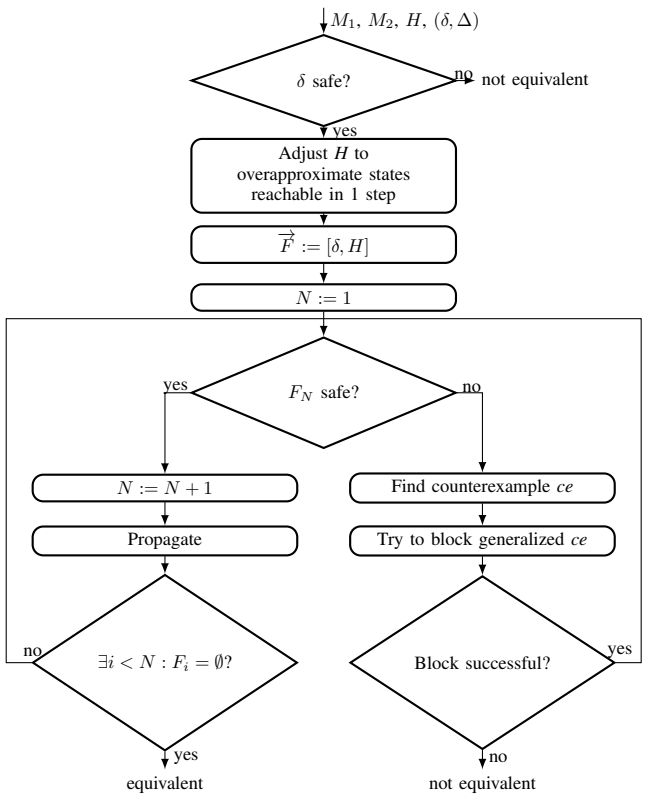


Fig. 2: Checking equivalence between M_1 and M_2 under (δ, Δ)

The basic algorithm is sketched in Fig. 2. We start by checking if the initial states δ of the lockstep machine are safe. If they are not safe, the models are not equivalent.

If they are safe, we check if the candidate invariant H overapproximates all states that are reachable in one step by executing methods from Δ. If this is not the case, we adjust H and remove clauses that are not fulfilled by all states that are reachable in one step. This adjustment is important to ensure, that the initial clause vector $\overrightarrow{F} = [\delta, H]$ fulfills property 2.

After initializing N to 1, we check if all states in F_N are safe. If there are states in F_N that are not safe, there exists a counterexample *ce* that describes an unsafe state. We try to block *ce* and similar assignments by adding according clauses to $\overrightarrow{F}$ without breaking property 2. If this is not successful, the detected unsafe state is reachable and the models are not equivalent. Otherwise, we repeat the process with the modified F_N.

If all states in F_N are safe, we increase the length of $\overrightarrow{F}$ by 1 and try to propagate the clauses as far towards F_N as possible. If there exists an F_i within the clause vector that is empty afterwards, we have proven equivalence according to Lemma 1. Otherwise, we check F_N for safe states again.

Example 2. Consider the counters from Example 1. Both counters are initialized with 0, i.e., $\delta = \{(0, 0)\}$, and both counters use the function `countUp`, i.e., $\Delta = \{\texttt{countUp}, \texttt{countUp}\}$. We want to provide a candidate invariant to speed up the process. We try to provide an upper limit for the counter, but chose faulty values. In addition, we do not add equality to the candidate invariant. This results in the bad candidate invariant $i = \mathit{mod} \leq 2 \wedge \mathit{if} \leq 2$, where *mod* and *if* denote the member variables `count` of two counter implementations, respectively. This results in two clauses for the set H: $\mathit{mod} \leq 2$ and $\mathit{if} \leq 2$.

When we start the algorithm, we check if all initial states in δ are safe. As both counters return 1 when `countUp` is called in the initial state, they are safe.

Next, we check if all states that are reachable within one step fulfill the candidate invariant. As the counter can only count up to 1 within one step, the candidate invariant holds in these states and we do not modify H. Then, $\overrightarrow{F}$ and N are initialized and we check if all states in F_N are safe. This is not the case and we find a counterexample $\mathit{ce} = (\mathit{mod} \equiv 1 \wedge \mathit{if} \equiv 2)$. As *ce* is not reachable from the initial state, we can add $\neg \mathit{ce}$ to F_1 or generalize *ce* and add $(\mathit{if} \equiv \mathit{mod})$ such that

$$F_1 = \{\mathit{mod} \leq 2, \mathit{if} \leq 2, \neg(\mathit{mod} \equiv 1 \wedge \mathit{if} \equiv 2), \mathit{if} \equiv \mathit{mod}\}$$

When checking again, we realize that F_N is safe now and increase N by 1. We then try to propagate clauses. Executing `countUp` in a state that fulfills $F_1\uparrow$ leads to a state where both counters are equal and the detected counterexample is not fulfilled. Thus, we can move $(\mathit{mod} \equiv \mathit{if})$ and $\neg(\mathit{mod} \equiv 1 \wedge \mathit{if} \equiv 2)$ to F_2. However, we cannot move $\mathit{mod} \leq 2$ and $\mathit{if} \leq 2$ as we can reach counter values above 2.

When we check if the states within $F_N = F_2$ are safe, we find a counterexample $\mathit{ce} = (\mathit{mod} \equiv 6 \wedge \mathit{if} \equiv 6)$. Since we already know that the values are equal, we can safely remove one assignment from the counterexample, resulting in $\mathit{ce} = (\mathit{mod} \equiv 6)$. We cannot reach a state that fulfills *ce* from a state that fulfills $F_1\uparrow$. Thus, we can add $\neg \mathit{ce}$ to F_2. In addition, we generalize *ce* and can even add $\mathit{mod} \leq 3$ to F_2, resulting in

$$F_2 = \{\neg(\mathit{mod} \equiv 1 \wedge \mathit{if} \equiv 2), \mathit{mod} \equiv \mathit{if}, \neg(\mathit{mod} \equiv 6), \mathit{mod} \leq 3\}$$

Now, F_2 describes exactly all reachable states of the lockstep machine. We find out that all states within F_N are safe and try to propagate clauses. Since for each clause c in F_2, that clause is fulfilled after calling any function from a state that fulfills $F_2\uparrow$, we can move c from F_2 to F_3. Since we can move all clauses, $F_2 = \emptyset$ afterwards and we return that the models are equivalent under the invariant $F_2\uparrow$. ■

In the following Sections V-A to V-D, we will describe the algorithm, starting at the top level and decreasing the level of abstraction with each section. In the final Section V-E, we will discuss features of the algorithm.

A. Top Level Algorithm

The top level algorithm of EASY is shown in Algorithm 1. It decides if two C++ classes given as Mealy transducers M_1 and M_2 are equivalent under a correspondence mapping (δ, Δ). To speed up the algorithm, a candidate invariant is given as input as well. If the models are equivalent, an invariant is returned. Otherwise, the algorithm returns $\perp$. In this case, F_N contains a reachable counterexample and $\overrightarrow{F}$ describes a way to reach that counterexample.

Algorithm 1: EASY-PDR

input : two Mealy transducers M_1 and M_2 of C++ classes, a correspondence mapping (δ, Δ), and a set of expressions H that describes the candidate invariant
output: an invariant if M_1 and M_2 are equivalent under (δ, Δ) or $\perp$ otherwise

```
1  //Check if initial states are safe
2  if Check(δ, true, M1, M2, Δ) ≠ ⊥ then
3  |  return ⊥
4  end
5  //Check candidate invariant for first step
6  ce := Check(δ, H, M1, M2, Δ)
7  while ce ≠ ⊥ do
8  |  //Weaken candidate invariant if needed
9  |  H := H \ {c ∈ H | c blocks ce}
10 |  ce := Check(δ, H, M1, M2, Δ)
11 end
12 //F0 are the initial states
13 F⃗ . push(δ)
14 //F1 is the candidate invariant
15 F⃗ . push(H)
16 N := 1
17 while true do
18 |  //Check for unsafe states
19 |  ce := Check(FN, true, M1, M2, Δ)
20 |  if ce ≠ ⊥ then
21 |  |  //Recursively block the counterexample
22 |  |  if ¬ BLOCK(TClause(¬ce, N), F⃗) then
23 |  |  |  return ⊥
24 |  |  end
25 |  else
26 |  |  //A new frame and propagating clauses
27 |  |  F⃗ . push(∅)
28 |  |  N := N + 1
29 |  |  F⃗ := PROPAGATE(F⃗, N)
30 |  |  if ∃i < N : Fi = ∅ then
31 |  |  |  return Fi↑
32 |  |  end
33 |  end
34 end
```

Algorithm 2: BLOCK

input : a TClause *c0* that contains a clause *c0.clause* and a number *c0.frame* where *c0.clause* needs to be fulfilled in *c0.frame* and a clause vector $\overrightarrow{F}$
output: a Boolean value that is true iff the counterexample was blocked

```
1  PrioQ<TClause>Q
2  Q . add(c0)
3  while Q . size > 0 do
4  |  c := Q . popMin()
5  |  f := c.frame, cl := c.clause
6  |  //Detected a real counterexample?
7  |  if f = 0 then return false
8  |  if ¬ follows(c, Ff↑) then
9  |  |  C := GENERALIZE(cl, Ff↑, Ff+1↑)
10 |  |  //Is C following from the previous frame?
11 |  |  ce := Check(Ff−1↑, C, M1, M2, Δ)
12 |  |  if ce = ⊥ then
13 |  |  |  Ff := Ff ∪ C
14 |  |  else
15 |  |  |  //ce and C need to be checked
16 |  |  |  Q . add(TClause(¬ce, f − 1))
17 |  |  |  foreach c′ ∈ C do
18 |  |  |  |  Q . add(TClause(c′, f))
19 |  |  |  end
20 |  |  end
21 |  end
22 end
23 return true
```

In the beginning of the algorithm, we check if the initial states δ given by the correspondence mapping are safe. This is done in line 2 by using Check from Section IV-C. If a non-safe initial state exists in δ, the models cannot be equivalent and $\perp$ is returned in line 3.

In lines 5 – 11, it is checked if the candidate invariant overapproximates all states that are reachable from an initial state within a single step and weaken the candidate invariant if needed.

After these initial checks, we initialize the vector $\overrightarrow{F}$ as $[\delta, H]$ and N as 1. N describes the last index of $\overrightarrow{F}$.

The following loop in lines 17 – 33 will refine the candidate invariant until equivalence is proven or a real counterexample to equivalence is found.

First, we check if the current approximation $F_N\uparrow$ contains only safe states in lines 19 and 20. If an unsafe counterexample *ce* exists, we try to recursively block the detected assignment by calling the algorithm BLOCK which is described in Section V-B. The input of BLOCK is a timed clause *TClause*. In this case, the clause is the negated counterexample and the frame is N since we detected the unsafe state in F_N. If BLOCK does not succeed in blocking, *ce* describes a reachable non-safe state and M_1 and M_2 are not equivalent. This is returned in line 23. Otherwise, *ce* and similar assignments are blocked in F_N.

If no unsafe states exist in F_N, we have proven, that no safe state is reachable in N steps. We add another frame to consider states that are reachable in $N + 1$ steps. Next, the algorithm PROPAGATE described in Section V-C is used to move clauses within $\overrightarrow{F}$ as close to F_N as possible.

If an empty set F_i with $i < N$ exists after the propagation, the models are equivalent according to Lemma 1 and $F_i\uparrow$ is returned as invariant in line 31.

B. Blocking Unsafe States Recursively

The algorithm BLOCK is used to block a detected counterexample in $\overrightarrow{F}$. The counterexample is given as a TClause *c0*, that contains a clause *c0.clause* and a frame number *c0.frame*. The clause *c0.clause* describes the negated assignment of the counterexample and the number *c0.frame* describes

Algorithm 3: PROPAGATE

input : a clause vector $\overrightarrow{F}$ and a number N that describes the size of $\overrightarrow{F}$
output: a clause vector $\overrightarrow{F}$ with propagated clauses

1 **foreach** $i := 1, \ldots, N-1$ **do**
2 **foreach** $c \in F_i$ **do**
3 **if** Check$(F_i\uparrow, c, M_1, M_2, \Delta) = \bot$ **then**
4 $F_{i+1} := F_{i+1} \cup \{c\}$
5 $F_i := F_i \setminus \{c\}$
6 **end**
7 **end**
8 **end**
9 **return** $\overrightarrow{F}$

Algorithm 4: GENERALIZE

input : a clause c that is a negation of an assignment to all variables and two hypothesis h_{pre} and h_{post}
output: a set of clauses C that describes the generalized clause c

1 $c' :=$ REMOVE-NO-CARE$(c, h_{\text{pre}}, h_{\text{post}})$
2 $C := \{c'\} \cup$ CHECK-EQUALS$(c', h_{\text{pre}}, h_{\text{post}})$
3 $C := C \cup$ CHECK-INTERVALS$(c', h_{\text{pre}}, h_{\text{post}})$
4 **return** C

the element of $\overrightarrow{F}$ where *c0.clause* needs to be blocked. The algorithm returns true iff the counterexample was successfully blocked and all states that are reachable within *c0.frame* steps fulfill *c0.clause*.

The algorithm uses a priority queue Q that initially only contains *c0*. While Q is not empty, we pop one element c of Q with the lowest frame number *c.frame* in line 4 and initialize f and *cl* as *c.frame* and *c.clause*, respectively in line 5. If *c.frame* is 0, we have detected a reachable counterexample, as the generated clauses describe a path that leads from an initial state to a state that does not fulfill *c0.clause* and we return false in line 7.

Otherwise, we check if c follows from the clauses of its current frame f. If c follows, we do not need to analyze it further as we know that states that are reachable in f steps fulfill *c.clause*. If c does not follow, we generalize c to also consider similar assignments in line 9 and get a set C of clauses.

If all executions of methods in states that fulfill $F_{f-1}\uparrow$ lead to states that fulfill C, all states that are reachable within f steps must fulfill C. Otherwise, we need to check if the detected counterexample is blocked in the previous frame $f-1$ and add the according TClause to Q in line 16. As we did not show that states that are reachable in f steps fulfill C, we need to put the according TClauses back on Q in lines 17 – 19.

When Q is empty, the loop terminates. The vector $\overrightarrow{F}$ has been modified to ensure that all states that are reachable within *c0.frame* steps fulfill *c0.clause* and the algorithm returns true.

C. Propagating Clauses

The algorithm PROPAGATE modifies $\overrightarrow{F}$ by moving clauses within the sets as far towards F_N as possible while keeping property 2. The algorithm is shown in Algorithm 3.

The outer loop is executed for each F_i except for F_0 because this is the special case of initial states and the last set F_N as a clause cannot be moved further than F_N. Starting with $i = 1$, we check for each clause $c \in F_i$, if we can move c to F_{i+1} in line 3. The clause c can be moved, if all states that are reachable in a single step under the pre-hypothesis $F_i\uparrow$ fulfill c. Since c has been moved to F_{i+1}, it is possible to move c even further as F_{i+1} is checked in the next iteration of the loop. As an optimization, in our implementation we check if we can move all clauses from F_i before checking for each clause individually. This speeds up the algorithm significantly, especially if a good initial candidate invariant was chosen. Finally, the modified vector $\overrightarrow{F}$ is returned in line 9.

D. Generalizing Counterexamples

Algorithm 4 is used to generalize a counterexample, so similar assignments can be considered at the same time. The algorithm receives a clause c as input. The clause c is a negated assignment of values to all variables of the two models. Furthermore, the pre- and post-hypothesis h_{pre} and h_{post} that were used to generate the assignment are given as well. The return value is a set of clauses C that describes the generalized clauses based on c.

Similarly to NSMC, the algorithm checks for irrelevant assignments, equal variables, and tries to limit variables to an interval instead of specific values. Unlike NSMC, a set of learned clauses is returned instead of a single logical formula.

The specific operations used in GENERALIZE are similar to those used in NSMC. Different from NSMC, we use binary search to check for intervals which results in a slightly better runtime in our experiments.

E. Discussion

The described algorithm can easily learn new clauses by using the provided heuristics in the algorithm GENERALIZE. In the current implementation, equality of variables or certain intervals can easily be detected and speed up the decision if these kind of clauses can describe an optimal invariant, i.e., an invariant that suffices to show equivalence inductively.

Furthermore, if there are faulty clauses in the initial candidate invariant, these are left within the sets with lower index during propagation and are easily dropped from the final invariant.

Compared to NSMC, we do not consider a single logical formula, but handle a set of clauses. This allows a finer control over the current candidate invariant and enables actions like dropping problematic clauses, which is not possible in NSMC, where the algorithm would need to learn all problematic states instead, which causes a significant overhead up to non-feasible runtimes. To handle the clauses, the algorithm is structured like PDR.

Like PDR, we create an empty set in F_i when we have successfully detected an inductive invariant as PROPAGATE and BLOCK are similar to PDR with some adjustments

to C++ setting and the initial candidate invariant. However, the algorithm GENERALIZE is different from PDR as we use different heuristics to generate insight into the modules while PDR generates cubes that describe partial assignment to the boolean variables. On the other hand, EASY considers different types of variables and considers relations that are specific to those types, e.g., upper and lower bounds of integer variables.

As further optimization, PROPAGATE, BLOCK, and GENERALIZE could easily be parallelized similar to PDR.

VI. Case Studies

In this section, we present two case studies and compare EASY's performance to the algorithm NSMC [11]. The first casy study attempts to check functional equivalence of two counter designs similar to Example 1. The second case study is dedicated to equivalence checking of an arithmetic unit (with and without pipelining) of a simple processor design.

All experiments were conducted on a Lenovo T43 with an Intel Core i5-3320M CPU with 2.6GHz and 8GB of RAM running Windows 7 Professional 32bit. As model checker, we use CBMC v4.9 [4]. We consider a time limit of 6 hours for finding an inductive invariant and report **T/O** if no such invariant is found within this time limit.

A. Counter

Unlike the counters from Example 1, we use integer variables and increase the maximum value of the counter to 9,999,999 instead of 3. In addition, we also consider up to 10 parallel counters in each model that can be accessed individually.

For these experiments, we consider different candidate invariants. The first candidate invariant *optimal* describes exactly all reachable states with

$$optimal = \bigwedge_{1 \leq i \leq \#\text{counters}} (mod_i \equiv if_i) \wedge 0 \leq mod_i \leq 9,999,999$$

where mod_i and if_i describe the i-th counter of the modulo- and if-counter, respectively.

The second candidate invariant is $\mathtt{true}$ which is the weakest possible overapproximation. This candidate is fulfilled for all possible states. Since the counters behave differently outside of the reachable range and cause faulty output if the counters store different values, the algorithms need to adjust this invariant to be true only for reachable states.

The final candidate invariant *lowBound* is similar to optimal but contains the faulty maximum value of 300 and thus underapproximates the reachable states:

$$lowBound = \bigwedge_{i \in \{1,\ldots,\#\text{counters}\}} (mod_i \equiv if_i) \wedge 0 \leq mod_i \leq 300$$

While this candidate invariant is false for all non-reachable states, it is also false for most reachable states. As such, the algorithms need to weaken the candidate invariant to include all reachable states.

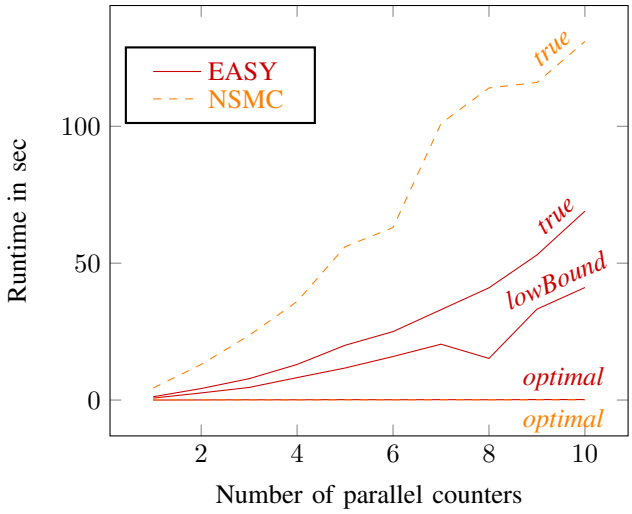
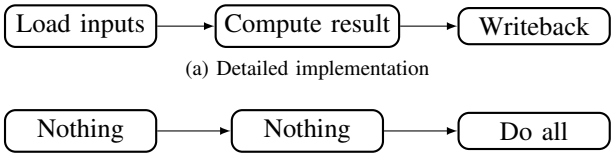


Fig. 3: Runtimes of NSMC and EASY on the counter models



(a) Detailed implementation

(b) Abstract implementation

Fig. 4: Sketch of implemented processors

The experiments in Fig. 3 show, that equivalence can be shown within less than a second when the candidate invariant *optimal* is used and also show that the unneeded overhead that results from the PDR-like implementation from EASY is marginal.

In the next experiment, the candidate invariant is $\mathtt{true}$. In these experiments an invariant needs to be learned. In these experiments, we can see that intervals are detected faster when using binary search. For one parallel counter, EASY needs 134 of its 162 calls to CBMC to find the optimal interval while NSMC requires 253 of its 260 calls.

The final experiment shows a case where EASY is significantly better than NSMC. The candidate invariant *lowBound* is an underapproximation of the reachable states. Both algorithms handle this problem quite differently and NSMC times out in all cases while EASY learns a correct invariant quickly by dropping the problematic clauses.

B. Processor

For the processor models, we described a very basic processor with a pipeline of length 3. The processor has 4 registers that can store 3-bit values. It can handle three operations: ADD, SUB and NOP. The implementations provide getter-methods for all registers and a nextStep-method to load a new operation that is given as input and execute one step of the pipeline.

TABLE I
Runtimes and CBMC calls on the processor models

Candidate Invariant	NSMC		EASY	
	Time [s]	#Calls [-]	Time [s]	#Calls [-]
proc-optimal	0.4	3	1.1	4
1-regNotEqual	10.0	62	11.6	50
2-regNotEqual	16.0	101	19.8	96
3-regNotEqual	31.0	175	28.2	142
4-regNotEqual	35.0	201	35.3	188
noResult	T/O	-	T/O	-
wrongPipeEqual	T/O	-	21.1	104

The models are sketched in Fig. 4. While the detailed model loads the input values, computes the result, and finally writes the result back in three steps, the abstract model just computes everything within the last step.

To ensure that these models behave equivalently, we block new operations that could lead to conflicts, i.e., use input or output registers that are currently used by other operations on the pipeline. When such an operation is used as input for nextStep, we put the operation NOP in the pipeline instead.

An optimal candidate invariant for this example contains the information that corresponding variables in the two models are equal, all values need to be within their valid ranges, there are no conflicts in the pipeline, and the additional variables in the detailed model contain correct values.

Splitting the described candidate invariant into clauses leads to 95 clauses. For the experiments, we modify the optimal candidate invariant *proc-optimal* by

1) Removing the equality of $1 \leq i \leq 4$ registers: *i-regNotEqual*
2) Removing the correctness of the result in the detailed model: *noResult*
3) Adding a faulty equality of an input of the second operation and the output of the third operation: *wrongPipeEqual*

The candidate invariant *proc-optimal* describes exactly all reachable states and therefore should support the equivalence check significantly. For the candidates *i-regNotEqual*, the algorithms need to detect the equality of the corresponding registers.

Removing the correctness of the result from the candidate invariant is a lot harder than removing equality of registers as the correctness of the result is complex to describe and depends on multiple factors.

Table I shows the runtime and the number of CBMC calls of NSMC and EASY for the different candidate invariants.

Proving correctness for *optimal* and the candidate invariants *i-regNotEqual* shows again, that EASY has a small overhead compared to NSMC but due to low level optimizations like the binary search becomes faster with more difficult candidate invariants.

Both equivalence checkers cannot decide equivalence for the hypothesis *noResult* as the used heuristics of the generalization are not applicable in this case and states are excluded from the candidate invariant one by one.

Finally, the underapproximation *wrongPipeEqual* timed out when run by NSMC but is decided within 21.1 seconds by EASY as EASY can easily detect the wrong clause and will not propagate it.

In summary, EASY has a little overhead that is not needed in some cases but only increases the runtime slightly compared to NSMC. However, EASY can decide equivalence in cases that NSMC cannot decide within the time limit.

VII. Conclusion

In this paper, we presented EASY, an algorithm for function equivalence checking of ESL description written in C++. The algorithm generates an inductive invariant in a style similar to PDR when two ESL descriptions are functionality equivalent or produces a counterexamples that can be used as a starting point for debugging. We proposed an implementation of EASY on top of the standard bounded model checker CBMC and presented two case studies to show the applicability of the approach.

References

[1] Bijan Alizadeh and Masahiro Fujita. Automatic merge-point detection for sequential equivalence checking of system-level and rtl descriptions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 129–144, 2007.

[2] Brian Bailey, Grant Martin, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Morgan Kaufmann/Elsevier, 2007.

[3] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference*, pages 368–371, 2003.

[4] 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.

[5] 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.

[6] Xiushan Feng and Alan J. Hu. Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification. In *Design Automation Conference*, pages 1063–1068, 2006.

[7] Alexander Finder, Jan-Philipp Witte, and Görschwin 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.

[8] 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.

[9] Alan Leung, Dimitar Bounov, and Sorin Lerner. C-to-Verilog translation validation. In *ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pages 42–47, 2015.

[10] Rajdeep Mukherjee, Daniel Kroening, Tom Melham, and Mandayam K.Srivas. Equivalence checking using trace partitioning. In *IEEE Computer Society Annual Symposium on VLSI*, pages 13–18, 2015.

[11] Niels Thole, Heinz Riener, and Goerschwin Fey. Equivalence checking on system level using a priori knowledge. In *International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, pages 177–182, 2015.

[12] Sheng Yu. *Handbook of Formal Languages: Volumne 1. Word, Languages, Grammar*, chapter Regular Languages, pages 41–110. Springer Science & Business Media, 1997.