Path Directed Abstraction and Refinement for SAT-Based Design Debugging

Brian Keng, *Graduate Student Member, IEEE,* and Andreas Veneris, *Senior Member, IEEE*

***Abstract*—Functional verification has become one of the most time-consuming tasks in the VLSI design flow accounting for up to 57% of the total project time. The largest component of this task is that of design debugging due to its resource intensive manual nature. With the ever growing size of modern designs and their error traces, the complexity of the debugging problem poses a great challenge to automated debugging techniques. To overcome this challenge, this work introduces a novel path directed abstraction and refinement algorithm for design debugging to manage excessive error trace lengths. A sliding window of the error trace is iteratively analyzed in a time-windowing framework, which is made possible by the use of the path directed abstraction. This abstraction forms a concise approximation of non-modeled parts of the error trace, while simultaneously providing an efficient representation for refinement. The result is an algorithm that dramatically reduces the memory requirements of debugging, while mitigating the incomplete results of past techniques. Experimental results on industrial designs with long error traces show that the proposed approach can analyze traces that are 64.6% longer while simultaneously decreasing peak memory usage compared to previous work.**

***Index Terms*—Debugging, Diagnosis, Verification, Abstraction, Refinement, RTL, VLSI**

I. Introduction

Modern System-on-Chip (SoC) designs have grown into enormously complex systems consisting of tens of millions of gates and multiple embedded processors. This sudden growth of complexity and size has put a large strain on our ability to verify the functional correctness of these designs. As a result, functional verification takes up to 57% of the total project time, inevitably leading to increased cost and lengthier time-to-market [1]. To mitigate the cost and time associated with this process, several impressive innovations in the past decade have been developed in order to cope with the growing size of modern SoC designs. Novel methodologies [2], [3], languages [4], [5] and algorithms [6]–[9] have all contributed to the dramatic increase in verification efficiency. Despite this effort, one area that has seen significantly less progress is that of design debugging. This task still accounts for up to 32% of the total verification time resulting in a major bottleneck in the design cycle [1].

Design debugging is the process of identifying which components of a design are erroneous after a failure has been detected during verification. Current state-of-the-art industrial debugging tools aid this process by allowing the engineer to manually navigate the failure. Tools [10] such as graphical waveform viewers, design tracing and navigation, and "what-if" analysis engines provide the user with intuitive methods for navigating the design complexity in order to determine the root-cause of failure. In contrast, automated design debugging techniques [11]–[14] aim to identify erroneous components that could potentially cause the observed functional failure without any user intervention. However, the applicability of these techniques has struggled to keep pace with the massive design sizes and error trace lengths of modern SoCs. In particular, contemporary automated debugging techniques [13], [15] make extensive use of formal methods and time-frame expansion to model the sequential behavior of the design. For current industrial design sizes and error trace lengths, the complexity of the debugging problem may increase dramatically leading to performance issues deeming these techniques sometimes impractical.

Tackling these two major components of debugging complexity (*i.e., design size* and *error trace length*) has been the focus of recent research efforts. Abstraction and refinement techniques [16], [17] have been developed to tackle the design size aspect of debugging complexity. These methods perform analysis on an abstract model of the design that is significantly smaller than the concrete one. Subsequently, they iteratively refine the abstraction to determine a sufficient set of components to solve the debugging problem. Orthogonally, time-windowing [18], [19] aims to deal with the excessive error trace length aspect of the debugging complexity. Instead of analyzing the entire error trace at once, they iteratively analyze a sliding window of time-frames along its length. To properly model the propagation of the error, time-frames not included in the current window are approximated using various methods. The use of approximation greatly reduces the memory requirements of automated debugging but it can also lead to poor resolution. This translates to the algorithm marking too many additional components as potentially erroneous, negating the benefits of localization in the first place.

In this work, we introduce a novel abstraction and refinement algorithm for design debugging built upon a time-windowing framework to manage excessive error trace lengths [20]. The key innovation of the algorithm is a *path-directed* abstraction that conceptually represents relevant struc-

Manuscript received September 4, 2012; revised December 24, 2012. Accepted for publication on April 18, 2013. This paper was recommended by Associate Editor S. Seshia.

Brian Keng is with the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (email: briank@eecg.toronto.edu).

Andreas Veneris is with the Department of Computer Science and the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 3G4, Canada (email: veneris@eecg.toronto.edu).

tural circuit paths in the time-frame expanded circuit. This abstraction provides a concise approximation of non-modeled time-frames while simultaneously providing an efficient representation for refinement, which consists of adding back omitted structural circuit paths. The result is an algorithm that dramatically reduces the memory requirements of debugging, while mitigating the incomplete results of past time-windowing techniques.

Due to the inherent iterative nature of the algorithm, performance remains the crucial issue behind its practical implementation. This paper additionally introduces several key optimizations to the basic refinement strategy that significantly reduce the number of iterations needed to achieve complete results. However, in certain worst case scenarios, the algorithm may still suffer from an excessive number of refinement steps. To mitigate this worst case scenario, a flexible path-directed algorithm is presented that allows a trade-off between performance and resolution by limiting the amount of refinement steps per problem instance. This novelty is shown experimentally to dramatically reduce the run-time requirements while having minor effects on the final resolution.

An extensive set of experimental results on large hardware designs from OpenCores [21] and industrial partners illustrates the benefits of this work. The proposed approach can analyze error traces that are 64.6% longer while simultaneously using significantly less memory when compared to previous work. Additionally, the refinement strategy of the flexible algorithm is able generate only 1.5% additional spurious solutions compared to 114.7% when no refinement is used.

The remaining sections are as follows. Section II presents background material. Section III and Section IV describe the proposed approach and refinement improvements, respectively. Section V presents a new flexible algorithm that allows a trade-off between performance and quality of results while Section VI presents experimental results. Finally, Section VII concludes this work.

II. Preliminaries

A. Design Debugging

Different types of debugging (or diagnosis) are performed at various stages of the design cycle based upon the type of malfunction [22]. In the post-silicon stage, the implementation is a fabricated silicon chip with extremely limited observability and controllability. *Fault diagnosis* [12] aims to find manufacturing defects in these chips using *fault models* (*e.g.,* stuck-at faults) that formally describe how a fault alters design behavior. While *(post-)silicon debug* techniques [22] aim to increase observability and controllability in order to help the engineer determine the cause of functional errors that escape pre-silicon verification in a silicon prototype.

In contrast, pre-silicon stages of design (*e.g.,* RTL) utilize simulation and formal techniques to detect functional errors, which have high observability and controllability. In this stage of design, techniques for automated logic or *design debugging* [13], [15] aim to find the root-cause of functional errors by using *error models* to formally describe how an error can alter design behavior. Although *fault models* can be used to model functional design errors [23], modern techniques for design debugging rely on model-free SAT-based methods [13]. These SAT-based techniques model the error as an arbitrary non-deterministic (model-free) function that does not make any assumptions on the specific type of fault/error. In this paper, we focus on design debugging and not fault diagnosis or post-silicon debug.

A design component is said to be a *suspect*, if changing that component's functional behavior can fix an observed failure detected during verification. Given a specification (*e.g.* SystemVerilog, SystemC, Matlab, etc.), its synthesizable implementation (*e.g.* RTL, gate-level netlist) and an input vector, automated design debugging methods aim to return all such suspects. More specifically, these techniques require a logic netlist derived from the design implementation, and an *error trace* consisting of an initial state assignment, input vector for each clock cycle, as well as an *expected* output vector derived from the specifications for each clock cycle. The expected output vector must differ from the behavior of the implementation under the initial state and input vector (or else there is no failure detected).

B. SAT-based Design Debugging

SAT-based design debugging [13] formulates the design debugging problem as a SAT instance for a given fixed number of errors. This construction formally models the unrolled design, error trace and, additionally, error models to identify locations of potential functional errors. Due to the output value(s) mismatch between the design and expected outputs for a given error trace, this instance will be UNSAT when all error models are off. However when an error model is on, it effectively replaces the associated component with an arbitrary non-deterministic function. If this non-deterministic function can fix the observed failure, then the entire instance will be SAT. Thus, each solution of this SAT instance corresponds to set of suspects that can potentially explain the observed failure.

More formally, let T describe the transition relationship of a sequential circuit C in conjunctive normal form (CNF). And let X^i, Y^i and S^i denote a predicate in CNF on the primary inputs, primary outputs and state elements of C for the i^{th} clock cycle, respectively, which encodes the respective values from the error trace. These can be encoded by a conjunction of unit clauses for each of the respective pin/state values.

The SAT instance is constructed in several steps. First, an *error model* is added for each design component (*e.g.,* gate, module, etc.) by augmenting its corresponding clauses in T. This is done by generating a new *suspect variable* (e_i), and adding it to each clause corresponding to that design component. When a suspect variable is active, it allows the component's outputs to be free, effectively encoding an arbitrary non-deterministic (model-free) function. Otherwise, the component's behavior remains unchanged. We denote this enhanced transition relationship by T_{en}. Note that this technique can be used to model RTL errors by adding one suspect variable to each clause corresponding to a RTL construct such as a Verilog assignment or module [24]. The mapping between RTL construct and clauses can be obtained by using a standard

RTL frontend to translate RTL to gates, and subsequently gates to clauses.

Next, T_{en} is replicated for the length of the error trace (*i.e.,* time-frame expanded), where each copy is labeled T^i_{en} for the i^{th} copy. During this process, the suspect variables are not replicated because each one of them corresponds to a single, potentially erroneous, design component regardless of the underlying time-frame. Additionally, constraints for the initial state (S^u), vector of inputs (X^i) and vector of expected or correct outputs (Y^i) from the error trace are added. Finally, cardinality constraints [25] (denoted by $\Phi(N)$) are used to limit the number of active suspect variables to exactly N, indicating the search for exactly N simultaneous errors in the circuit. The following equation models an error trace indexed from cycle u to cycle v:

$$Debug^v_u(N) = S^u \wedge \Phi(N) \wedge \bigwedge_{i=u}^{v} (X^i \wedge Y^i \wedge T^i_{en}) \quad (1)$$

A *solution* to this instance is defined to be the set of active suspect variables in a SAT assignment, which corresponds to the associated set of suspects that can explain the observed failure. Note that in Eq. 1, the observed failure on a primary output must occur within cycles u to v in order to generate a mismatch on the outputs between the expected and erroneous circuit behavior. If this is not the case, the equation is trivially satisfiable when $N = 0$ (and thus for all $N >= 0$) indicating that the failure cannot be observed in the modeled window and an extended error trace must be used.

C. Time Diagnosis and Time-Windowing

An alternate formulation of the debugging problem, known as *time diagnosis* [26], can be generated by using a variation of Eq. 1. The key difference is that when unrolling the enhanced transition relationship, the suspect variable for a component is shared only within a fixed time-window of clock cycles rather than the entire error trace. That is, each component has several associated suspect variables, where each variable corresponds to a different time-window. This models the excitation of an error within a time-window, but not across time-windows. This formulation has been shown [26] to dramatically reduce problem complexity while still being able to model real-life bugs with long error traces. Due to its relation to this work, we provide more detail on the underlying formulation.

To model a single time-window Eq. 1 can be used. However, as previously mentioned, if the failure is not observed within this window, the instance will be trivially satisfiable. To remedy this issue, one can model subsequent suffix time-windows so that the observed failure at the mismatched primary outputs is included. For time-windows of width w, we can model the p^{th} window in the sequence by the following equation:

$$W_p = \bigwedge_{i=p\cdot w}^{(p+1)\cdot w-1} (X^i \wedge Y^i \wedge T^i) \quad (2)$$

Note that if $N = 0$ is set in Eq. 1, it simplifies to a time-frame shifted version of Eq. 2 (with the addition of the initial state constraints S^u) because it eliminates all additional circuitry relating to error models.

By combining Eq. 1 and 2, one can overcome the issue of debugging a time-window that does not contain the observed failure. This formulation is shown with respect to a set of time-windows from p to q with a width of w:

$$TimeDebug^q_p(N) = Debug^{(p+1)\cdot w-1}_{p\cdot w}(N) \wedge \bigwedge_{i=p+1}^{q} W_i \quad (3)$$

This formulation permits the effects of the error models in time-window under analysis (*e.g., p*) to propagate and affect the observed failure in a subsequent suffix time-window (*e.g., q*).

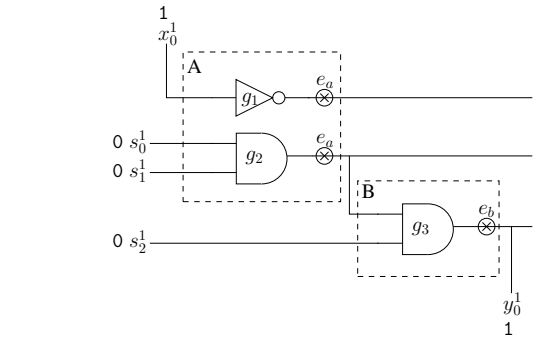
Example 1 *Figure 1(a) visualizes Eq. 3 when $p = q = 1$, $w = 1$ and $N = 1$. The circuit under debug has three internal gates (g_1, g_2, g_3), one input (x_0), one output (y_0), and three state-variables (s_0, s_1, s_2). In this circuit, g_1 and g_2 form module A and g_3 forms module B. The error models are denoted by $\otimes$. There are two suspect variables, e_a, e_b corresponding to module A and B respectively. The suspect variable e_a, corresponds to the two outputs of g_1 and g_2 from module A. Similar is the situation for suspect variable e_b and gate g_3. Error trace values for the initial state, input and expected outputs are shown in that the figure.*

When the instance in Figure 1(a) is passed to a SAT-solver, one solution is returned, namely $e_b = 1$, that corresponds to module B being a suspect. When $\overline{e_b}$ is added as a unit clause to block it from appearing again, the instance is `UNSAT` *indicating that no more solutions are found.*

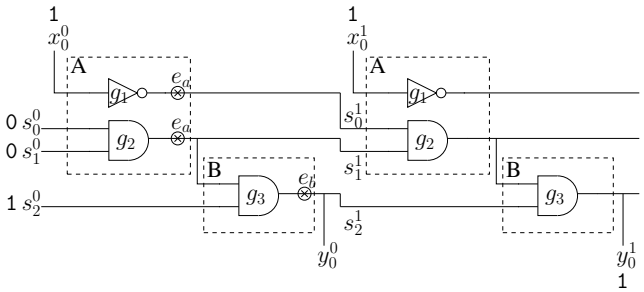
Figure 1(b) shows a visualization of Eq. 3 with the next time-window where $p = 0, q = 1, w = 1$ and $N = 1$. In this instance, the suspect variables now only correspond to modules in the earliest time-frame. When sent to the SAT-solver, one solution is returned where $e_a = 1$. After blocking it as a solution, the instance is `UNSAT` *indicating that all solutions have been found.*

Time-windowing techniques [18] use a similar formulation to Equation 3. For a given time-window width, they divide the error trace into multiple non-overlapping time-windows. Each window is iteratively analyzed starting from the last time-window in the error trace that contains the observed failure (*i.e., $p = q$* for Eq. 3). This "sliding window" of time-frames moves towards earlier time-windows ($p = q, p = q-1, \ldots, p = 0$) iteratively analyzing the error trace. To ensure that the current time-window under analysis (*e.g.,* window p) can propagate the effect of its error models to the observed failure (*e.g.,* window q), various methods are used to model the suffix time-windows.

In the most basic formulation, suffix time-windows in the trace are explicitly modeled as in Eq. 3. This results in iteratively solving instances of Eq. 3, where each iteration results in a larger problem size. The proposed strategy leads to complete results but suffers from performance and memory issues in later iterations. Subsequent formulations forego the need to explicitly model suffix time-frames and instead rely on approximations to model later time-windows (W_i). This ensures that the effect of the error models in the window under analysis can propagate and affect the observed failure. The



(a) $TimeDebug_1^1(N=1)$



(b) $TimeDebug_0^1(N=1)$

Fig. 1. Time Diagnosis Example

approximations effectively reduce the problem size but at the cost of poor quality of results since they provide no means of refinement.

D. Conflict Graphs, UNSAT Cores, and Counter-Example Guided Abstraction and Refinement

Modern SAT-solvers [27] use a conflict-driven clause learning process whereby new clauses are learnt during backtracking in the search process. In the case of an UNSAT instance, a proof of unsatisfiability can be extracted that shows how original clauses can be resolved together to produce the empty clause. This requires keeping track of all resolution steps performed to generate the learnt clauses. However, if this information is not retained, a partial proof can be obtained that shows how a combination of original and learnt clauses can be resolved to generate the empty clause. We denote this proof graph as the *conflict graph*. This partial proof can be extracted using the final implication graph of a SAT solver that led to the UNSAT result.

Alternatively, given an unsatisfiable (UNSAT) Boolean formula ϕ in conjunctive normal form (CNF), an *UNSAT core* is a subset of clauses that are unsatisfiable. With respect to debugging, an UNSAT core intuitively represents paths in which a counter-example can excite an error, traverse its effects through the design components and cause a failure at the observation points (*e.g.,* primary outputs).

Conflict graphs and UNSAT cores are related in that they both represent how an instance is UNSAT. They also both represent an *over-approximation* of the original SAT instance due to the fact that the clauses are either a subset of the original instance (UNSAT cores), or implied from the original instance (conflict graphs). However due to the restriction that UNSAT cores must contain all original clauses, they incur additional overhead during the solving process to maintain the necessary bookkeeping information for generating original clauses. In this work, we use conflict graphs instead of UNSAT cores to efficiently generate an approximation of an UNSAT instance.

SAT-based Counter-Example Guided Abstraction and Refinement (CEGAR) [28] is an iterative model checking algorithm composed of three general steps: abstraction, model checking and refinement. The initial abstraction begins with an over-approximation of the concrete problem. During the model checking phase, if no counter-example is found, then the target property is proved due to the over-approximate abstraction. Otherwise it produces a counter-example, which may be spurious and must be verified. This is done by applying the counter-example to an unrolled concrete transition relationship. If this instance is SAT then the counter-example is valid. In the UNSAT case, the counter-example is spurious and refinement begins which much strengthen the abstraction to ensure that the spurious counter-example is not found. After refinement, this process repeats until either a valid counter-example is found, or the property is determined to be valid.

III. Path Directed Abstraction and Refinement

This section presents a design debugging abstraction and refinement algorithm built upon both a CEGAR [28] and time-windowing framework with the debugging model from Eq. 3.

This key idea is shown in Figure 2 with an error trace containing $k+1$ time-frames. In this example, the trace is divided into $\frac{k+1}{w}$ non-overlapping time-windows with $w=2$. As with other time-windowing techniques, the algorithm iteratively analyzes non-overlapping time-windows starting from the latest window. The abstract time diagnosis instance for time-window $\frac{i}{w}$ is shown in greater detail. This instance is constructed by concretely modeling the current window (*e.g.,* cycles i and $i+1$) and abstracting all later time-windows in the suffix (*e.g.,* cycles $i+2$ to k). This abstraction is iteratively refined in a CEGAR-like abstraction and refinement loop with three overall steps:

1) **Abstraction:** Generate an abstract time diagnosis instance by concretely modeling the current time-window combined with an over-approximate abstraction for the later suffix time-windows. This abstraction is conceptually based upon structural circuit paths that directly led to the observed failure in the error trace.
2) **Debugging:** Find all solutions to the abstract time diagnosis debugging instance. Since this instance over-approximates Eq. 3, it generates a superset of the solutions compared to explicit modeling. To ensure only valid solutions are returned, each solution must then be verified in the refinement phase.
3) **Refinement:** For each solution, propagate its associated satisfying assignment forward to concretely modeled time-windows in separate SAT instances. If each instance is SAT, then the solution is valid and can be blocked from appearing in future iterations. Otherwise, a conflict is generated indicating that the abstraction is not sufficient to prove the current solution. In this case, refinement occurs where additional paths (in the form of clauses

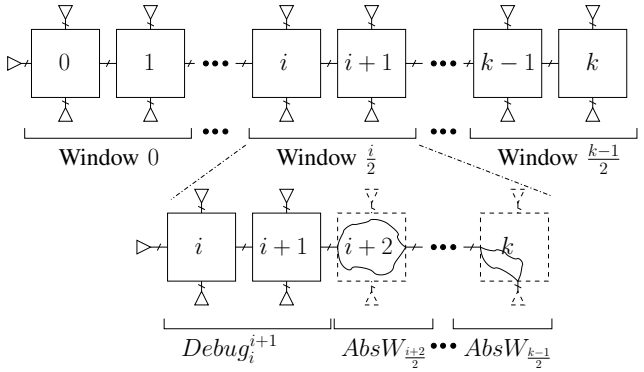


Fig. 2. Path Directed Abstraction and Refinement

involved in the conflict graph) are added back to the abstraction.

4) Repeat steps (2)-(3) until no more new valid solutions are found in step (2).

After all solutions have been found for the current time-window, the algorithm moves on to an earlier time-window re-using the refined abstraction generated for suffix time-windows from the current iteration.

The following subsections describe the formulation, theoretical results and pseudo-code for the path-directed abstraction and refinement algorithm in greater detail.

A. Path-based Abstraction

The path-based abstraction approximates Eq. 3 by replacing consecutive windows of time-frames (W_i) with an abstract version, denoted by $AbsW_i$. This is initially formulated by adding paths in the time-frame expanded circuit that are directly involved in the observed failure. Practically, this is a set of clauses extracted from the SAT-solver conflict graph generated while modeling the observed failure. The rest of this subsection describes this process in greater detail.

Consider the first abstract time-window occurring at the end of the trace ($AbsW_q$) for cycles $q \cdot w$ to $(q+1) \cdot w - 1$. This abstract window is first needed for use in the next iteration once we have finished analyzing window $p = q$ using Eq. 3. Notice that in this case by setting $N = 0$ in Eq. 3, it simplifies to W_q with the addition of the initial state constraints. This precisely models the circuit behavior for cycles $q \cdot w$ to $(q+1) \cdot w - 1$ that led to the observed failure. The resulting conflict graph from the SAT-solver will contain clauses that are directly involved in the observed failure at the primary outputs. These clauses, excluding the initial state constraints, form the initial abstraction $AbsW_q$.

In general, the abstraction is formed by maintaining a set of clauses for each time-window that has been previously analyzed. When a time-window has completed analysis, we set $N = 0$ for that instance and clauses within that window are extracted. This is shown in the following equation:

$$AbsW_i = Conflict_i(AbsDebug_i^q(N = 0)) \tag{4}$$

Here, $Conflict_i$ denotes all clauses with variables within time-frames i to $i + w - 1$ involved in the conflict of the input formula, excluding the initial state constraints. This can be extracted by analyzing the conflict graph of the SAT-solver when it returns UNSAT, extracting only clauses that correspond to the relevant time-frames. $AbsDebug_i^q(N)$ denotes the abstract debugging instance for the previously analyzed time-window. In the base case for $i = q$, this simplifies to Eq. 1.

Using Eq. 4, the initial abstract debugging problem can be created, shown in the following formula:

$$AbsDebug_p^q(N) = Debug_{p \cdot w}^{(p+1) \cdot w - 1}(N) \wedge \bigwedge_{i=p+1}^{q} AbsW_i \tag{5}$$

This formula mirrors Eq. 3 except it uses the abstract windows for later time-frames instead of explicitly modeling them. It results in a greatly reduced memory footprint because the abstract windows typically are much smaller than the explicit model. In addition, since each abstract window ($AbsW_i$) contains either clauses that were in the explicit model of that window (W_i) or implied by it, Eq. 5 is in fact an over-approximation of Eq. 3 as stated in the next lemma.

Lemma 1 *Let $\mathcal{E}$ be a set of N active suspect variables found in a satisfying assignment to $TimeDebug_p^q(N)$, then there is a satisfying assignment to $AbsDebug_p^q(N)$ that will also contain the active suspect variables from $\mathcal{E}$.*

Proof: Let $\mathcal{A}$ be the satisfying assignment to $TimeDebug_p^q(N)$. We will show that each of the sub-formulas of $AbsDebug_p^q(N)$ is satisfiable under $\mathcal{A}$. $Debug_{p \cdot w}^{(p+1) \cdot w - 1}(N)$ is SAT under $\mathcal{A}$ because it is a subset of the clauses in $TimeDebug_p^q(N)$. Each $AbsW_i$ is derived from extracting clauses from a SAT-solver conflict graph that are within the same time-frames modeled by W_i. This means that these clauses are either a subset of W_i or implied by the overall problem. Therefore, if all W_i are SAT under $\mathcal{A}$, so are all $AbsW_i$. Since each component of is SAT under $\mathcal{A}$ so is $AbsDebug_p^q(N)$ with active suspect variables $\mathcal{E}$. ■

Lemma 1 implies that the initial abstract instance will return a superset of debugging solutions when compared to explicit modeling. This abstraction resembles the method for approximating time-frames in [18] because they both over-approximate a suffix of time-frames and are derived from an UNSAT problem. However, they differ in the method used to computer the approximation. While [18] calculated an interpolant directly from the proof of unsatisfiability, our method uses the conflict graph. This method is more efficient because it removes the need to record and traverse the UNSAT proof to calculate the interpolant, while still retaining equivalent information. Additionally, the abstraction allows an efficient means to incrementally refine the problem efficiently, which will be described in later sections.

Due to the abstraction providing a superset of the debugging solutions, each solution must be verified to ensure it is not spurious. To accomplish this, the satisfying assignment of each solution is propagated forward to each subsequent concretely modeled time-window. If a solution is indeed spurious, the resulting conflict will act as a refinement step, a process discussed in the following subsection.

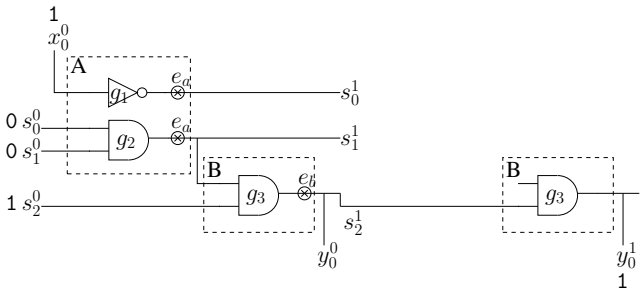


Fig. 3. Abstraction Example

Example 2 *Consider the debugging instance from Figure 1(a) in Example 1. Once this instance returns* UNSAT*, the algorithm sets $N = 0$, it runs a SAT-solver and it extracts clauses involved in the resulting conflict graph to generate $AbsW_1$.*

In the next iteration of the main loop, it generates an abstract debugging problem using Eq. 5 with $p = 0, q = 1, w = 1$ and $N = 1$. A visualization of this instance is found in Figure 3. Here, most of circuitry from time-frame 1 has been abstracted leaving only the path directly involved in the conflict from the SAT-solver. Note this path is not unique and another conflict graph has the potential to generate a different path.

B. Path Directed Refinement

Due to the use of abstract windows in Eq. 5, a satisfying assignment, denoted by $\mathcal{A}$, may not correspond to one in the concrete instance and therefore may be spurious. A straightforward method to verify $\mathcal{A}$ would be to apply it to the entire concrete instance in Eq. 3. However, this would negate the benefit of a reduced memory footprint since the instance would involve explicit modeling of all time-frames. Instead, by utilizing the previous abstract windows, it is possible to create several smaller SAT instances that can equivalently verify $\mathcal{A}$.

In order to propagate $\mathcal{A}$ forward, only a subset of assignments are actually needed. Notice that in Eq. 5, time-frames from $p \cdot w$ to $(p+1) \cdot w - 1$ are modeled explicitly, while the remaining are not. This means that an assignment for the first $p \cdot w$ to $(p+1) \cdot w - 1$ time-frames on the abstract model should also work for the respective frames on the concrete model in Eq. 3. However, the only way for these concrete time-frames to affect forward time-frames are through the state variables at time-frame $(p+1) \cdot w$. If these state assignments are used to constrain subsequent concretely modeled time-frames, then we can iteratively propagate the effect of the original assignment forward to each concretely modeled time-window. We denote this subset of assignments of $\mathcal{A}$ on state variables for time-frame i by $cube^i$.

To accomplish this propagation, we create multiple instances each of which models precisely w concrete time-frames and uses the abstract windows for the others. This construction is represented by the following equation:

$$Prop_r^q = cube^{r \cdot w} \wedge W_r \wedge \bigwedge_{i=r+1}^{q} AbsW_i \tag{6}$$

The state assignment $cube^{(p+1) \cdot w}$, extracted from $\mathcal{A}$, is propagated using the above equation to generate $Prop_{p+1}^q$. If this results in SAT, another cube, $cube^{(p+2) \cdot w}$, will be extracted and propagated to the next instance, $Prop_{p+2}^q$, and so on, until all time-frames have been verified. If all subsequent instances result in SAT, then the original abstract satisfying assignment can be extended to one in the the concrete model. This is stated more precisely in the following lemma.

Lemma 2 *Let $\mathcal{E}$ be a set of N active suspect variables found in a satisfying assignment of $AbsDebug_p^q(N)$. If $AbsDebug_p^q(N)$, $Prop_{p+1}^q$, ..., $Prop_q^q$ are* SAT*, then there is a satisfying assignment to $TimeDebug_p^q(N)$ that will also contain the active suspect variables from $\mathcal{E}$.*

Proof: Let $\mathcal{A}_p, \mathcal{A}_{p+1}, \ldots, \mathcal{A}_q$ be the satisfying assignments to $AbsDebug_p^q(N)$, $Prop_{p+1}^q$, ..., $Prop_q^q$, respectively. We show how to construct an assignment $\mathcal{A}$ to $TimeDebug_p^q(N)$ from $\mathcal{A}_p, \mathcal{A}_{p+1}, \ldots, \mathcal{A}_q$.

From $\mathcal{A}_p$, we add all assignments to variables involved in the subformula $Debug_{p \cdot w}^{(p+1) \cdot w - 1}(N)$ to $\mathcal{A}$. From each of the subsequent $\mathcal{A}_r$, we add assignments to all variables of W_r from the respective instance $Prop_r^q$. Notice the overlapping variables that were added to $\mathcal{A}$ are precisely the state variable from the cubes $cube^{p \cdot w}$, $cube^{(p+1) \cdot w}$, ..., $cube^{q \cdot w}$. But from the formulation of $Prop_{r-1}^q$, each cube $cube^{r \cdot w}$ is generated from an instance that is constrained by the previous cube, $cube^{(r-1) \cdot w}$, except for the first one derived from the assignment $\mathcal{A}_p$. This means that there is no conflict between these overlapping state variables because each one is implied by the previous one in the sequence which originates from $\mathcal{A}_p$.

The final constructed assignment $\mathcal{A}$ composes a full assignment to $TimeDebug_p^q(N)$ since it contains assignments to all variables in each component of the formula. Moreover, $\mathcal{A}$ is a valid satisfying assignment to each of the components of $TimeDebug_p^q(N)$. We can see this because we constructed $\mathcal{A}$ by extracting exactly the assignments involved in the concrete parts of the abstract formulas which are identical to the concrete model. These concrete parts compose exactly the clauses of $TimeDebug_p^q(N)$, so it too is SAT under $\mathcal{A}$ with active suspect variables $\mathcal{E}$. ■

After a set of active suspect variables is found, they are blocked so that the next solution can be found for $AbsDebug_p^q$. Notice that the propagation generates a separate instance for every w time-frames resulting in exactly w concretely modeled time-frames for any given instance. This key point ensures that the memory footprint of the entire process is kept to a minimum.

On the other hand, if $Prop_r^q$ is UNSAT, this means that the original assignment $\mathcal{A}$ cannot be extended to the concrete model. In this case, the abstract window was not sufficiently refined and additional clauses must be added. Similar to the initial abstraction, we extract clauses from the SAT-solver conflict graph and add them to the abstract window. Intuitively, this indicates that additional paths are required. This is shown in the following equation:

$$\hat{AbsW}_r = AbsW_r \cup Conflict_r(Prop_r^q) \tag{7}$$

Notice that the refined abstract window, $\hat{AbsW}_r$, still is an over-approximation of the concrete window because it only contains clauses that are either a subset, or implied by, the concrete window it models.

With the refined abstract window $\hat{AbsW}_r$, we can similarly refine all previous abstract windows. This can be accomplished by re-creating Eq. 6 with $j < r$ and the current refined abstract window ($\hat{AbsW}_r$). Since $\mathcal{A}$ is known to be invalid, each one of these instances will be UNSAT because they are just an extension of the assignment $\mathcal{A}$. After the abstract windows have been updated, the refined formula defined by updating Eq. 5 can be solved for all solutions again. These solutions similarly can be either be confirmed or used to refine the abstract time-windows until the refined instance, $\hat{AbsDebug}_p^q$, is unsatisfiable.

Once all solutions are found, $\hat{AbsDebug}_p^q$ is UNSAT indicating that debugging has been completed on this time-window. The next theorem confirms the completeness of our approach, that is, the set of solutions found in the refined abstract model equals to the one found for the concrete model.

Theorem 1 *Let $sols_{abs}$ be the set of confirmed debugging solutions returned by iteratively debugging and refining $\hat{AbsDebug}_p^q$ and let $sols_{time}$ be the set of debugging solutions returned by $TimeDebug_p^q$. If the final refined abstract instance $\hat{AbsDebug}_p^q$ is UNSAT by blocking all solutions in $sols_{abs}$, then $sols_{abs} = sols_{time}$.*

Proof: From Lemma 1, we know that $sols_{abs} \supseteq sols_{time}$. From Lemma 2, we also know that any set of debugging solutions $\mathcal{E}$ found and verified by $\hat{AbsDebug}_p^q$ is also valid for $TimeDebug_p^q$ *i.e.,* $sols_{abs} \subseteq sols_{time}$. Therefore, $sols_{abs}$ is both a superset and subset of $sols_{time}$, so $sols_{abs} = sols_{time}$. ■

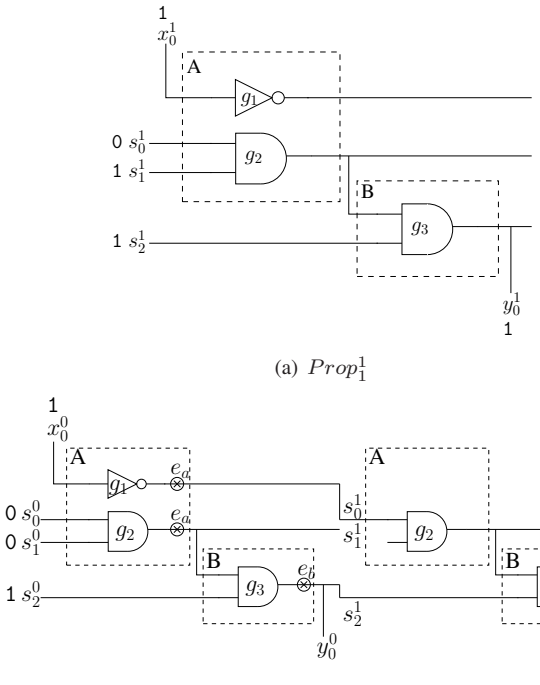
Example 3 *Building upon the abstract problem from Example 2, when the instance in Figure 3 is given to the SAT-solver, this instance returns one solution where $e_a = 1$ since e_b was already blocked from the previous time-window. The associated cube with this satisfying assignment is $cube^1 = \overline{s_0^1} \wedge s_1^1 \wedge s_2^1$. This cube is verified by generating a new instance using Eq. 6 with $r = q = 1$, it is shown in Figure 4(a) and one can show that it is* UNSAT. *The clauses from the conflict graph are extracted and the abstract window $AbsW_1$ is refined.*

Figure 4(b) visualizes the refined abstract debugging problem $\hat{AbsDebug}_0^1(N = 1)$. This instance again finds a solution with $e_a = 1$ and a new cube, $\hat{cube}^1 = s_0^1 \wedge s_1^1 \wedge s_2^1$. This cube is verified using Eq. 6 which returns SAT. *Thus $e_a = 1$ is confirmed as a solution matching the results from Section 1. Once this solution is blocked, the refined instance returns* UNSAT *indicating that this time-window has completed analysis.*

C. Overall Algorithm

Algorithm 1 presents pseudo-code for the path directed abstraction and refinement algorithm for time-windows from p to q. The main loop from lines 3-13 iterates through each time-window analyzing them separately. Line 4 constructs the initial abstract instance using Eq. 5 with any previously generated abstract windows ($AbsW_i$). In the first iteration, this equation simplifies to Eq. 1 where the very last time-window is analyzed. The inner WHILE loop (lines 5-11) finds all satisfying assignments and confirms each one by passing the result to the VERIFY procedure (Algorithm 2). If the assignment is not confirmed, then the procedure refines the relevant abstract windows. Once all the current assignments have been verified, the refined abstract problem is reconstructed (line 10), blocking any solutions that were confirmed from being found again. When the algorithm has finished analyzing the current time-window, it exits the WHILE loop and generates the initial abstraction for the current window on line 12 for use in the next time-window.

The pseudo-code for the VERIFY procedure is shown in Algorithm 2. The procedure begins by extracting the state cube for window $p + 1$ from the assignment $\mathcal{A}$ on line 2. The outer FOR loop (lines 3-13) propagates the cube forward to subsequent windows using Eq. 6 (line 4). If any of the subsequent time-windows are UNSAT (line 5), then the abstract windows are refined by iterating backwards through the windows (lines 6-9). Each backward iteration reconstructs Eq. 6 with the refined abstract windows on line 7. The refinement step (line 8) extracts clauses from each of these UNSAT instances to update the abstract window. The procedure either returns the confirmed solution (line 14) or will return an empty solution otherwise (line 10).



(a) $Prop_1^1$

(b) $\hat{AbsDebug}_0^1(N = 1)$

Fig. 4. Refinement Example

Algorithm 1 Path Directed Abstraction and Refinement

```
1: procedure PATHDEBUG
2:     sols ← ∅
3:     for p ∈ q, q − 1, . . . , 1, 0 do
4:         inst ← AbsDebug_p^q ∧ BLOCK(sols)
5:         while inst is SAT do
6:             Assignments ← SOLVEALL(inst)
7:             for A ∈ Assignments do
8:                 sols ← sols ∪ VERIFY(A, p, q)
9:             end for
10:            inst ← AbsDebug_p^q ∧ BLOCK(sols)
11:        end while
12:        AbsW_p ← Conflict_p(inst)
13:    end for
14:    return sols
15: end procedure
```

Algorithm 2 Solution Verification

```
1: procedure VERIFY(A, p, q)
2:     cube^{(p+1)·w} ← EXTRACTCUBE(A, p + 1)
3:     for i ∈ p + 1, p + 2, . . . , q − 1, q do
4:         inst ← Prop_i^q
5:         if inst is UNSAT then
6:             for j ∈ i, i − 1, . . . , p + 2, p + 1 do
7:                 inst ← Prop_j^q
8:                 AbsW_j ← AbsW_j ∪ Conflict_j(inst)
9:             end for
10:            return ∅
11:        end if
12:        cube^{(i+1)·w} ← EXTRACTCUBE(inst, i + 1)
13:    end for
14:    return EXTRACTSUSPECT(A)
15: end procedure
```

IV. Improved Refinement

A key step in Algorithm 1 is the VERIFY procedure which propagates state cube assignments forward to later time-windows. If any of the later windows are UNSAT, then the conflict graph can be used to extract clauses to refine the abstract windows. This section presents several methods to improve the basic refinement strategy to either reduce the overall number of iterations in Algorithm 1, or improve its performance.

A. Finding Additional Conflicts using Necessary Assignments

As described previously, given an UNSAT instance of Eq. 6, one can extract conflicts to refine the current abstract window. However, since only one conflict is extracted per instance of Eq. 6, this may require many calls to the VERIFY procedure, decreasing the overall performance of the algorithm. To mitigate this problem, one can extract multiple conflicts out of an UNSAT instance of Eq. 6.

One method to accomplish this task is to remove literals in the propagated assignment ($cube^i$ for $i = r \cdot w$ in Eq. 6) that were involved in the conflict and send the instance to the SAT solver again. If this less constrained problem is still UNSAT, then an additional conflict can be extracted to strengthen the refined abstraction. However, one drawback of this method is that the two conflicts will not have any overlapping literals in $cube^i$ *i.e.*, the cube literals involved in the two conflicts are disjoint. This restricts the number of conflicts (and thus paths) that can be found because in many cases additional conflicts only occur when the literals overlap.

Alternatively, a more costly method would be to try all possible subsets of $cube^i$ and see if any one of these results in an UNSAT instance. However, this method can be inefficient especially when the number of literals involved in the original conflict grows. A more efficient method for finding these types of conflicts is possible by determining additional information with regards to the literals of $cube^i$.

Consider the EXTRACTCUBE procedure on line 2 of Algorithm 2. For each assignment $\mathcal{A}$, we wish to store additional information about the extracted cube to distinguish which literals are due to the effect of the active error model (e_i) from the assignment, and which are necessary regardless of the assignment to the error model's free variables. When finding additional conflicts, one can simply remove only those cube literals that are not within this necessary assignment. This allows for additional overlapping conflicts from $cube^i$ to be efficiently generated without the need to try all possible subsets of the cube literals.

Although it is possible to compute these necessary assignments to $cube^i$ precisely using multiple SAT calls [29], a more efficient approximate method is used. Consider the following equation of the abstract problem where the active suspect variable e_i are constrained for a given assignment:

$$AbsDebug_p^q \wedge e_i \tag{8}$$

This instance has multiple SAT assignments due to the free variables from the error models disconnecting the active suspect's fan-in from its fan-out. One important observation is that these free variables do not necessarily affect every state variable in the extracted cube ($cube^i$). Thus, an efficient method to determine which assignments are necessary is to run the solver's Boolean constraint propagation (BCP) engine on the above instance. This will determine a large set of directly implied assignments to the extracted cube state variables ($cube^i$), thus producing a subset of the state cube that are necessary for every assignment involving e_i.

This same process can be repeated for Eq. 6 as well by using only the necessary literals of the initial state cube:

$$Prop_r^q = Fixed(cube^{r \cdot w}) \wedge W_r \wedge \bigwedge_{i=r+1}^{q} AbsW_i \tag{9}$$

Here $Fixed$ extracts only the literals that were marked as necessary. Similarly, the solver's BCP engine can be run on this instance to determine which literals in the resulting SAT assignment are necessary, and which ones are due to the effect of e_i. This will allow one to calculate the necessary variables for the next state cube $cube^{(r+1)\cdot w}$.

Using this information for each UNSAT instance of Eq. 6, additional overlapping conflicts can be extracted by removing

non-necessary literals in $cube^i$ that were involved in the conflict, and re-running the SAT engine. This process can be repeated until either all non-necessary literals have been removed, or the instance becomes SAT.

B. Finding Additional Conflicts using Multiple State Cubes

As mentioned in the previous sub-section, each state cube ($cube^i$) consists of both literals that are affected by the error models, and literals that are not affected (*i.e.*, necessary literals). Since the error model allows its corresponding circuit lines to be free, multiple assignments are possible to these subset of cube literals that are affected by the error models. In fact given a $cube^i$, the corresponding suspect variable may in fact be a solution but due to the approximate nature of the abstraction, a non-valid assignment to the free variables (with respect to the concrete problem) was chosen. This results in the subsequent propagation instances of $Prop_j^q$ (*i.e.*, Eq. 6), to be UNSAT allowing refinement to take place. Eventually, the abstraction will be refined enough to exclude these spurious non-valid assignments, however, this may require many iterations. To overcome this issue, we can speculatively generate multiple additional state cubes from an original cube ($cube^i$) in an effort to find additional conflicts.

For a given $cube^i$, an additional cube can be speculatively generated by inverting a non-necessary literal of the cube. The intuition here is that a single bit flip from a non-necessary literal has a high likelihood of being a valid solution to the abstract problem $AbsDebug_p^q$ (and thus a valid cube). However, it is possible that this assignment is not a solution to the abstract problem. In this case, it is not relevant so long as the speculatively generated cube results in a conflict to allow additional paths to be refined. These additional paths may not necessarily be used directly for the current suspect variable e_i, but may be useful in later time-windows or for other suspect variables. Since the conflicts are still with respect to Eq. 6, any refinement still maintains the over-approximate nature of the abstraction. This results in a net benefit because these additional cubes are efficient to calculate and accelerate refinement.

The additional cubes can be applied to Eq. 6 on line 7 of Algorithm 2. If the resulting instances are UNSAT, additional clauses can be extracted from the conflict-graph accelerating refinement in an efficient manner.

C. Improving the Initial Abstraction

The initial abstraction for each time-window (as calculated on line 12 of Algorithm 1) is derived from a single conflict graph. This typically is a very weak abstraction because it contains only a small subset of paths involved in the original failure. An efficient means to generate a stronger initial abstraction would greatly increase performance by reducing the number of refinement iterations needed to analyze later time-windows.

Observe that each abstract time-window has the same unrolled circuit, differing only in the error trace values that are applied. This leads to the idea that many of the paths that are needed for an error to propagate are similar between different time-windows. More precisely, we can use the previously refined abstract time-windows ($\hat{AbsW}_{p+1}$) to strengthen the initial abstraction for the current time-window ($AbsW_p$).

For $AbsW_p$, many of the clauses from the previous refined window, $\hat{AbsW}_{p+1}$, will be valid. We can use these clauses to strengthen $AbsW_p$ by shifting the literals in the respective clauses to the corresponding circuit lines in the current window. However, not all of the clauses will be valid due to differences in the error trace values. Therefore, each clause from $\hat{AbsW}_{p+1}$ must be checked in order to ensure its validity. This can be accomplished efficiently in the following way.

For each clause in $\hat{AbsW}_{p+1}$, remap its variables so they correspond to the same circuit lines in the current window ($AbsW_p$). Next, negate the clause and add the corresponding literals to the abstract debug problem as shown in the next equation:

$$AbsDebug_p^q \wedge \overline{clause_i} \tag{10}$$

If this instance is UNSAT, then the clause is implied by the abstract problem and can be added to the current abstract window ($AbsW_p$). Since this needs to be performed for many clauses, an efficient approximate method to accomplish this is to use the SAT solver's BCP engine to propagate the literals in $\overline{clause_i}$. If the result is UNSAT, then we can add the clause to the current abstract window. Otherwise, do not add it. This will not necessarily determine if the clause is implied by the abstract problem in all cases, but rather provides an efficient means to gather a large subset of them compared to explicit calls to the SAT-solver.

D. Leveraging the SAT-solver for Efficient Implementation

Beyond improvements to refinement, an essential aspect in realizing the proposed algorithm is making extensive use of the SAT-solver. Two key implementation notes are discussed in this sub-section. The first improvement involves extensive use of unit assumptions and incremental SAT capabilities of modern solvers [27], [30]. This capability of modern solvers is used wherever possible to reduce the number of newly created instances, greatly reducing the computational friction between different SAT instances. One example of where this provided significant benefit is a modification of Algorithm 1. During the VERIFY procedure instead of verifying one assignment at a time, multiple assignments can be verified using one instance of $Prop_i^q$. This is accomplished by setting each cube, $cube^{i \cdot w}$, as unit assumptions and re-solving the instance using incremental SAT. This greatly reduces the overhead of re-solving the same instance (with the exception of the state cube) multiple times.

The second improvement involves clause subsumption [27]. During refinement, many new clauses are added to the abstraction. Some are redundant due to overlapping paths. To ensure that the abstraction does not contain this unnecessary bloat, subsumption is used to reduce duplicate information being stored. This step can be performed each time an abstract window is refined, maintaining an efficient abstraction.

V. Flexible Path Directed Debug

Algorithm 1 provides a complete method to find the exact set of suspects as the explicit formulation of Eq. 3 while dramatically reducing the problem size. However in certain worse-case scenarios, the iterative nature of calling the VERIFY procedure will result in long run-times. In this section, we introduce a novel flexible algorithm that allows a trade-off between performance and quality of results to mitigate these problematic cases.

The worse-case scenario in Algorithm 1 occurs when an excessive number of iterations of the WHILE loop on line 5 occurs. This can happen when a suspect requires an excessive number of paths to prove that it is valid. The algorithm will continually attempt to refine the abstract problem and will eventually converge to prove or disprove the solution's validity but may require a long run-time.

To overcome this issue, consider the case where a suspect is verified on an abstract model. The fact that it is verified on the abstract model does give some confidence that it is valid, especially if it is on a partially refined model. This translates to the suspect being valid among a subset of paths that were present in the abstraction, but not necessarily among all paths in the concrete model.

Using this idea, if a suspect has been propagated forward multiple times without ruling it out, then it can be recorded as partially verified, and skipped for future iterations of the algorithm. In this manner, we remove the worse-case scenario by limiting how many times a given assignment for a suspect can be propagated forward. Practically, this results in a small number of suspects being skipped while providing a large benefits in run-time.

Algorithm 3 shows pseudo-code for the modified algorithm named FLEXIBLE PATHDEBUG. This algorithm defines two new variables, the $skip\_limit$ parameter and $skips$ set. The $skip\_limit$ parameter defines the trade-off between quality of results and performance. In the case of $skip\_limit = \infty$, the behavior mimics Algorithm 1. In the case of $skip\_limit = 0$, no verification is done on the solutions resulting in a similar algorithm to [18] where no refinement is present. For values in between these two limits, the user can define an refinement strategy that trade-offs between complete results and a loose abstraction. The $skip$ set contains all suspects that were skipped due to the $skip\_limit$. This distinguishes them from the suspects that have been fully verified giving additional information to the user.

In Algorithm 3, the main loop along with the WHILE loop retain the same functionality of analyzing time-windows and verifying satisfying assignments. However, the algorithm differs on line 8 where it checks if the assignment's suspect has reached the $skip\_limit$. If it is within the limit, the algorithm runs the VERIFY procedure as before (line 9). Otherwise, the suspect associated with this solution is added to the $skips$ set (line 11). The $skips$ set is then blocked on line 14 so that it does not show up again as an assignment in the WHILE loop. Finally, the algorithm will return both the $sols$ and $skips$ set to the user in order to distinguish between solutions that have been verified and ones that have not.

Algorithm 3 Flexible Path Directed Abstraction and Refinement

```
1:  procedure FLEXPATHDEBUG
2:      sols ← ∅, skips ← ∅
3:      for p ∈ q, q − 1, ..., 1, 0 do
4:          inst ← AbsDebug_p^q ∧ BLOCK(sols)
5:          while inst is SAT do
6:              Assignments ← SOLVEALL(inst)
7:              for A ∈ Assignments do
8:                  if count_sol(A)++ < skip_limit then
9:                      sols ← sols ∪ VERIFY(A, p, q)
10:                 else
11:                     skips ← skips ∪ SOL(A)
12:                 end if
13:             end for
14:             inst ← AbsDebug_p^q ∧ BLOCK(sols ∪ skips)
15:         end while
16:         AbsW_p ← Conflict_p(inst)
17:     end for
18:     return sols,skips
19: end procedure
```

VI. Experiments

This section presents the experimental results for the proposed algorithms. All experiments are run on a single core of an Intel Core i5 3.1 Ghz quad-core workstation with 8 GB of RAM and a timeout of 7200 seconds. All experiments use MINISAT [30] as the SAT solver with a Verilog frontend to allow for diagnosis of RTL designs.

The effectiveness of the techniques are shown on industrial Verilog RTL designs from OpenCores [21] as well as two commercial designs (`fxu`, `comm`) provided by our industrial partners. Table I presents relevant design statistics for each of the RTL designs. The three columns show the design name, number of gates, and the total number of potential error locations where error models are applied.

Each debugging instance is created by randomly selecting a line in the RTL and inserting a typical industrial RTL error (wrong state transition, incorrect operator, erroneous instantiation etc.). It is important to emphasize that each RTL error typically corresponds to multiple gate-level errors locations, effectively allowing for diagnosis of multiple errors. Next, the buggy RTL is simulated with its accompanying testbench to observe the failure and record its error trace. Finally, each resulting instance is run through a cone of influence reduction [31] to remove logic that is not involved in the observed failure. This may result in different effective problem sizes for different instances of the same design. Finally, each instance is run through the respective debugging algorithm with $N = 1$ to compare the performance and quality of results.

A. Refinement Improvements

The first set of experiments shows the effect of the refinement improvements from Section IV. This results are shown in Table II. The basic algorithm from Section III is compared to

TABLE I
DESIGN CHARACTERISTICS

design	# gates (k)	# susps
ac97	22.9	1380
comm	164.4	20155
div64	59.7	197
fdct	239.5	4662
fpu	72.5	1982
fxu	447.3	33087
mips_sys	50.3	2636
rsdecoder	14.5	2040
usbfunct	35.8	4061
vga	48.8	1731

the same algorithm but with the refinement improvements from Section IV-A to IV-C. To ensure comparable results, instances are selected such that they are able to run to completion without causing a time-out or mem-out using 10 time-windows of size $w = 10$ (*i.e.,* an error trace of 100 time-frames).

The first column in Table II shows the instance name. The next six columns show the run-time, peak memory usage, and number of calls to VERIFY procedure in Algorithm 2 for both the basic algorithm and the one with the refinement improvements.

The benefits of the refinement improvements manifest themselves in both the run-time and number of calls to VERIFY with an average decrease of 33.4% and 66.0% respectively. The average peak memory increased slightly by 1.5%, which is an acceptable trade-off for the increased performance. The reduction in run-time is primarily caused by the reduced number of calls to VERIFY showing the efficacy of the improvements to quickly refine the abstraction.

From Table 2, the refinement improvements do incur some overhead as can be seen in `ac97_1`, where the number of calls to VERIFY are identical, but the improvements take longer to run. This can be explained by the fact that `ac97_1` case does not require much refinement, and the improvements have little effect in these cases. However for most of the other instances, the extra work required for the refinement improvements show a large reduction in run-time improvement. For `usbfunct_2`, the refinement improvements from Section IV-C led to no calls to the VERIFY procedure. This was caused by all valid solutions being found in the first analyzed time-window, as well as the improvements creating a strong initial abstraction for each time-window which filtered out any potential spurious solutions. However, without the refinement improvements spurious solutions are generated that must then be excluded through further refinement by calls to the VERIFY procedure.

B. Window Size

The next set of experiments profiles the effects when varying the width of the time-windows (w). The same nine instances from Section VI-A are used to ensure that all instances are able to run to completion without causing a time-out or memory-out condition for the given window sizes. The trace size is set to 100, while varying the width of each window for $w = 5, 10, 25, 50, 100$. In the case of $w = 100$, the problem does not use a time-windowing technique and models the entire trace directly.

TABLE II
REFINEMENT IMPROVEMENT EXPERIMENTS

	no optimizations			optimizations		
instance	time (s)	mem (MB)	verify	time (s)	mem (MB)	verify
ac97_1	47	703	14	54	722	14
ac97_2	132	716	115	81	775	29
div64_1	392	1366	110	275	1368	55
div64_2	206	1352	213	72	1270	6
fpu_1	108	1450	25	59	1475	3
rsdecoder_2	1354	729	1468	1582	861	552
usbfunct_1	106	1122	29	83	1102	20
usbfunct_2	88	1051	36	45	1040	0
vga_2	3544	1155	3733	657	1060	352

Table III shows the run-time and peak memory results of the experiments. The first column shows the instance name, followed by the run-time in seconds and peak memory for each of the different values of w. Overall, using a time-windowing technique dramatically reduces the peak memory usage of the algorithm resulting in an average decrease of 62.0%, 60.1%, 52.0%, and 36.4% in peak memory for $w = 5, 10, 25, 50$, respectively, compared to $w = 100$ (*i.e.*, no time-windowing). This decrease is directly related to the reduction in problem size due to using time-windowing. Figure 5 shows this relative decrease in peak memory for several instances.

As the window width w shrinks, the peak memory does not approach $\frac{1}{w}$ of the peak memory of directly modeling the entire trace. This is caused by two reasons. First, the data structures used to hold the RTL constructs, error trace, and internal netlist are always present in memory resulting in a lower bound regardless of window size. Secondly, the abstraction is still present in memory. Regardless of how many time-frames are explicitly modeled, the abstraction must still contain enough clauses in order to verify each solution. This places an additional constraint on the lower bound of the problem. Despite this bound, the time-windowing technique still produces dramatic decreases in peak memory.

When comparing the run-time, the results show an overall increase when compared to using no time-windowing ($w = 100$). For $w = 5, 10, 25, 50$, time-windowing increases run-time on average by 216.6%, 46.0%, 25.1%, and 183.54%, respectively. In some cases time-windowing produced faster run-times such as `div64_1`, and in other cases slower run-times such as `vga_2`. The former case occurs primarily when the abstract problem is small enough allowing the solver to run faster to outweigh the overhead of multiple SAT instances. In the latter case, multiple SAT instances increase the run-time, especially in cases where many calls to VERIFY are needed. Here explicit modeling of the trace ($w = 100$) produces significantly better results in terms of run-time due to the solver having access to the entire problem in memory. However, it should be noted that the primary benefit of time-windowing techniques are targeted for large industrial instances where it is not possible to model the entire trace explicitly. In these cases, time-windowing makes analysis possible while explicit modeling is unable to handle these problems.

The run-time performance of an instance across different window widths varies significantly as in the case of `vga_2` as well as the large difference in the average run-time. The run-

TABLE III
WINDOW SIZE EXPERIMENTS

	$w = 5$		$w = 10$		$w = 25$		$w = 50$		$w = 100$	
instance	time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)	time (s)	mem (MB)
ac97_1	80	692	54	722	39	875	39	1226	30	1893
ac97_2	104	699	81	775	51	923	43	1219	32	1883
div64_1	531	1250	275	1368	161	1832	444	2556	1782	4673
div64_2	244	1225	72	1270	75	1690	86	2400	106	4127
fpu_1	82	1279	59	1475	58	1904	61	2741	89	4521
rsdecoder_2	1875	873	1582	861	1494	919	802	1025	623	1342
usbfunct_1	124	961	83	1102	58	1285	78	1751	144	2841
usbfunct_2	72	954	45	1040	36	1304	35	1771	53	2838
vga_2	2709	1136	657	1060	671	1226	3639	1531	195	2230

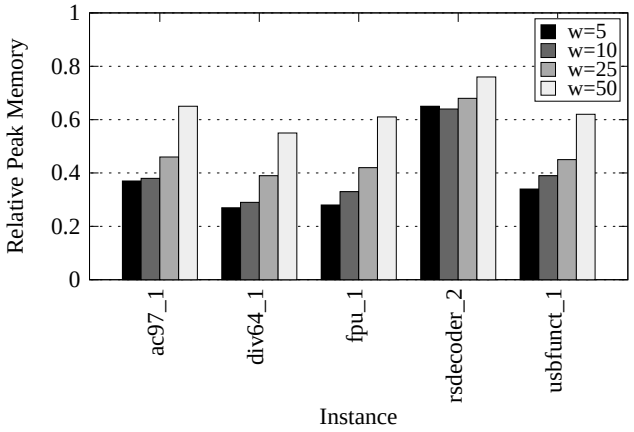


Fig. 5. Relative Memory for Different Window Sizes

time is dependent primarily on the number of SAT instances that are created and the size of each one of those instances. For smaller window widths, the size of each instance is small and runs quickly, but many small SAT instances are created (primarily in the VERIFY procedure). Inversely for longer window widths, the size of each instance is large and takes longer to run, but the number of SAT instances is significantly less. In addition to these two effects, the conflict driven refinement procedure is dependent on the solution found in the abstract problem. This adds additional variability, where in a worst cast, significantly more calls to the VERIFY procedure may be needed. However, as shown in the next section, using the flexible algorithm from Section V (Algorithm 3), this variability can be reduced.

C. Flexible Path Directed Debug

This set of experiments investigates the FLEXPATHDEBUG algorithm from Algorithm 3 with respect to the $skip\_limit$ parameter. Instances are chosen that are able to run to completion for all values of $skip\_limit$ without a time-out or memory-out condition. The trace is set to analyze 100 time-frames with $w = 10$. Four different sets of experiments are conducted for $skip\_limit = 0, 5, 10, \infty$. Note that $skip\_limit = \infty$ corresponds to the basic PATHDEBUG algorithm from Algorithm 1, while $skip\_limit = 0$ is comparable to the approximate time-windowing technique from [18] where no refinement is present.

Table IV shows the results of the experiments. The first column shows the instance name, followed by four sets of columns corresponding to the run-time, peak memory, number of verified and un-verified solutions (*i.e.,* $sols$ and $skips$ from Algorithm 3) for the four different values of $skip\_limit$.

When looking at the $skip$ column, small values of $skip\_limit$ have a large effect on the number of un-verified solutions. On average, the number of skipped solutions are 114.7%, 1.6% and 1.5% times the size of $skip\_limit = \infty$ (*i.e.,* the exact set of solutions) for $skip\_limit = 0, 5, 10$, respectively. This shows that refinement can greatly reduce the number of spurious solutions compared to past approximate time-windowing techniques such as [18] where no refinement is present. On the other hand, even for small values of $skip\_limit$ the number of skipped solutions is greatly reduced.

In terms of run-time, the average run-time reduction for $skip\_limit = 0, 5, 10$ relative to $skip\_limit = \infty$ is 52.4%, 9.72% and 7.5% respectively. As expected $skip\_limit = 0$ is able to show the greatest run-time reduction at the cost of greatly decreased quality of results. For other values of $skip\_limit$, the refinement can dramatically reduce the run-time as in the case of `comm_1` and `vga_2`, where a small subset of suspects causes the algorithm to spend significantly more time in the VERIFY procedure. This worst-case scenario is exactly the case PATHDEBUG aims to address.

When looking at peak-memory, the average reduction is 5.4%, 0.6%, and 0.5% for $skip\_limit = 0, 5, 10$ respectively. When no refinement is present ($skip\_limit = 0$), a slight decrease in peak memory is shown due to the fact that the abstraction is smaller compared to when refinement is present.

D. Trace Length

The final set of experiments investigates the maximum length of error trace each technique can analyze until either a time-out, memory-out or entire trace is analyzed. The experiments compares the WINDOWEXPANSION time-windowing technique from [18] (a complete debugging algorithm), the basic PATHDEBUG algorithm from Section III, and the flexible algorithm FLEXPATHDEBUG from Section V with $skip\_limit = 5$. All instances derived from designs in Table I are used in this set of experiments. The window size is set to $w = 10$.

Table V shows the results of experiments. The first column shows the instance name, while the next eight columns show the run-time, peak memory, number of solutions, and number of cycles analyzed for the WINDOWEXPANSION and PATHDEBUG algorithms respectively. The last five columns

TABLE IV
FLEXIBLE PATHDEBUG EXPERIMENTS

	$skip\_limit = 0$				$skip\_limit = 5$				$skip\_limit = 10$				$skip\_limit = \infty$			
instance	time (s)	mem (MB)	sols	skip	time (s)	mem (MB)	sols	skip	time (s)	mem (MB)	sols	skip	time (s)	mem (MB)	sols	skip
ac97_1	27	720	15	11	54	722	26	0	54	722	26	0	54	722	26	0
ac97_2	25	722	9	11	81	775	11	0	81	775	11	0	81	775	11	0
div64_1	146	1334	23	17	275	1368	36	0	275	1368	36	0	275	1368	36	0
div64_2	63	1260	18	3	72	1270	21	0	72	1270	21	0	72	1270	21	0
fpu_1	50	1444	29	3	59	1475	32	0	59	1475	32	0	59	1475	32	0
rsdecoder_2	54	579	132	275	1582	861	381	0	1582	861	381	0	1582	861	381	0
comm_1	1139	5152	32	201	3442	5176	74	2	3986	5248	74	2	6169	5475	74	0
comm_2	1062	5190	32	203	3449	5315	75	2	3981	5318	75	2	6168	5373	75	0
usbfunct_1	52	1080	98	11	86	1099	105	1	83	1102	105	0	83	1102	105	0
usbfunct_2	45	1040	10	0	45	1040	10	0	45	1040	10	0	45	1040	10	0
vga_2	101	1033	28	195	514	1063	46	5	580	1053	46	5	657	1060	46	0

show the run-time, peak memory, number of verified and un-verified solutions (*i.e., sols* and *skips* from Algorithm 3), and number of analyzed cycles.

When comparing the number of cycles analyzed, PATHDEBUG and FLEXPATHDEBUG are able to analyze 35.5% and 64.6% more cycles on average than WINDOWEXPANSION. This shows one of the main benefits of the proposed technique: the ability to analyze debugging instances that were previously too large to analyze. Moreover, the increased trace length analyzed for PATHDEBUG and FLEXPATHDEBUG is achieved with significantly less memory. This can be seen by WINDOWEXPANSION experiments causing 12 memory-out conditions compared to 2 for the path-based algorithms.

Figure 6 shows the cycles analyzed for several different instances. In most cases, such as `fdct_1` and `comm_1`, the FLEXPATHDEBUG is able to analyze many more cycles compared to WINDOWEXPANSION. In contrast, in a few cases such as `mips_sys_1`, WINDOWEXPANSION is able to analyze more cycles. This is due to the computation required by the VERIFY procedure, where the SAT instances created are more difficult to solve. However, the FLEXPATHDEBUG algorithm can be used with a different value of *skip_limit* parameter to overcome some of these hard-to-solve cases.

For the FLEXPATHDEBUG algorithm, the absolute number of un-verified solutions is relatively small allowing for a good trade-off between performance and quality of results. Moreover, the additional cycles analyzed allowed FLEXPATHDEBUG to find on average 11.6% more solutions (both verified and un-verified) compared to WINDOWEXPANSION. This shows that the additional cycles analyzed do indeed contain some additional solutions, which could potentially be needed to find the underlying bug, a desirable side-effect for the proposed methodology.

VII. CONCLUSION

Debugging has become a significant bottleneck in the VLSI design flow due to its resource intensive nature. The problem is further exacerbated by the ever growing size of modern designs and their error traces. This work introduces a novel path directed abstraction and refinement framework that aims to manage excessive error trace lengths. It uses a time-windowing framework to iteratively analyze a sliding window of the error trace. Non-modeled portions of the trace are approximated using a novel path directed abstraction that conceptually represents structural circuit paths. This representation provides a means for analyzing a sliding window of the error trace, as well providing a basis for efficient refinement. The result is an algorithm that can significantly reduce the memory requirements of debugging industrial instances, while mitigating the incomplete results of past techniques. Experimental results on industrial designs demonstrate that 64.6% longer error traces can be analyzed with the proposed approach while significantly reducing the memory usage.

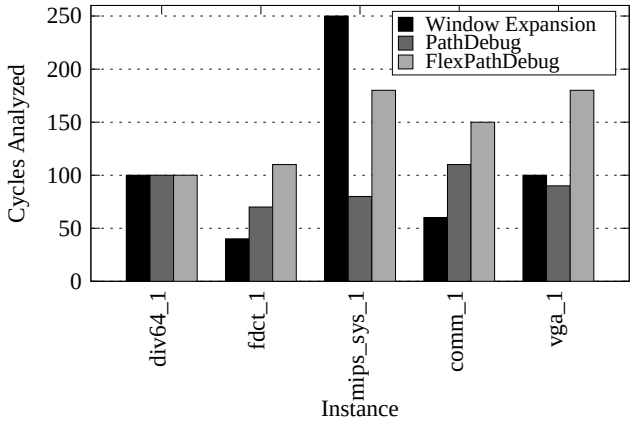


Fig. 6. Length of Error Trace Analyzed

REFERENCES

[1] H. Foster, "From Volume to Velocity: The Transforming Landscape in Function Verification," in *Design Verification Conference*, 2011.
[2] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.
[3] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, 2003.
[4] "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800-2009*, pp. 1–1285, 2009.
[5] "IEEE Standard for Property Specification Language (PSL)," *IEEE Std 1850-2010*, pp. 1–171, apr. 2010.
[6] R. K. Ranjan, C. Coelho, and S. Skalberg, "Beyond verification: leveraging formal for debugging," in *Design Automation Conf.*, 2009, pp. 648–651.
[7] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," ser. Advances In Computers, no. 60, 2003.
[8] K. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, 2003.

TABLE V
TRACE LENGTH EXPERIMENTS

	WINDOWEXPANSION				PATHDEBUG				FLEXIBLE PATHDEBUG (r=5)				
instance	time (s)	mem (MB)	sols	debug cyc	time (s)	mem (MB)	sols	debug cyc	time (s)	mem (MB)	sols	skip	debug cyc
ac97_1	2812	**MO**	46	600	**TO**	3830	51	1280	**TO**	3830	51	0	1280
ac97_2	4238	**MO**	122	540	**TO**	3955	123	1350	**TO**	3955	123	0	1350
fxu_1	790	**MO**	174	10	925	**MO**	174	40	941	**MO**	174	0	40
fxu_2	295	**MO**	37	10	354	**MO**	37	10	337	**MO**	37	0	10
div64_1	419	3971	38	100	**TO**	1551	36	100	1304	1410	36	8	100
div64_2	383	3517	21	90	72	1301	21	90	72	1301	21	0	90
fdct_1	2491	**MO**	78	40	**TO**	4798	80	70	**TO**	5800	73	13	110
fdct_2	3476	**MO**	69	40	**TO**	4214	69	50	**TO**	5291	58	19	90
fpu_1	797	**MO**	35	210	340	2419	37	230	340	2419	37	0	230
fpu_2	40	1944	5	30	27	1000	5	30	27	1000	5	0	30
mips_sys_1	2375	7454	126	250	**TO**	1298	103	80	**TO**	2082	104	4	180
mips_sys_2	6104	**MO**	260	280	**TO**	1242	185	40	**TO**	2204	249	9	150
rsdecoder_1	**TO**	1849	81	60	**TO**	1216	75	60	**TO**	1339	69	24	100
rsdecoder_2	548	1093	384	100	2473	1051	384	100	**TO**	1051	384	0	100
comm_1	990	**MO**	72	60	**TO**	5515	74	110	**TO**	6356	74	2	150
comm_2	1113	**MO**	73	60	**TO**	5490	75	110	**TO**	6477	75	2	160
usbfunct_1	3788	**MO**	105	410	351	2016	105	410	343	2031	105	1	410
usbfunct_2	4248	**MO**	106	410	2777	3186	106	720	**TO**	3186	106	0	720
vga_1	**TO**	4489	20	100	**TO**	1890	20	90	**TO**	2392	20	17	160
vga_2	**TO**	3790	46	240	**TO**	1422	46	140	**TO**	1630	46	37	180

[9] A. R. Bradley, "SAT-based model checking without unrolling," in *Int'l Conf. on Verification, Model Checking, and Abstract Interpretation*, 2011, pp. 70–87.

[10] Y.-C. Hsu, B. Tabbara, Y.-A. Chen, and F. Tsai, "Advanced techniques for RTL debugging," in *Design Automation Conf.*, 2003, pp. 362–367.

[11] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[12] N. K. Jha and S. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2002.

[13] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[14] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita, "A formal approach for debugging arithmetic circuits," in *IEEE Trans. on CAD*, vol. 28, no. 5, May 2009, pp. 742–754.

[15] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.

[16] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Trans. on CAD*, vol. 28, no. 10, pp. 1597–1608, 2009.

[17] B. Keng and A. Veneris, "Managing complexity in design debugging with sequential abstraction and refinement," in *ASP Design Automation Conf.*, 2011, pp. 479–484.

[18] B. Keng, S. Safarpour, and A. Veneris, "Bounded Model Debugging," *IEEE Trans. on CAD*, vol. 29, pp. 1790–1803, November 2010.

[19] C. S. Zhu, G. Weissenbacher, and S. Malik, "Post-silicon fault localisation using maximum satisfiability and backbones," in *Formal Methods in CAD*, 2011, pp. 63–66.

[20] B. Keng and A. Veneris, "Path directed abstraction and refinement in SAT-based design debugging," in *Design Automation Conf.*, 2012, pp. 947–954.

[21] OpenCores.org, "http://www.opencores.org," 2007.

[22] A. Veneris, B. Keng, and S. Safarpour, "From RTL to silicon: The case for automated debug," in *ASP Design Automation Conf.*, 2011, pp. 306–310.

[23] M. S. Abadir, J. Ferguson, and T. Kirkland, "Logic verification via test generation," *IEEE Trans. on CAD*, vol. 7, pp. 172–177, 1988.

[24] M. F. Ali, S. Safarpour, A. Veneris, M. Abadir, and R. Drechsler, "Post-verification debugging of hierarchical designs," in *Int'l Conf. on CAD*, 2005, pp. 871–876.

[25] N. Eén and N. Sörensson, "Translating pseudo-boolean constraints into SAT," in *JSAT*, vol. 2, 2006, pp. 1–26.

[26] Y.-S. Yang, A. Veneris, and N. Nicolici, "Automating data analysis and acquisition setup in a silicon debug environment," *IEEE Trans. on VLSI Systems*, vol. PP, no. 99, pp. 1–14, 2011.

[27] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[28] E. Clarke, A. Gupta, and O. Strichman, "SAT-based counterexample-guided abstraction refinement," *IEEE Trans. on CAD*, vol. 22, no. 7, pp. 1113–1123, 2004.

[29] J. Marques-Silva, M. Janota, and I. Lynce, "On computing backbones of propositional theories," in *European Conference on Artificial Intelligence*, 2010, pp. 15–20.

[30] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Int'l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.

[31] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

PLACE PHOTO HERE

Brian Keng received the B.A.Sc. degree in computer engineering from the University of Waterloo and the M.A.Sc. degree in computer engineering from the University of Toronto.

He is currently a Ph.D. student at the University of Toronto in the Department of Electrical and Computer Engineering. His research interests are in CAD for design debugging, test and verification of digital circuits and systems. In particular, he is interested in topics relating to the theory and application of formal methods.

PLACE PHOTO HERE

Andreas Veneris received a Diploma in Computer Engineering and Informatics from the University of Patras in 1991, an M.S. degree in Computer Science from the University of Southern California, Los Angeles in 1992 and a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1998. In 1998 he was a visiting faculty at the University of Illinois until 1999 when he joined the Department of Electrical and Computer Engineering and the Department of Computer Science at the University of Toronto where today he is an Associate Professor. His research interests include CAD for debugging, verification, synthesis and test of digital circuits/systems, and combinatorics. He has received several teaching awards and a best paper award. He is the author of one book and he holds three patents.

He is a member of ACM, IEEE, AAAS, Technical Chamber of Greece, Professionals Engineers of Ontario and The Planetary Society.