# Debugging HDL Designs Based on Functional Equivalences with High-Level Specifications

Alexander Finder* final@informatik.uni-bremen.de
Jan-Philipp Witte* jpwitte@informatik.uni-bremen.de
Görschwin Fey*† Goerschwin.Fey@dlr.de
*University of Bremen 28359 Bremen, Germany
†German Aerospace Center 28359 Bremen, Germany

***Abstract*—The increasing complexity of circuits and systems is forcing design specifications to software-like programming languages like C. Since the conversion from software to hardware is a difficult task solved manually, bugs are frequently introduced in the HDL design. Sophisticated automated error localization and correction techniques, i.e. debugging, are a challenge.**

**In this paper a new automated method is presented for debugging hardware implementations when a software-like specification in C is given. Based on functional equivalences between software and hardware, error localization and correction are automated. We present experimental results for different types of designs and different types of faults.**

## I. Introduction

The increasing complexity of circuits and systems forces designers to move to higher software-like programming languages above the *Register Transfer Level* (RTL), like *Embedded System Languages* (ESL) or C/C++ [1]. Thus, often a "golden model" is written in software or software-like languages, like ANSI-C. The typical approach is then to convert this "golden model" used as the specification into an implementation in HDL, like VHDL or Verilog. Since there are many differences between hardware and software, a correct conversion from software to hardware turns out to be a difficult task. For instance, in hardware designs computation is inherently parallel. Clocking synchronizes the interaction between computational units that run in parallel. Verifying the resulting hardware descriptions becomes an essential task. Moreover, debugging a hardware implementation manually is time consuming and costly. Up to 80% of the overall design cost are due to verification [2] and often more than 60% of today's verification effort is consumed by debugging.

In this paper we present a new debugging approach for hardware descriptions using reference implementations either in software or HDL. The proposed method focuses on debugging untimed C programs versus timed HDL designs. Symbolic co-simulation is applied for error localization and error correction, exploiting functional equivalences between two descriptions. To correct an erroneous implementation, the specification is used to find corrections by replacing parts of the implementation with parts of the specification.

The methodology proposed in this paper is not restricted to specifications in C. Rather, each combination of C programs and HDL descriptions can be handled. Even an HDL implementation may serve as specification to correct a C program handled as implementation or two HDL descriptions can be considered.

Our implementation has no dynamic memory model such that neither pointers nor flexible sized arrays in C are supported. We also deem recursion and the usage of external libraries as unnecessary for our purposes, as the C program only serves as a specification for an HDL implementation. In general, we expect a similar way of implementation, i.e. the same algorithms, within the different abstraction levels of a design such that data and control paths are similar. This allows to utilize functional matching to reduce the complexity of equivalence checking as known from combinational equivalence checking [3] and to use this information for debug automation.

The contributions of this paper are:

- An algorithm tightly integrating verification and debugging.
- A new error localization and error correction technique based on functional equivalences.

The remainder of this paper is organized as follows: In Section II related work is reviewed. Section III introduces terminology and definitions and shows the transformation of a C program to an FSM. In Section IV the SEC approach is described. The principle of debugging faulty implementations based on functional equivalences is explained in Section V. Experimental results are presented in Section VI. The paper is concluded in Section VII.

## II. Related Work

Several approaches have been proposed to verify system-level specifications versus HDL descriptions [1], [4], [5], [6]. Vasudevan et al. apply equivalence checking [4] to verify system-level design descriptions against their implementations in RTL. However, a cycle-accurate behavior of both descriptions is assumed. In [1] a *Bounded Model Checking* (BMC) technique has been proposed that takes a C program and a Verilog implementation. The original C program is instrumented to describe the cycle-accurate behavior of the Verilog implementation. The approach is limited as loops always have to be unwound for a fixed number of times. If the required limit is unknown, it has to be determined gradually or using more sophisticated techniques, like in advanced compiler or synthesis optimization [7]. The authors in [5] target only model checking of C programs. For this purpose, C programs are modeled in the form of FSMs and model checked using techniques based on *Boolean Satisfiability* (SAT) and *Binary Decision Diagrams*. The extension of the model checking procedure to hardware designs was out of scope. Koelbl et al. [6] discuss solver technology for system-level to RTL equivalence checking. First, both models (system-level and RTL) are converted to a formal representation which is a word-level *Data Flow Graph* (DFG). Next, the two DFGs are combined for the equivalence check. For the proof engine the DFGs have to be combined such that timing differences are eliminated, which means that the problem is reduced to a cycle-accurate equivalence check.

All methods mentioned above use a monolithic approach where two design descriptions as a whole are checked for

This work was supported in part by the European Union (Project DIAMOND, FP7-2009-IST-4-248613).

equivalence. In general, verifying and diagnosing the full designs is very difficult due to capacity limitations of the reasoning engine. Various optimizations were proposed to reduce the verification problem. One optimization approach to reduce the size of the problem instance, is to extend the verification algorithm by cutpoint detection [8], [9], [10]. Cutpoints represent parts within two designs (e.g., the specification and the implementation) which are functionally equivalent. The verification procedure then can be improved by using the cutpoints as new starting points or reducing the verification problem by merging the functionally equivalent structures [3]. For many designs functional equivalences are only detectable under certain conditions, e.g., a loop condition has to be fulfilled. In [11] a first attempt is proposed to verify conditionally functional equivalent designs using an invariant generation framework. However, all these methods apply only error detection but not error localization and correction.

In order to localize and correct an erroneous design, error detection is only the first step. For hardware or software, there exist several debugging methodologies [12], [13], [14], [15], [16], [17], [18], [19]. The approach in [16] simplifies a failing test case to a minimal test case that still produces the failure by successive testing. An automated fault localization algorithm based on SAT has been introduced by Smith et al. [15]. In [17] faulty components are determined with respect to formal properties. However, in all these methods the specification is always only used for error detection and localization but never for error correction. Jobstman et al. [12] transform an erroneous design to a game and compute a correction as a strategy in this game. The tool in [13] removes parts of the software that are assumed to be erroneous and then tries to synthesize an implementation for these missing parts such that the specification is satisfied. A very simple but fast approach for design error correction is mutation [14] where the incorrect program is repeatedly mutated in different ways and every mutant is checked for correctness. Hoffmann and Kropf described in [18] an approach for automatic error correction of combinational circuits on gate-level using symbolic methods. The tool in [19] applies counterexample-guided resynthesis of erroneous circuits on gate-level based on simulation. To perform the analysis no specification is needed but only input stimuli with expected output responses. However, all of these techniques focus only on the correction of a design at one particular description level and do not exploit functional equivalences between an erroneous hardware implementation and its high-level reference specification.

## III. Preliminaries

In order to model C programs and corresponding HDL implementations *deterministic Finite State Machines* (FSMs) are used. An **FSM** is defined as a quintuple $M = (I, S, S_0, \delta, F)$, where $I$ is the finite, nonempty input alphabet, $S$ is a finite, nonempty set of states, $S_0$ is a finite set of initial states, $S_0 \subseteq S$, $\delta$ is the state-transition function: $\delta : S \times I \rightarrow S$, and $F$ is the set of final states, $F \subseteq S$.

Each state $s \in S$ is represented by a valuation of a finite set of state variables $\{v_1, \ldots, v_n\}$, where $|s| = n$. The state-transition function $\delta$ determines the next state of a machine $M$ based on its inputs and the current state. Considering C programs and HDL implementations, each application of $\delta$ transfers the current state of the program (circuit) to the next state. In other words, $\delta$ encodes the assignment statements of the C program and the combinational part of the circuit, respectively. To indicate the time step we use the superscript $t$, like e.g., $s^t$ denotes the state $s$ at time step $t$.

Functional equivalences between two FSMs $M_0$ and $M_1$ are considered as cutpoints. A **cutpoint** $c$ is given by a tuple $(v^k_{i_{M_0}}, v^l_{j_{M_1}})$ of state variables associated to a time step, where $v^k_{i_{M_0}} \in s^k_{M_0}$ and $v^l_{j_{M_1}} \in s^l_{M_1}$ are functionally equivalent at the respective time step. We define a partial order for pairs of state variables associated to time steps $(v^k_{i_{M_0}}, v^l_{j_{M_1}})$ and $(v^u_{i_{M_0}}, v^w_{j_{M_1}})$ as

$$(v^k_{i_{M_0}}, v^l_{j_{M_1}}) \leq (v^u_{i_{M_0}}, v^w_{j_{M_1}}) \Leftrightarrow (k < u) \vee (k = u \wedge l \leq w).$$

The *first* pair $(v^k_{i_{M_0}}, v^l_{j_{M_1}})$ in a set of pairs $C$ is defined by $\forall (v^u_{i_{M_0}}, v^w_{j_{M_1}}) \in C : (v^k_{i_{M_0}}, v^l_{j_{M_1}}) \leq (v^u_{i_{M_0}}, v^w_{j_{M_1}})$.

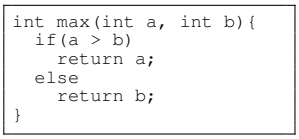
To compare a specification given in C with an implementation in VHDL or Verilog, we use FSMs as a common intermediate representation for these descriptions. Modeling both descriptions as FSMs enables us to check C programs which do not have to be cycle-accurate with respect to their hardware implementations. Hence, specification and implementation do not need to use the same number of time cycles for the computation. In addition, loops do not have to be unwounded in advance but this is done implicitly when unrolling the FSM such that the depth does not have to be determined gradually or be computed using more sophisticated techniques. Since the modeling of synchronous sequential circuits as FSMs is well-known, we only describe how we model a C program as an FSM, similar to [5].

To create an FSM $M$ from an ANSI C program $P$, which serves as the specification, each method of $P$ is split into a finite set of nonempty basic blocks $B$ and transformed to *Static Single Assignment* (SSA)-form [20]. Each basic block $b_i \in B$ consists of a sequence of assignment statements that is followed either by a conditional jump or a direct jump to another basic block $b_j$. All statements within a basic block are executed sequentially without interruption. Each basic block has only one entry point and one exit point. There are two special blocks in each method of $P$, the first block $b_{entry}$ is the entry point and a common return block $b_{exit}$ is the exit point.

In SSA-form statements in $P$ are split into statements with three operands at the maximum and each variable is assigned exactly once. Each occurrence of an existing variable in $P$ is assigned to a unique version number in SSA-form. If a variable in $P$ is written in different blocks, a so-called *PHI*-function determines which one should be used in the current block.

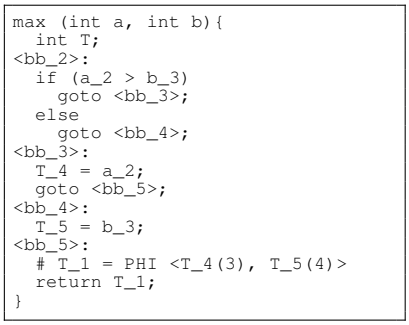
**Example 1.** *In Figure 1 a C program is shown and its corresponding SSA-form in Figure 2. The original program is split into four basic blocks (*`bb_2` *to* `bb_5`*). A temporary variable* `T` *is introduced to store the result. Each occurrence of the new variable and the input variables (*`a` *and* `b`*) on the left side of an assignment statement is made unique by incrementing the index of the affected variable. In* `bb_5` *the* PHI*-function determines which version of* $T$ *should be taken for the return value.*

In order to transform a program $P$ to an FSM $M$, we provide each method of $P$ with a program counter $PC$ to control the state transitions of $M$. By this, the execution of each basic block $b \in B$ of $P$ represents a state transition such that all variable changes within a basic block are mapped to a state $s \in S$ of $M$. Since each variable of a basic block $b$



```
int max(int a, int b){
  if(a > b)
    return a;
  else
    return b;
}
```

Fig. 1. C program.



```
max (int a, int b){
  int T;
<bb_2>:
  if (a_2 > b_3)
    goto <bb_3>;
  else
    goto <bb_4>;
<bb_3>:
  T_4 = a_2;
  goto <bb_5>;
<bb_4>:
  T_5 = b_3;
<bb_5>:
  # T_1 = PHI <T_4(3), T_5(4)>
  return T_1;
}
```

Fig. 2. SSA form.

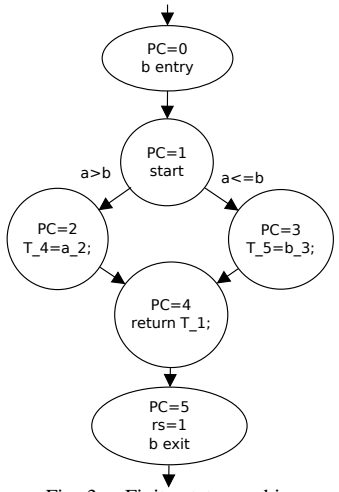



Fig. 3. Finite state machine.

is mapped to a unique version number in the SSA-form, each SSA-variable is directly mapped to a state variable $v_i^t$ of a state $s^t$. The value assigned to the state variable is implied by the $PC$ that substitutes the functionality of the *PHI*-function.

**Example 2.** *In Figure 3 the FSM $M$ for the program in Figure 2 is shown. Each basic block of Figure 2 is mapped to a state in $M$. Additionally, the basic blocks $b_{entry}$ and $b_{exit}$ are added representing the entry and exit point of method* `max()`.

Typically the specification and the implementation need different numbers of transitions to produce the same results. To resolve this, we add a special so-called *ready state* variable $rs_{spec}$ to the FSM representation of the specification which defines when a final state $s^t \in F$ is reached. The variable $rs_{spec}$ is set to $rs_{spec} = 1$ when $PC$ reaches the last basic block of $P$ and otherwise to $rs_{spec} = 0$. Similarly, the implementation has an appropriate variable $rs_{impl} = 1$ or a fixed number of transitions that has to be specified by the user.

**Example 3.** *In Figure 3 a final state is reached if $PC = 5$ where the ready signal $rs = 1$. This signals that the computation has been completed. In this case, e.g., the return value is handled as an output variable which can be compared to the corresponding output variable of the implementation.*

## IV. Verification

Our correction procedure is tightly integrated with the verification procedure exploiting functional equivalences, called cutpoints.

### A. Sequential Equivalence Checking

The FSMs modeled for a C program and the corresponding HDL implementation serve as a starting point for verification and diagnosis. For a sequential equivalence check of the primary outputs only final states have to be verified to be equivalent. After the creation of the two FSMs $M_{spec} = (I_{spec}, S_{spec}, S_{0_{spec}}, \delta_{spec}, F_{spec})$ and $M_{impl} = (I_{impl}, S_{impl}, S_{0_{impl}}, \delta_{impl}, F_{impl})$ for SEC the inputs of the specification have to be mapped to the inputs of the implementation. This is either done automatically if variable names are matching or has to be done manually, otherwise. The inputs of both descriptions are set to be equal. Analogously, we proceed with the outputs.

The idea for the equivalence checking algorithm is as follows: Both FSMs are unrolled in parallel and in each time step it is checked if one of the FSMs can reach a final state given by $rs_{spec} = 1$ or $rs_{impl} = 1$. In this case, only the other FSM is traversed further to find the corresponding final state. If for both FSMs a final state has been found, these are checked for equivalence. If the final states are not equivalent, a counterexample is returned; otherwise the algorithm continues. As reasoning engine for SEC a SAT- or SMT-solver can be used.

### B. Finding cutpoints

To reduce the state space to be verified, cutpoints between the specification and the implementation are determined. A set of cutpoints $C$ is given by pairs of state variables ($v_{i_{spec}}^t$, $v_{j_{impl}}^u$) which have the same value under equal input assignments. We use these cutpoints to simplify the verification problem in two aspects:

- Additional constraints are added to the problem instance to be solved by the SAT- or SMT-solver which, though enlarging the verification instance, reduce the search space. These constraints force identical values to the pair of state variables of each cutpoint.
- Cutpoints found can be used as new starting points, removing the input cone. Due to reachability aspects, in our approach, only the input cone of the implementation is removed in order to avoid the creation of unreachable states. By this, we assure that only parts of the implementation are removed that have no further effect on parts of the circuit which are not covered by the output cone of the cutpoints. Removing the input cone reduces the problem instance to be verified.

Since we keep the input cone of the specification false negatives are avoided which could occur otherwise, if the cutpoints are used as new starting points.

To find potential cutpoints, random input simulation is applied in a first step. Both descriptions are simulated for a given number $n$ of time steps in parallel with identical random input values in the first cycle. Each state variable $v_{i_{spec}}^t$ of each state $s_{spec}^t$ in $M_{spec}$ is compared with every state variable $v_{j_{impl}}^u$ in each state $s_{impl}^u$ in $M_{impl}$ for equivalence. State variables $v_{i_{spec}}^t$, $v_{j_{impl}}^u$ are assigned to equivalence classes where all variables are collected which have the same value. Variables $v_{i_{spec}}^t$ or $v_{j_{impl}}^u$ which have different values, are split into different equivalence classes. After $n$ simulation runs only state variables that remain in the same equivalence class, are considered as potential cutpoints.

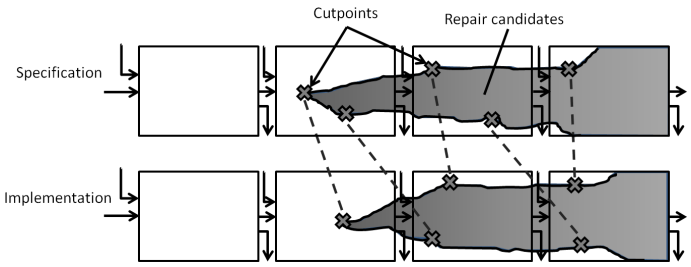



Fig. 4. Principle of correction based on functional equivalences

Because every state variable in the specification is compared to state variables in the implementation for every time-step in the simulation, there may be many potential cutpoints. The reason is that a variable often keeps its value over many time-steps. For two variables, which have more than one potential cutpoint, there is only the first chosen for verification.

**Example 4.** *If two variables have the same value for five time steps in a sequence, there are 25 cutpoints determined by simulation between these variables. Thus, out of the 25 potential cutpoints only the first one is chosen.*

Because the results may differ for various numbers of simulation runs and the randomly generated input values, the potential cutpoints chosen for verification may differ. In general, the number of potential cutpoints is significantly reduced with the approach of selecting only one cutpoint in a sequence of potential cutpoints, e.g., from more than 340k to less than 40 for the `tcas` benchmark (see Section VI).

Since simulation is not complete, the set of cutpoints $C$ found has to be verified in a next step which is integrated into the SEC procedure described above. Always when a state is unrolled which has one or more state variables marked as potential cutpoints by simulation, the cutpoints are verified using a formal reasoning engine. This results in a set of verified cutpoints $C_{verified}$.

## V. Correction

In this section we describe our approach for an automated localization and correction of bugs integrated into the verification procedure. Figure 4 illustrates the principle. If the implementation is not equivalent to the specification, the proposed technique for finding cutpoints computes a frontier of cutpoints. The cutpoint frontier is a subset $C_f \subseteq C$ of all verified cutpoints $C_{verified}$ computed by the algorithm described in Section IV-B. The two state variables in any pair $(v^t_{i_{spec}}, v^u_{j_{impl}}) \in C_f$ have no further state variables of verified cutpoints in their output cone.

Next the set of **repair candidates** $rc \in RC$ is computed where a repair candidate is a tuple $(v^w_{l_{impl}}, v^s_{k_{spec}})$, i.e. replace variable $v^w_{l_{impl}}$ from the implementation with variable $v^s_{k_{spec}}$ from the specification. In order to find a repair candidate $rc$ for the implementation, we start with the first cutpoint $(v^t_{i_{spec}}, v^u_{j_{impl}}) \in C_f$ which has no further cutpoints $c \in C_{verified}$ within its output cone. We get the first cutpoint $(v^t_{i_{spec}}, v^u_{j_{impl}})$ with respect to the partial order of the cutpoints in $C_f$.

Since the specification and the implementation should be functionally equivalent further cutpoints in subsequent time steps are expected. Starting with the first variable, all state variables $v^w_{l_{impl}}$ within the output cone of $v^u_{j_{impl}}$ are considered as *repair candidates* $rc \in RC$. To correct the implementation, variables $v^w_{l_{impl}}$ are subsequently substituted by state variables $v^s_{k_{spec}}$ of the specification such that $v^w_{l_{impl}} \equiv v^s_{k_{spec}}$. The replacement procedure starts with the first variable in the output cone of the cutpoint variable $v^t_{i_{spec}}$. Then again simulation is performed to decide if new cutpoints can be computed. If this is not the case, the repair candidate $rc = (v^w_{l_{impl}}, v^s_{k_{spec}})$ is discarded and the procedure is repeated for the next repair candidate. Finally, the repair candidates are verified. If the verification of a repair fails, the next repair candidate in $RC$ is chosen. In case that the verification holds, a **valid repair** with respect to the repair candidate $rc$ is found and applied to the implementation. Finally, the set of cutpoints is updated with respect to the change in the implementation and the method proceeds as before.

**Algorithm 1** Correction

```
1: const maxDepth
2: FSM M_spec, M_impl
3: C = getCutpoints(M_spec, M_impl), C_verified = ∅
4: for i = 0...maxDepth and
         not allFinalStatesReached(M_spec, M_impl) do
5:   C_verified.add(verifyCutpoints((v^{0...i}_{x_spec}, v^{0...i}_{y_impl})))
6:   if canReachReady(M_spec, i) then
7:     for j = i...maxDepth do
8:       C_verified.add(verifyCutpoints((v^{0...i}_{x_spec}, v^{0...j}_{y_impl})))
9:       if canReachReady(M_impl, j) then
10:        if notEquivalent(M_spec, i, M_impl, j, C) then
11:          C_f = computeCutpointFrontier(C_verified)
12:          RC = computeRepairCandidates(C_f)
13:          for all (v^w_{l_impl}, v^s_{k_spec}) in RC do
14:            if verifyRepairCandidate((v^w_{l_impl}, v^s_{k_spec})) then
15:              updateCutpoints(C)
16:              break
17:            end if
18:          end for
19:        end if
20:      end if
21:    end for
22:  end if
23: end for
```

### A. Algorithm

The entire debugging procedure is sketched in Algorithm 1. First, a set of cutpoints $C$ between $M_{spec}$ and $M_{impl}$ is computed using simulation. Next, the verification is started with the two nested loops in Lines 4 and 7. These loops are executed as long as there are further final states, or the given maximal depth is reached. We determine if all final states have already been covered by property checking whether there exist input assignments for which no final state, i.e. $rs_{spec=1}$ or $rs_{impl} = 1$, has been reached yet. For each step, both FSMs are checked for final states in Lines 6 and 9. In addition, in each time step we check whether there are cutpoints until that time step and verify all cutpoints $(v^t_{x_{spec}}, v^u_{y_{impl}})$ where $t \leq i$ and $u \leq i$ in Line 5 (and $u \leq j$ in Line 8) starting with the first cutpoint.

If there is a counterexample in the verification step (Line 10), which uses the currently verified cutpoints as additional constraints, the repair is started. To get a repair, we compute the cutpoint frontier $C_f$ for all verified cutpoints $C_{verified}$ in function `computeCutpointFrontier()` in Line 11 as explained above. Next, we compute the set of *repair*

*candidates* $RC$ in Line 12 which are pairs of variables $(v^w_{l_{impl}}, v^s_{k_{spec}})$ in the output cone of variables in $C_f$. For this we combine each state variable $v^w_{l_{impl}}$ of the implementation with each state variable $v^s_{k_{spec}}$ in the specification. Next, each repair candidate is evaluated by simulation. If a replacement of $v^w_{l_{impl}}$ by a $v^s_{k_{spec}}$ leads to further cutpoints between both descriptions the repair candidate is kept within $RC$. Otherwise it is discarded. By default, we consider all combinations of state variables between the specification and the implementation within five time steps as possible repair candidates but the search space is adjustable by the user.

Beginning with the first repair candidate $(v^w_{l_{impl}}, v^s_{k_{spec}})$ we formally check whether it is a valid repair (Lines 13-18). In this case, we apply the repair to the implementation and update the set of cutpoints (Line 15) based on the changed behavior of the implementation. This is done by applying again simulation for cutpoint detection and adding all new cutpoints to $C$. The algorithm proceeds with Line 7. The method terminates either when all final states of both descriptions have been verified, the maximal depth has been reached, or when no repair could be found.

### B. Discussion

The algorithm depends on the existence of functional equivalences between two descriptions. Otherwise an accurate computation for a repair becomes difficult and the algorithm most probably would end up with a repair at the primary outputs. However, typically functional equivalences between a specification and an implementation are present if no different implementations for an algorithm are used.

As user-defined parameter for the methodology we have the maximal depth, which serves as an upper bound for how many steps the FSMs should be verified at the maximum. In addition, the parameter is used for cutpoint detection to have an upper bound for simulation. In case of a hardware implementation with constant runtime, an additional parameter can be given to the procedure, which defines when a final state is reached for the implementation. Furthermore, the number of simulation runs can be given, or specific input stimuli to support simulation in cutpoint detection. For instance, while detecting repair candidates we apply the counterexample determined by the formal engine and additional stimuli which are similar. This is done to create a cutpoint frontier close to the fault within the implementation.

If a repair is found, the implementation is not resynthesized by replacing parts of the implementation with parts from the specification. Instead, a suggestion is given, how to fix the implementation of the design, e.g., "replace the computation of signal $s$ in the implementation by the computation of variable $v$ from the specification". By this means, the proposed repair algorithm is not restricted to a particular subset of fault types.

## VI. Experimental Results

The experiments have been carried out on an *Intel(r) Core(tm) i5-2500K processor* with 8 GB of memory. For verification, the SAT-solver MiniSat2 [21] has been used as the underlying reasoning engine. We restricted the equivalence check for all benchmarks to a time limit of 60 minutes per instance. For the cutpoint detection described in Section IV-B, 100 simulation runs have been carried out for each design. In order to create FSMs out of the C programs considered, we need the corresponding SSA-forms. For this purpose, we hook into the internal data structure of the *GNU Compiler Collection* (GCC), called *GIMPLE* [22].

TABLE I
Benchmarks

| name | LOC | | #reg | #gates | unroll depth |
|---|---|---|---|---|---|
| | spec | impl | | | |
| pipeline3 | 28 | 56 | 11 | 3704 | 10 |
| pipeline6 | 43 | 95 | 20 | 7119 | 15 |
| tcas | 175 | 175 | 37 | 17488 | 39 |
| dlx-decode | 23 | 23 | 25 | 3904 | 17 |

TABLE II
Cutpoint detection

| benchmark | #potential cp | #selected cp | simulation time | #cp | verification time |
|---|---|---|---|---|---|
| pipeline3 | 296 | 9 | 1s | 8 | 16s |
| pipeline6 | 920 | 13 | 1s | 12 | 1200s |
| tcas | 346643 | 38 | 8s | 28 | 132s |
| dlx-decode | 21296 | 40 | 1s | 37 | 15s |

The presented procedures have been evaluated on three different sets of benchmarks. The first set contains two pipelines with different numbers of stages in the hardware implementation. Circuit `pipeline3` is a 3 stage pipeline with one addition, one comparison and dependent on the result either another addition or subtraction per stage: `pipeline6` has the same structure but contains 6 stages. The second set contains an implementation from the Siemens benchmark suite [23] of the *Traffic Collision Avoidance System* (TCAS). This benchmark is a simplified version of TCAS II which can give an up or down resolution advisory to an aircraft pilot to avoid collisions between aircrafts. The third set consists of a `DLX` microprocessor instruction set architecture based on the WinDLX [24]. In particular, the `decode` step is examined, which is implemented with shift and bitwise operators and can distinguish between three different types of instructions. The 32-bit wide input instruction is decomposed into two, four, or five parts, dependent on the type of instruction.

Table I gives an overview of the specifications and implementations. Column `LOC` contains the *Lines Of Code* for the C specification (`spec`) and the HDL implementation (`impl`). In column `#reg` the number of registers and in `#gates` the number of gates in the synthesized hardware implementation are shown. Column `unroll depth` gives the number of states, the FSM has to be unrolled at the minimum to reach all final states. Since specification and implementation are in C for `tcas`, the number of code lines is equal. A simple transformation of the `tcas` design in a gate-level netlist resulted in 37 flip flops and more than 17k gates.

In Table II we show how many cutpoints are detected by our procedure described in Section IV-B. In column `#potential cp` the number of potential cutpoints detected by simulation are given. Column `#selected cp` shows the number of cutpoints selected for verification (see Example 4 in Section IV-B). In column `simulation time` the runtime of the simulation-based cutpoint detection is given. In the last two columns of Table II the number of verified cutpoints (`#cp`) and the verification time for the verification procedure are shown. The runtime for finding cutpoints always lies between 1 and 9 seconds. Most of the cutpoints found by simulation were verified to be real cutpoints, i.e. the implementation and the specification are equivalent at these points. However, for `tcas` 10 of 38 cutpoints have been sorted out during verification. The verification time for all cutpoints lies between 15 seconds (`dlx-decode`) and 20 minutes (`pipeline6`).

TABLE III
RUNTIMES FOR EQUIVALENCE CHECK

| benchmark | time |
|---|---|
| pipeline3 | 8s |
| pipeline6 | 974s |
| tcas | 169s |
| dlx-decode | 25s |

TABLE IV
CORRECTION

| benchmark | #rep. cand. | repair time |
|---|---|---|
| tcas1 | 19 | 20s |
| tcas3 | 7 | 8s |
| tcas4 | 6 | 7s |
| tcas12 | 3 | 5s |
| tcas22 | 6 | 7s |
| tcas28 | 108 | 102s |
| tcas29 | 1 | 2s |
| tcas30 | 1 | 3s |
| tcas41 | 6 | 7s |
| pipeline3a | 1 | 3s |
| pipeline3b | 1 | 3s |
| pipeline6a | 7 | 22s |
| pipeline6b | 4 | 15s |
| dlx-decode1 | 57 | 470s |
| dlx-decode2 | 82 | 737s |

Next, in Table III runtimes for the equivalence check of correct versions for the benchmarks in Table I are shown. If erroneous implementations are considered for debugging, the first counterexample has always been returned in a few seconds.

In Table IV experimental results for the correction algorithm described in Section V are shown. In column `#rep. cand.` the number of repair candidates is shown which have been tried for a potential repair (see Algorithm 1) until a valid repair has been found. The repair time, given in the last column, represents the time to find a repair for correcting a design.

For the `tcas` benchmark from Siemens faulty versions are included within the benchmark suite which are considered to be realistic design errors. In Table IV we provide results for a selection of faulty versions. In each case single operator faults or operand faults are repaired. Versions with multiple faults exceeded the runtime limit for the debugging method. Better heuristics for the cutpoint detection step can speed up this process. For the pipeline benchmarks in each case two faulty versions (`a` and `b`) have been created where the faults have been injected randomly. Similarly we proceeded for the `dlx-decode` method where we also randomly injected faults. Because most of the work is done in parallel in the decode method, the cutpoint frontier is wide spread, such that the runtime strongly depends on the order in which the candidates are selected. Consequently, although the benchmark is not so large, it has the highest repair time.

The repair candidates for the faulty designs are chosen subsequently from the variables in the output cone of the cutpoint frontier as sketched in Algorithm 1. For each of the designs a repair has been computed within less than 13 minutes. In most cases even less than a minute is needed in order to provide a suggestion to the designer how to fix the implementation given in HDL by providing the corresponding code of the specification.

## VII. CONCLUSION

Verifying hardware implementations versus software specifications is a hard problem. This work shows how to exploit functional equivalences (cutpoints) between the specification and implementation for error localization and correction. The proposed methodology successfully computed repairs on different sets of benchmarks and different types of faults in a range of few seconds to less than 15 minutes.

## REFERENCES


[1] E. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and Verilog programs using bounded model checking," in *Proceedings of Design Automation Conference*, 2003, pp. 368–371.

[2] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Proceedings of International Conference on Computer Aided Verification*, 2008, pp. 5–10.

[3] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of Design Automation Conference*, 1997, pp. 263–268.

[4] S. Vasudevan, J. Abraham, V. Viswanath, and J. Tu, "Automatic decomposition for sequential equivalence checking of system level and RTL descriptions," in *Proceedings of Formal Methods and Models for Co-Design*, 2006, pp. 71–80.

[5] F. Ivanicic, I. Shlyakhter, A. Gupta, M. K. Ganai, V. Kahlon, C. Wang, and Z. Yang, "Model checking c programs using f-soft," in *Proceedings of International Conference on Computer Design*, 2005, pp. 297–308.

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

[7] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Coordinated transformations for high-level synthesis of high performance microprocessor blocks," in *Proceedings of Design Automation Conference*, 2002, pp. 898–903.

[8] B. Alizadeh and M. Fujita, "Automatic merge-point detection for sequential equivalence checking of system-level and RTL descriptions," in *Proceedings of International Conference on Automated Technology for Verification and Analysis*, 2007, pp. 129–144.

[9] X. Feng and A. Hu, "Early cutpoint insertion for high-level software vs. RTL formal combinational equivalence verification," in *Proceedings of Design Automation Conference*, 2006, pp. 1063–1068.

[10] C. Karfa, C. Mandal, D. Sarkar, S. Pentakota, and C. Reade, "A formal verification method of scheduling in high-level synthesis," in *Proceedings of International Symposium on Quality Electronic Design*, 2006, pp. 71–78.

[11] J. Baumgartner, H. Mony, M. Case, J. Sawada, and K. Yorav, "Scalable conditional equivalence checking: An automated invariant-generation based approach," in *Proceedings of International Conference on Computer-Aided Design*, 2009, pp. 120 –127.

[12] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem, "Finding and fixing faults," *Journal of Computer and System Sciences*, vol. 78, no. 2, pp. 441–460, 2012.

[13] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia, "Combinatorial sketching for finite programs," in *Proceedings of Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 404–415.

[14] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proceedings of International Conference on Software Testing, Verification and Validation*, 2010, pp. 65–74.

[15] A. Smith, A. Veneris, M. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1606–1621, 2005.

[16] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[17] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 6, pp. 1138–1149, 2008.

[18] D. W. Hoffmann and T. Kropf, "Efficient design error correction of digital circuits," in *Proceedings of International Conference on Computer Design*, 2000, pp. 465–472.

[19] K.-h. Chang, I. L. Markov, and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 1, pp. 184–188, 2008.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[21] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[22] Free Software Foundation, Inc., *GNU Compiler Collection (GCC) Internals*, http://gcc.gnu.org/onlinedocs/gccint/, 2010.

[23] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[24] WinDLX Home Page. (2012) http://cs.uns.edu.ar/ jechaiz/arquitectura/windlx/DLXinst.html.