Scalable Specification Mining for Verification and Diagnosis

Wenchao Li
UC Berkeley
wenchaol@eecs.berkeley.edu

Alessandro Forin
Microsoft Research
sandrof@microsoft.com

Sanjit A. Seshia
UC Berkeley
sseshia@eecs.berkeley.edu

ABSTRACT

Effective system verification requires good specifications. The lack of sufficient specifications can lead to misses of critical bugs, design re-spins, and time-to-market slips. In this paper, we present a new technique for mining temporal specifications from simulation or execution traces of a digital hardware design. Given an execution trace, we mine recurring temporal behaviors in the trace that match a set of pattern templates. Subsequently, we synthesize them into complex patterns by merging events in time and chaining the patterns using inference rules. We specifically designed our algorithm to make it highly efficient and meaningful for digital circuits. In addition, we propose a pattern-mining diagnosis framework where specifications mined from correct and erroneous traces are used to automatically localize an error. We demonstrate the effectiveness of our approach on industrial-size examples by mining specifications from traces of over a million cycles in a few minutes and use them to successfully localize errors of different types to within module boundaries.

Categories and Subject Descriptors

B.7.2 [**Design Aids**]: Verification; B.8.1 [**Reliability, Testing and Fault-Tolerance**]

General Terms

Algorithms, Experimentation, Verification

Keywords

Formal specification, verification, assertions, diagnosis, debugging, error localization, post-silicon validation

1. INTRODUCTION

Formal specifications can precisely capture a system's desired behavior. One can then leverage verification techniques such as model checking [8] or assertion-driven simulation to ensure the correctness of the system. However, the difficulty of manually creating a complete set of formal properties (assertions) and of maintaining those properties through design changes and evolution has significantly hindered the wide-spread adoption of formal specifications. There is therefore a need for scalable techniques for automatically generating formal specifications.

Specification mining, a promising alternative to manually writing specifications, is the process of extracting specifications, either statically from the description of a system, or dynamically from its executions. The mined specifications in turn allow us to better understand the system, verify its correctness, and manage possible evolutional changes. In this paper, we present a new approach to scalably mine recurring patterns from existing simulation traces. These patterns can then be examined by the engineer to see whether they match the designer's intent and can be checked with further verification. The intuition is that frequent patterns are likely to be true. Figure 1 illustrates the high-level tool flow. Our tool takes a set of traces and optionally a user-defined event definition as input, and generates a set of behavioral patterns which are true in the trace as output. A trace is a sequence of events ordered by time of occurrence. Events in this case are the valuations of a set of signals in a circuit. Given the trace, we match it to a library of parametric patterns. The matching algorithm is discussed in detail in Section 4. We also provide a post-processing ranking module to produce a heuristically-ranked list of the most interesting properties.

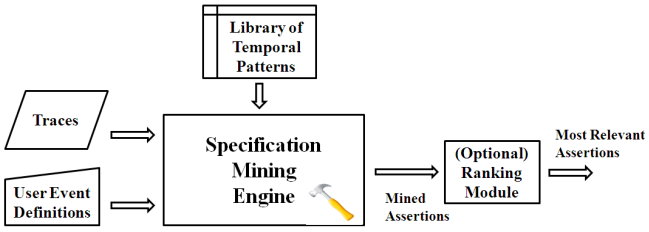

Figure 1: Specification Mining for Verification

Specification mining not only helps to automate coverage-driven simulation or formal verification, it can also provide useful information for diagnosis. We propose a specification mining-based diagnosis framework that can be used to simultaneously understand the error and locate it. Figure 2 shows how our specification-mining engine is used as a subroutine to determine assertions that distinguish between a normal trace and an error trace. Distinguishing assertions are those that exist in one trace but not in the other. After finding these patterns, we apply a ranking procedure to pinpoint the error to the module in which the fault occurs. We show in Section 6 that our technique is able to successfully localize faults in both RTL (e.g. programming mistakes such as erroneous state machine transitions) as well as faults that may arise from physical defects (e.g., stuck-at or transient faults).

To summarize, we make the following key contributions:

- A new dynamic specification mining technique especially designed for general digital circuits. Our tool SAM (Scalable Assertion Miner) efficiently mines non-trivial specifications and is highly scalable: for a design with over 20,000 signals, over 1000 properties were mined in under a minute;
- A novel trace-diagnosis technique based on specification mining that achieves good localization accuracy for large circuits.

The paper is organized as follows. Section 2 surveys related literature on specification mining. Section 3 introduces the main concepts and formally defines the problem. Section 4 describes the algorithms for mining specification in digital circuits. Section 5 de-

DAC 2010, June 13-18, 2010, Anaheim, California, USA.

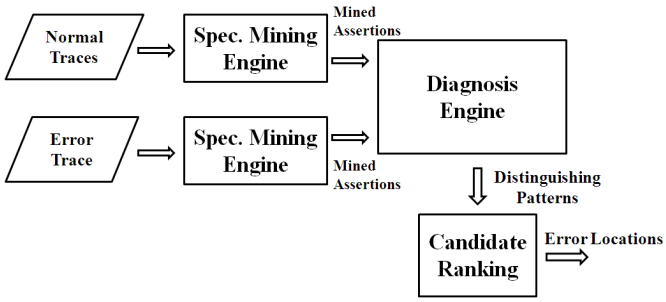

Figure 2: Specification Mining for Diagnosis

scribes a pattern-mining based technique for diagnosis. Section 6 presents experimental results. We conclude in Section 7.

2. RELATED WORK

The study of automatically generating specifications goes back as early as 1974 [6][26]. Many techniques have been recently proposed to automatically reverse-engineer specifications from a programs [25][16][3][28][15][13]. These specifications can be simple predicates or temporal specifications which specify the ordering of events, or rules of API usage. The generated specifications can then be used to formally verify a program's correctness, to assist in debug [27], or to detect malicious behaviors [7].

Many techniques seek to learn specifications dynamically from an execution trace (or a set of traces). Daikon [13] is one of the earliest tools that mine single-state invariants or pre-/post-conditions in programs. In contrast, we focus on mining temporal properties for hardware designs in this work. Most existing mining tools produce temporal properties in the form of automata. Automata-based techniques generally fall into two categories. The first class of methods learn a single complex specification (usually as a finite automaton) over a specific alphabet, and then extract simpler properties from it. For instance, Ammons et al. [3] first produce a probabilistic automaton that accepts the trace and then extract from it likely properties. However, learning a single finite state machine from traces is NP-hard [17]. To achieve better scalability, an alternative is to first learn multiple small specifications and then post-process them to form more complex state machines. Engler et al. [11] first introduce the idea of mining simple alternating patterns. Several subsequent efforts [16, 27, 28, 15] built upon this work. For example, Javert [15] locates all instances of the alternating pattern $(a\ b)^*$ and a resource usage pattern $(a\ b^*\ c)^*$. The tool then composes these patterns into larger ones by using a set of inference rules. Our work is similar to Javert, however we focus on additional patterns that are meaningful for digital circuits and we provide a merging procedure that composes patterns in time.

Specifications can also be generated by reasoning about the program statically. For example, Alur et al. [2] proposes the use of predicate abstraction together with automata learning to automatically synthesize interface specifications for Java classes. Static and dynamic analyses complement each other. We refer the readers to [12] for a detailed comparison of the two techniques. Additionally, we also note that, for hardware designs, static analysis is particularly challenging because it is difficult to infer causal dependencies between events across multiple cycles from the structure of the RTL code.

Various circuit-specific mining techniques have been proposed for hardware-specific properties. The IODINE tool [18] mines simple likely invariants such as one-hot encodings or fixed-delay pairs. Fey and Drechsler [14] present an approach to mine repeated patterns where patterns are valuations of signals at various time steps (e.g. $s_t = 1 \wedge s_{t+1} = 0$). While their approach is general, the timing requirement can be too strict for complex interactions and it deals with only a small set of signals over a predefined interval each round. The Dianosis [24] tool mines more complex properties from simpler properties from an OVL. Isaksen and Bertacco [19] propose the use of inferred boundary labels to generate transaction diagrams from a trace. Their methodology is particularly suitable for analyzing protocols. Our approach mines a class of general temporal properties for digital circuits in a scalable manner and, to our knowledge, is the first to effectively use these mined properties for fault diagnosis.

3. CONCEPTS AND DEFINITIONS

This section formally introduces the dynamic specification mining problem for digital circuits and describes the types of patterns that we mine.

Let S be the set of signals in a digital circuit. Each $s \in S$ can be either a register or a wire. We use $v_{s,t}$ to denote the valuation of s at time t. (We restrict ourselves to valuations of signals at rising edges of their corresponding clocks).

Definition 3.1 (Event) An event e is a tuple $< \vec{s}, \vec{v}, t >$, where $\vec{s}$ is a set of signals and $\vec{v}$ is the corresponding valuations at time t.

Note that we do not define events as assignments of signals across cycles, but this can be addressed by introducing suitable user-defined events. The alphabet Σ is the set of distinct events. A trace τ is a set of events (partially) ordered by their time of occurrence. A slice η of a trace is defined as the set of events that occur at the same time. We limit ourselves to finite-length traces in this work.

Definition 3.2 (Projection) The projection π of a trace τ over an alphabet Σ, $\pi(\tau)$ is defined as τ with all events not in Σ deleted.

Definition 3.3 (Specification Pattern) A specification pattern is a finite state automaton $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the set of input events, $\delta : Q \rightarrow \Sigma \times Q$ is the transition function, q_0 is the single starting state, and F is the set of accepting states. A pattern is satisfied over a trace τ with alphabet $\Sigma \supseteq \Sigma$ iff $\pi(\tau) \in \mathcal{L}(A)$, i.e. the projection of the trace on the pattern alphabet is in the language of the pattern automaton.

Definition 3.4 (Binary Pattern) A binary pattern is defined as a specification pattern with an alphabet size of 2. A binary pattern between events a and b is denoted as $a\ \mathbf{R}\ b$.

Our specification pattern miner takes a trace, a set of pattern templates and optionally a manually-provided event definition as input, and produces all pattern instances that are satisfied over the trace as output. In this paper, we focus on the mining of specifications without an explicit event definition, instead using the notion of a delta event, defined below.

We adapt our techniques to some special characteristics of digital circuits to make them both efficient and meaningful. One key difference between a digital circuit and a software program is that a digital circuit is a concurrent process, in which multiple events can occur at every clock cycle. In traditional software specification mining problems, the size of the problem is relatively small in both the length of the trace and the size of the event alphabet; in contrast, we have to handle traces of millions of cycles in length, potentially thousands of events at every cycle, and generally a large alphabet. In addition, we need to modify the definition of events to make the analysis meaningful. For example, an interesting event can be the start of a request (transition of value 0 to value 1), but the request signal can stay at 1 for several cycles until a response is received. We introduce the notion of delta event, formally defined as follows:

Definition 3.5 (Delta Event) A delta event, denoted Δe, is an event such that at least one of its constituent signals changes value from the previous valuation, i.e. $e \doteq < \vec{s}, \vec{v}, t >$ such that $\exists\ v_{s,t} \in \vec{v}, v_{s,t} \neq v_{s,t-1}$.

In mining patterns, we restrict ourselves to delta events. Our mined patterns can be expressed succinctly in linear temporal logic (LTL) [21] or as regular expressions. For example, we can express that every request must be eventually followed by a grant as "$\mathbf{G}$ (request $\rightarrow \mathbf{F}$ grant)" in LTL, where the operator $\mathbf{G}$ specifies that globally at every point in time a certain property holds, and F specifies that a property holds either currently or at some point in the future. The binary patterns mined by our tool are listed below.

Alternating (A) An alternating pattern between two delta events Δa and Δb is true when each occurrence of Δa alternates with an occurrence of Δb. Note that this does not mean Δb follows Δa immediately in the next cycle. This pattern can be described by

the regular expression $(\Delta a\ \Delta b)^*$. We denote this pattern as $a\ \mathbf{A}\ b$. Figure 3 shows the corresponding finite automaton (self-transitions are such that the automaton is deterministic).

Until (U) The until pattern can be used to describe behaviors such as "the request line stays high until a response is received." Figure 4 shows a trace where this pattern is satisfied. Formally, the LTL formula is "$\mathbf{G}\ (\Delta a \rightarrow \mathbf{X}\ (a\ \mathbf{U}\ \Delta b))$."

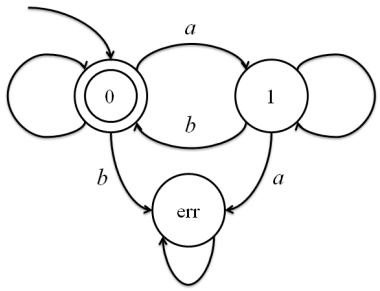

Figure 3: Finite Automaton for the Alternating Pattern

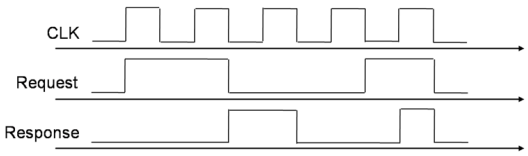

Figure 4: Request stays high until a response is received.

Next (X) The next pattern corresponds to the LTL formula "$\mathbf{G}(\Delta a \rightarrow \mathbf{X}\ \Delta b)$." We denote it as $a\ \mathbf{X}\ b$. Note that $a\ \mathbf{X}\ b$ implies $a\ \mathbf{U}\ b$. One can easily generalize this pattern to fixed-delay pairs.

Eventual (F) The eventual pattern can be described by the LTL formula "$\mathbf{G}\ (\Delta a\ \rightarrow\ \mathbf{X}\ \mathbf{F}\ \Delta b)$." We denote this as $a\ \mathbf{F}\ b$. Note that $a\ \mathbf{A}\ b$, $a\ \mathbf{U}\ b$ and $a\ \mathbf{X}\ b$ all imply $a\ \mathbf{F}\ b$. Since typically many such patterns appear in a trace, the user can put a lowerbound on the number of times that it appears and an upperbound on the time of separation between the two events base on their knowledge of the design, to extract more relevant behaviors. We also output $a\ \mathbf{X}\ b$ from $a\ \mathbf{F}\ b$ based on the timing bounds.

Timing bounds are often crucial to specify a behavior. For example, a system may require every two requests to be separated by *at least* 3 cycles and the response to be received *within* 5 cycles of issuing a request. In this work, we track such timing bounds during the mining of patterns.

Our tool first mines all binary pattern instances that are satisfied over the trace given the templates. Subsequently, the simple patterns are merged in time and then synthesized to form more complex patterns using the inference rules. These procedures are described in detail in Section 4.2.

4. MINING ALGORITHMS

We exploit the highly modular nature of hardware design to keep the problem tractable. For scalability, we partition the trace by module into many disjoint sub-traces and analyze them separately. The rest of this section describes our techniques in detail. Section 4.1 describes the mining algorithm. Section 4.2 describes how complex specifications can be synthesized. Section 4.3 discusses some useful specification ranking metrics.

4.1 Specification Mining

We adopt the approach of mining small automata. Gabel and Su [16] formalized the problem of mining parameteric patterns such as the alternating pattern we consider (but not general LTL) and showed that it is NP-hard through reduction from the Hamiltonian Path problem. Therefore, it makes sense to mine patterns with a small pattern alphabet size to avoid the potential worst-case exponential blow-up. Their approach builds on that of Perracotta [28] which requires $O(nk)$ space and $O(n^{k-1}l)$ time for an input alphabet size of n, a pattern alphabet size of k, and a trace of length l. Gabel and Su also propose the use of Binary Decision Diagrams (BDDs) [5] to improve the tractability of the problem. However, while they show some speed-up using the symbolic technique, the input alphabet size is still limited in practice to 3. In addition, the performance of BDD-based techniques depends heavily on having a good variable ordering, and finding the optimum variable ordering is again NP-hard [4].

Our algorithm mines binary patterns with timing bounds as discussed in Section 3. We adopt the approach in [28] but extend it to handle traces with multiple events at the same cycle and to mine richer binary patterns. The algorithm is briefly outlined below. All the events are delta events here, which we write just as e. The algorithm for mining $a\ \mathbf{R}\ b$ is shown below, where a and b are events that contain a single binary variable, and $\mathbf{R}$ denotes one of $\mathbf{A}$, $\mathbf{X}$, $\mathbf{F}$, or $\mathbf{U}$.

Algorithm 1 Mine $a\ \mathbf{R}\ b$

```
1:  τ_Δ = Preprocess(τ)
2:  {τ_1, ..., τ_M} = Modularize(τ_Δ)
3:  for each τ_m ∈ {τ_1, ..., τ_M} do
4:      S = Create_Event_List(τ_m)
5:      T = Allocate_Pattern_Table(S)
6:      for each slice η ∈ τ_m do
7:          Update(T,η,R)
8:      end for
9:      Output_Patterns(T)
10: end for
```

The main data structure used by the algorithm is the pattern table T. We have one pattern table for each module m and each pattern type $\mathbf{R}$. Table T has as many rows (and columns) as the number of delta events for module m. While simulating the pattern automaton for $e_i\ \mathbf{R}\ e_j$, the entry (i, j) in T stores the current state of the automaton.

The algorithm works as follows. We first preprocess the trace into a delta event trace (line 1). We then partition the trace into a set of traces according to the module tree and analyze them separately (line 2). This allows us to effectively cope with the memory overhead of allocating a quadratic space for the pattern table T (lines 5). In our tool, one can heuristically choose the level of modularization – small sub-modules are merged so that the mined specifications are not too local. Next, the algorithm updates the associated pattern automaton according to the type of binary relations $\mathbf{R}$ for each slice of the trace (lines 7). We first update row $e\ \mathbf{R}\ *, \forall e \in \eta$, then update column $* \mathbf{R}\ e, \forall e \in \eta$. The update rule is determined by the transition function of the finite-state automaton corresponding to $\mathbf{R}$. One feature of our work that is different from [28] is that since we are mining from a concurrent process, there are normally multiple events at the same slice. When performing iterative row and column updates on the pattern table, we need to avoid updating the same entry multiple times at the same cycle that have already been processed in the current cycle. Figure 5 illustrates this concept when the current slice contains events e_i and e_{i+1}. Notice that when updating T for e_{i+1}, the entry $(i, i + 1)$ remains unchanged because e_i was already processed. During the update procedure, we also compute timing bounds on-the-fly for each pattern.

Complexity: Given an input trace τ with signals $s_1, s_2 \ldots s_N$, the preprocessing step converts the trace to a delta event trace τ_Δ with n delta events. This greatly enhances the scalability of our technique because typically $|\tau_\Delta| \ll |\tau|$ and $n \ll |\Sigma_i 2^{|s_i|}|$. Modularization further decomposes the problem such that in a design with M modules, for each module m we mine from a trace $\tau_m(|\tau_m| \leq |\tau_\Delta|)$, each with $n_m(n_m < n)$ delta events. Hence, the final algorithm requires $O(\tilde{n}_m^2)$ space and $O(\Sigma_m q n_m |\tau_m|)$ time, where $\tilde{n}_m$ is $\max_m n_m$ and q is the average number of delta events per slice.

4.2 Specification Summarization

Specification summarization eliminates redundant specifications and helps users to understand specifications better. We perform three summarization procedures – *event merging, pattern chaining* and *graph composition*.

Pattern Merging: We first merge patterns by matching their time of occurrence and conjoining events that always occur together. For example, if both $(\Delta a\,\Delta b)^*$ and $(\Delta a\,\Delta c)^*$ are true at the same time points, and Δb and Δc always occur at the same time, we can merge them to form $(\Delta a\ (\Delta b \wedge \Delta c))^*$. During the pattern mining phase, we maintain a timestamped list of occurrences for the events in each pattern. Alternative, one can re-simulate the mined patterns on the trace to obtain the timestamps progressively. These lists are processed during the pattern merging phase by first partitioning the set of patterns into mergeable subsets, and then merging the patterns within each partition by conjoining events that always appear together. The algorithm is outlined below.

Algorithm 2 Merge Patterns

```
1: P : list of pattern instances
2: L_p : list of timestamps of p ∈ P
3: ind_p :index of the current timestamp in L_p
4: min_t : min {L_p[ind_p]}, ∀p ∈ P
5: function Partition(P)
6: if (|P| = 1) ∨ (ind_p = |L_p|, ∀p ∈ P) then
7:    return {P}
8: else if ∃p, ind_p = |L_p| then
9:    return append({p|ind_p = |L_p|}, Partition({p|ind_p ≠ |L_p|})
10: else
11:    P' = {p|L_p[ind_p] = min_t})
12:    ∀p' ∈ P', ind_p' = ind_p' + 1
13:    return append(Partition(P'), Partition({p|L_p[ind_p] ≠ min_t}))
14: end if
15: end function
16: function Merge(P)
17: {P_1, ..., P_k} = Partition(P)
18: Merge events to form a single pattern in each P_i
19: end function
```

The algorithm essentially first partitions the patterns into sets of patterns such that the occurrences of events all match in time for each set, and then merge these patterns into a single one. The pattern merging procedure is particularly useful when no event definition is manually provided. A hardware module in a typical CPU core can have hundreds of signals running in parallel and many of them are highly correlated. In Section 6, we demonstrate that this simple recursive procedure significantly reduces the number of mined specifications and in practice generates better quality specifications for the end user.

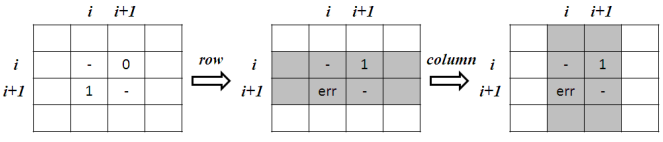

Figure 5: Iterative Row and Column Updates in T for $e_i\ \mathbf{R}\ e_j$

Pattern Chaining: After we merge the patterns in parallel, we repeatedly apply a set of inference rules to the results to obtain even more complex patterns. The current inference rules for chaining binary relational patterns are illustrated below.

Alternating Pattern Chaining Rule (adapted from [15]):

$$\frac{(\Delta a\ \Delta b)^* \quad (\Delta b\ \Delta c)^* \quad (\Delta a\ \Delta c)^*}{(\Delta a\ \Delta b\ \Delta c)^*}$$

Eventual Pattern Chaining Rule:

$$\frac{\mathbf{G}\ (\Delta a \rightarrow (\mathbf{X}\ \mathbf{F}\ b)) \quad \mathbf{G}\ (\Delta b \rightarrow (\mathbf{X}\ \mathbf{F}\ \Delta c))}{\mathbf{G}\ (\Delta a \rightarrow \mathbf{X}\ \mathbf{F}\ (\Delta b \rightarrow \mathbf{X}\ \mathbf{F}\ \Delta c))}$$

Until Pattern Chaining Rule:

$$\frac{\mathbf{G}\ (\Delta a \rightarrow \mathbf{X}\ (a\mathbf{U}\ \Delta b)) \quad \mathbf{G}\ (\Delta b \rightarrow \mathbf{X}\ (b\mathbf{U}\ \Delta c))}{\mathbf{G}\ (\Delta a \rightarrow \mathbf{X}\ (a\mathbf{U}\ (b\mathbf{U}\ c)))}$$

Graph Composition: We further graphically compose the resulting patterns. In each graph, a node represents an event and an edge from node a to node b represents a binary pattern $a\,\mathbf{R}\,b$. Every disjoint sub-graph represents a complex behavior amongst its constituent events. In Section 6, we will show that such visualization helps in understanding the behaviors of a design.

4.3 Specification Ranking

The process of merging and chaining also allows us to further sieve through the set of specifications for the most interesting ones. For example, if one is interested in complex interactions, we can output only patterns with alphabet size greater than a user-specified threshold. In our tool, we also rank patterns according to their *frequency of occurrences* , *time of first occurrence* and *statistics (e.g. variance) in terms of separation between constituent events*.

5. FAULT DIAGNOSIS

We now consider the problem of debugging an error given a set of correct traces and a single error trace. Our goal is to localize the error to the part of the circuit where the error occurred. For transient errors, another goal is to localize in time, i.e., to find the approximate time of occurrence of a transient error. One potential application is post-silicon debugging where bugs are difficult to diagnose due to limited observability, reproducibility, and possible dependence on physical parameters.

A number of diagnosis approaches have been proposed in the classic AI literature. As observed by Console et al [9], these approaches either require models that describe the correct behavior of the system or they need models for the abnormal (faulty) behaviors. Our approach is similar to the consistency-based methods [10]. In the traditional consistency-based reasoning approach, if a system can be described using a set of constraints, then diagnosis can be accomplished by identifying the set (often minimal) of constraints that must be excluded in order for the remaining constraints to be consistent with the observations. While this approach does not require knowledge of how a component fails (a fault model), it requires a reasonably complete specification of the correct system. In the EDA literature, while there has been substantial work on fault diagnosis and debugging, to our knowledge none of the work has made use of automatically mined specifications.

Our approach is similar to the consistency-based method but we do not need to start with a set of specifications. Instead, we mine specifications from traces and use them to localize the errors. Our approach does not directly make use of the RTL description for diagnosis (other than the module hierarchy), which makes it scalable and appealing for post-silicon debug. In addition, we do not need to time-align the correct traces with the incorrect trace. The trace diagnosis problem can be described as the following:

Given a correct trace τ jointly produced by a set of modules M, and an incorrect trace τ' over the same alphabet Σ produced by M' such that some $m \in M'$ is erroneous (different from its counterpart in M), the diagnosis task is to localize the error to m.

We assume that the error is detectable at the system level. This means that there exists a mechanism to label a trace (erroneous or otherwise) with respect to some correctness criteria. Typically, such a mechanism relies on checking some end-to-end behaviors or observing whether an exception is thrown in software.

Consistency is defined with respect to the specifications mined from the correct trace. Specifically, consistency is violated if

- A pattern is observed in the error trace but it fails at some point in the correct trace; or
- A pattern is observed in the correct trace but it fails at some point in the error trace.

A pattern that violates consistency is termed a *distinguishing pattern*. An error can propagate to other modules and in turn cause more erroneous behaviors later. In light of this, we rank the mined distinguishing patterns by the time of first violations – the point where a pattern is expected to hold but does not. The module which the top ranked pattern belongs to gives the localization result. The time of the pattern's first violation also gives the time-localization

in the case of transient faults. Since the pattern itself describes a specific erroneous behavior, our approach not only localizes the error, but can also produces useful insights about the error.

6. EXPERIMENTAL RESULTS

We have implemented the proposed approach in a tool called SAM (Scalable Assertion Miner). In this section, we present case studies illustrating that our approach has the following desirable characteristics:

- *Scalable:* We can mine specifications from traces that are millions of cycles long, with thousands of signals, all within two minutes.
- *Effective for diagnosis:* In fault injection experiments, our mined specifications correctly localized the fault to within module boundaries.
- *Relevant:* The learned specifications, after the summarization procedure, are sufficiently high-level so as to be useful to designers.

We use ModelSim to simulate the designs and to record the trace as a VCD dump file (which already extracts delta events). The experiments are run on a netbook with an Intel Atom 1.60 GHz processor and 1.0 GB of RAM.

Benchmarks: We used 4 benchmarks in our experiments. The first is the MIPS core in the extensible MIPS processor developed by Pittman et al [23]. eMIPS is a dynamically extensible processor architecture based on the MIPS R4000 instruction set. The design has 278 modules and contains more than 20000 signals. Router is a chip-multiprocessor router designed by Peh [22]. We use a simplified 2-port version with 14 modules here. I2C and CAN are the I2C interface and CAN interface designs obtained from Opencores [1].

Experiments: In our experiments, we only track latches with width less than 5. This is a simple heuristic to quickly prune away the various data paths, because we do not start with a manual event definition.

Scalability: The first experiment is meant to evaluate both the efficiency of our specification mining algorithm and the usefulness of the specification compaction procedure. We simulated each of the benchmarks on the default testbench supplied with it to generate one very long trace for that benchmark. Then we applied our mining tool to the resulting trace. (Applying our tool to multiple traces will yield similar results.)

Table 1 gives the statistics for our performance experiments.

	$\lvert\tau\rvert$	$\lvert\overline{\tau_\Delta}\rvert$	$\tilde{n}_m$	$\lvert S\rvert$	$\lvert S_m\rvert$	$R_t(s)$
eMIPS	5.0×10^6	5408	108	2079	1028	51
Router	2.3×10^5	12420	28	120	74	13
I2C	1.6×10^6	20904	33	389	308	9
CAN	2.6×10^7	36100	175	3272	1356	71

Table 1: Performance Results

$|\tau|$ is the size of the original trace. $|\overline{\tau}_\Delta|$ and $\tilde{n}_m$ are the average length of delta traces and the maximum number of delta events per module respectively. $|S|$ is the total number of specifications mined before any compaction is performed. $|S_m|$ is the total number of specifications resulting from applying the parallel merging and chaining procedures. R_t is the overall runtime of the tool for each benchmark in seconds.

As we can observe, the transformation to a delta trace reduces the length of the trace by about 1000X. Moreover, the modularization then reduces the number of delta events to be processed by our algorithm to at most 175 (and only 9-16 on average). The original number of specifications we generate is in the few thousands, but merging can reduce this number by 2X. The run-times are very small – less than 2 minutes for all benchmarks.

Relevance: Figure 7 shows part of an example specification (after all the composition procedures have been performed) mined in the "vcstate" module for the CMP router. This module contains a state machine that controls the handshake with the arbiter and the decision to move the flits in the buffer to the corresponding output channels. Figure 6 shows the actual behavior of the state machine. The number after the colon corresponds to the value of the signal. Together they constitute an event (in this case a delta-event). State_status corresponds to the state in the input controller state machine.

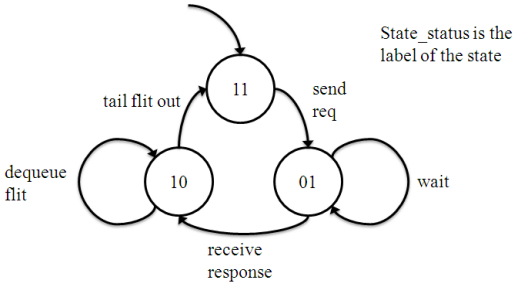

Figure 6: State Machine in the "vcstate" Module

In the mined patterns, we can see that for example, State_swreq always stays at $2b10$ (requesting output channel 1) until State_status moves to $2b10$. In fact, it stays high until State_queuelen (which registers the number of flits in the buffer) goes to 0. If State_queuelen goes to 0, then State_status eventually goes back to its initial state $2b11$. Due to the particular configuration of this router and the test bench, with a buffer size of 4, a data packet consisting of a head, a body and a tail, and the frequency of packet injection at about 0.23, the specification captures the behavior correctly – a data packet (3 flits) comes in and fills up the buffer; the head flit triggers a request and before another packet comes to the same input channel, it manages to secure an output channel and the entire packet gets dequeued through that channel. Although the quality of the specification depends heavily on the quality of the simulation trace, the mined specifications are still useful since it can indicate the parts of behaviors that the current test-bench has covered, allowing future endeavors to be directed to the uncovered behaviors quickly, e.g. test with different router configurations and traffic patterns.

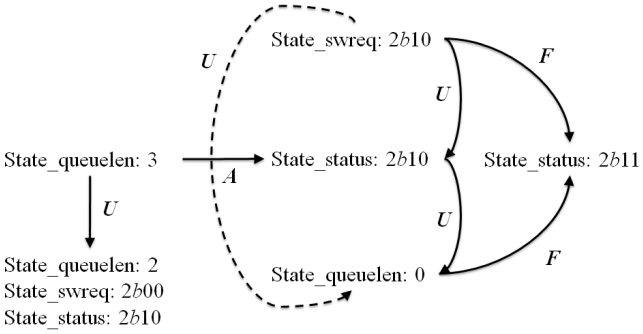

Figure 7: Example Mined Specification from the CMP Router

Fault Diagnosis: In the second set of experiments, we syntactically injected faults into the designs and then used our mining-based diagnosis approach to localize the fault. Readers can refer to [20] for a full set of diagnosis results. We discuss the experiments on the eMIPS processor and the CMP router here in detail.

eMIPS Processor. The faults we injected included single-bit errors, multiple-bit errors, and erroneous state transitions. In the first case, we inverted the "dne_r" signal in the BlockRAM Controller from $1b0$ to $1b1$. In the second, we changed the "we_r" signal in the BlockRAM Interface module from $4b0$ to $4b6$. In the last case, we changed the "rdstate" signal conditionally in the Register_File_Read module from $2b00$ to $2b10$ when it is in state $2b01$. This represents an erroneous transition in a state machine. We produced from each case an error trace of 1 million cycles, which is also approximately 1 million cycles before the error failed some system-level end-to-end specification. *In all three fault injection experiments, our diagnosis technique ranked the faulty module as a top candidate among the 278 modules.* However, on average 6 other modules are also ranked as top candidates. This is due to the fact that some signals in these modules are combinational output of

Type of Fault	N_f	Cov_A	$Local_t$	$Local_m$
stuck-at	5	100%	–	100%
erroneous transition	3	100%	–	100%
erroneous assignment	7	100%	–	57%
transient	16	100%	81%	56%

Table 2: Diagnosis Results on the CMP Router

the error, and these signals in turn violated some local properties mined in their modules. While it is possible to overcome this by tracking only the registers, the tradeoff is that since we track less signals, we will lose some behavioral coverage.

CMP Router. With the same configuration of the Router used in the specification mining experiments, we use a testbench with random packet generation at the input but with a fixed traffic pattern. The same testbench is used to generate both the correct trace and the faulty trace. This time, our mined specifications only covered signals that are registers. We inject different types of faults into the router, including stuck-at faults on wires and registers, erroneous transition in state machines, erroneous assignment (syntactical) for wires and registers, and single transient errors in registers. Table 2 shows the results of our diagnosis. N_f is the number of fault injection experiments that were performed for each type of fault. Cov_A is the percentage of times that the assertions mined from the correct trace is falsified by the error trace — this is a form of assertion coverage. $Local_t$ (for transient errors) is the percentage of times that error was trapped by some distinguishing pattern within after 15 cycles of the transient error – this measures localization in time. Finally, $Local_m$ is the percentage of times that our diagnosis returned the correct module where the error occurred – this measures localization in space.

As we observe, the technique has perfect assertion coverage, localizes transient errors well in time, and more than 50% of the time, the top-ranked assertion gives perfect localization in space also. For example, the assertion "$\mathbf{G}$ ((state_queuelen = 2) $\rightarrow \mathbf{X}\,\mathbf{F}$ (state_swreq = 0))" is the distinguishing pattern that successfully trapped a transient error in the "state_status[0]" signal and returned the correct module localization, even though the distinguishing pattern does not contain the error signal.

7. CONCLUSION

We have proposed a scalable specification-mining tool that is suitable for general digital circuits. In addition, we have shown that our mined specifications are effective for fault diagnosis. Evaluation shows that (a) the mining algorithm is practical, requiring only minutes of computation even for a very large-scale example, (b) for human benefit, the mined specifications can be automatically compacted by a significant factor, (c) the diagnostic use is effective in pinpointing the error location to the correct module when programming mistakes and hardware faults are applied to the benchmarks.

An inherent limitation of dynamic specification mining is that the quality of the specification mined is only as good as the set of traces. In the case of digital circuits, we can leverage techniques in coverage-directed testing to simulate many behaviors as fast as possible. An alternative to using coverage tools is to improve the mined specifications online, as the circuit runs in a testing or production mode on real workloads. For example, in a testing mode one can prototype the circuit on a piece of reconfigurable logic and iteratively generate online assertion checkers for specifications mined from each trace. Online monitoring and refinement of properties will be interesting directions for future work.

Acknowledgement. We would like to thank the anonymous reviewers for their comments. The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This work was also supported in part by a Hellman Family Faculty Fund Award, and by an Alfred P. Sloan Research Fellowship.

8. REFERENCES

[1] Opencores benchmarks. www.opencores.org.
[2] Alur, R. et al. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109, 2005.
[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.
[4] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Comput.*, 45(9):993–1002, 1996.
[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.
[6] M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the international conference on Reliable software*, pages 165–171, 1975.
[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE'07*, pages 5–14, 2007.
[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2000.
[9] L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. pages 78–88, 1992.
[10] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnoses and systems. *Artif. Intell.*, 56(2-3):197–222, 1992.
[11] Engler, D. et al. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP*, pages 57–72, 2001.
[12] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA*, pages 24–27, 2003.
[13] Ernst, M. et al. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
[14] G. Fey and R. Drechsler. Improving simulation-based verification by means of formal methods. In *ASP-DAC*, pages 640–643, 2004.
[15] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. In *FSE*, pages 339–349, 2008.
[16] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *ICSE*, pages 51–60, 2008.
[17] E. M. Gold. Complexity of automatic identification from given data. 37:302–320, 1978.
[18] Hangal, S. et al. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *DAC*, pages 775–778, 2005.
[19] B. Isaksen and V. Bertacco. Verification through the principle of least astonishment. In *ICCAD*, pages 860–867, 2006.
[20] W. Li, A. Forin, and S. A. Seshia. Scalable specification mining for verification and diagnosis. Technical report, EECS Department, UC Berkeley, April 2010.
[21] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. 1992.
[22] L.-S. Peh. *Flow control and micro-architectural mechanisms for extending the performance of interconnection networks*. PhD thesis, 2001.
[23] R. Pittman, N. Lynch, and A. Forin. emips, a dynamical extensible processor. Technical Report MSR-TR-2006-143, Microsoft Research, 2006.
[24] Rogin, F., et al. Automatic generation of complex properties for hardware designs. In *DATE*, pages 545–548, 2008.
[25] S. Sankaranarayanan, F. Ivanči, and A. Gupta. Mining library specifications using inductive logic programming. In *ICSE*, pages 131–140, 2008.
[26] B. Wegbreit. The synthesis of loop predicates. *Commun. ACM*, 17(2):102–113, 1974.
[27] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *TACAS*, pages 461–476, 2005.
[28] Yang, J. et al. Perracotta: mining temporal api rules from imperfect traces. In *ICSE*, pages 282–291, 2006.