Specification Mining for Digital Circuits with Applications on Verification and Diagnosis

Wenchao Li, Alessandro Forin
Microsoft Research

August 21, 2009

Technical Report
MSR-TR-2009-114

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Specification Mining for Digital Circuits with Applications on Verification and Diagnosis

Wenchao Li, Alessandro Forin
Microsoft Research

Abstract

Software and hardware systems are often built without detailed documentation. The correctness of these systems can only be verified as well as the specifications are written. The lack of sufficient specifications often leads to misses of critical bugs, design re-spins, and time-to-market slips. In this paper, we address this problem by mining specification dynamically from simulation traces. 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 error-labeled traces are used to automatically pinpoint the sources of error. In this work, we focus on traces from digital circuits, but any ordered trace of events is amenable to this analysis. We demonstrate the effectiveness of our approach on industrial-size examples.

1 Introduction

Formal specifications can succinctly capture a system's behaviors. One can then leverage verification techniques such as model checking [23] to ensure the correctness of the system. However, as hardware designs rapidly grow in complexity, this is becoming a very expensive proposition. For instance, Intel reported that in 2005 the manpower assigned to design was a third of that assigned to validation [17]. Furthermore, the difficulty of formalizing a complete set of formal properties has significantly hindered the wide-spread adoption of formal and semi-formal techniques. Hence, in order to render automated verification techniques effective, there is a constant need to add new properties or modify existing ones. Recently, Kupferman et al. [2] developed a theory of mutations that enables the strengthening of *weak* specifications. However, the goal of automatically synthesizing a complete set of proper specifications remains elusive even for the state-of-the-art.

Specification mining is a promising alternative. It is the process of extracting specifications, either statically from the description of a system, or dynamically from its executions. The specifications mined in turn allow us to better understand the system, verify its correctness, and manage possible evolutional changes. In this work, we dynamically 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 check 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 trace and optionally a user-defined event definition as input, and generates a set of behavioral patterns that are always true in the trace as output. A trace is a sequence of events ordered by the 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 the most interesting properties.

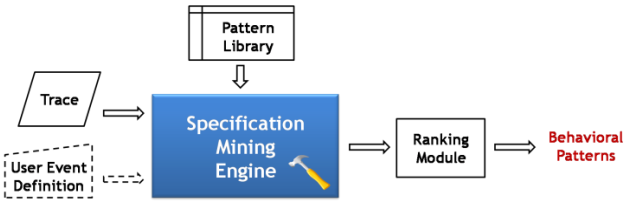

Figure 1: Specification Mining for Verification

Specification mining not only closes the loop in verification environments such as coverage-driven simulation or formal verification, it can also provide useful information for diagnosis. We propose a pattern-mining based trace diagnosis framework that can be used to simultaneously understand the error and locate it. Figure 2 show how our specification mining engine is used as a subroutine to determine the distinguishing patterns between a normal trace and an error trace. The distinguishing patterns are the patterns that exist in one trace but not in the other, or patterns

that exist in both traces but with conflicting timing bounds. After finding these patterns, we apply a localization procedure to pinpoint the source of error.

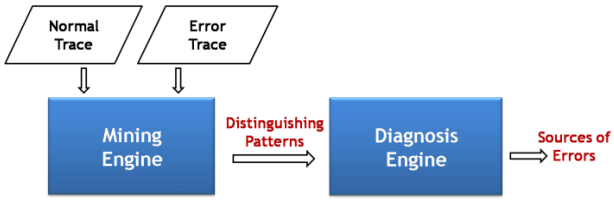

Figure 2: Specification Mining for Diagnosis

In this paper, we make these key contributions:

- *A novel dynamic specification mining technique especially designed for general digital circuits. Our tool efficiently mines non-trivial specifications and is highly scalable;*
- *A novel trace-diagnosis technique based on specification mining that achieves good accuracy for very large circuits.*

The paper is organized as follows. Section 2 gives an extensive survey on related literature of specification mining. Section 3 illustrates the main concepts and formally defines the problem. Section 4 describes the algorithms for mining specification in digital circuits. Section 5 describes a pattern-mining based technique for diagnosis. Section 6 presents experimental results. Section 7 concludes.

2 Related Work

Many techniques have been proposed to automatically reverse-engineer specifications from a system [6][7][8][10][12][13][14]. The study of automatically generating specifications goes back as early as [3][4]. These specifications can be simple predicates in programs, or temporal specifications which specify the ordering of events (such as function calls), or general programming rules [5]. The specifications generated can then be used to formally verify a program's correctness, to assist in debug [9], or to detect malicious behaviors [11].

Specifications are typically learned *dynamically* from an execution trace (or a set of traces). A trace can be a sequence of function calls or a sequence of memory transactions on a bus. Daikon [13] is one of the most well-known tools that mine *value-based* invariants. We focus on mining *temporal properties* in this work. Most existing mining tools produce temporal properties in the form of automata. In addition, dynamic analysis mainly targets strongly-observed sub-behaviors. Automata-based techniques generally fall into two categories. The first one learns a single complex specification (usually as a finite automaton) over a specific alphabet [8][25], and then extracts scenarios to one's flavor. In [8], Ammons et al. first produces a probabilistic automaton that accepts the trace and then extracts from it likely properties. However, extracting a single finite state machine from trace has been proved to be NP-hard [26]. To achieve better scalability, one can first learn multiple small specifications and then post-process them to form more complex state machines. Engler et al. [27] first introduce the idea of mining simple alternating patterns. Several subsequent efforts [9][7][10][12] built upon this work. For example, Javert [12] 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. Javert is similar in spirit to our work. We focus on patterns that are meaningful for digital circuits and provide a merging procedure that can compose events in time.

Dynamic techniques have also been directed to cope with potentially imperfect traces. Various measures have been proposed to measure a property's statistical significance. *Support* and *confidence* are two common measures used in the data mining community. PR-Miner [5] makes use of frequent itemset mining to find highly correlated function calls. These two measures sometimes result in desirable monotonicity and non-redundancy properties that can give a potentially combinatorial speed-up and reduction in the number of properties mined [14]. One advantage of the template-based approach over the data mining approach is that the template-based approach is particularly amenable to online analysis whereas the data mining approach usually requires indexing the events at the beginning. In our work, we do not explicitly handle imperfectness in traces other than allowing them to be open-ended. However, it is straightforward to extend our technique to compute the statistical significance for each property by keeping track of the number of occurrences of the constituent events.

In general, the quality of the trace directly impacts the quality of the specifications mined. Since dynamic analysis can only infer specifications from existing traces, coverage remains a fundamental limitation. Recently, Csallner et al. [28] leverage dynamic symbolic execution to improve the quality of their inferred invariants.

Specifications can also be learned by reasoning about the program *statically*. For example, Alur et al. [1] proposes the use of predicate abstraction together with automata learning to automatically synthesize interface specifications for Java classes. Whaley et al. [25] proposes the use of type information in object-oriented programs to produce an automaton that

encodes legal call sequences of operations of that type. Shoham et al. [29] mines Java client API rules using an automata-based abstraction that represents unbounded event sequences. Static and dynamic analyses complement each other. We refer the readers to [22] for a detailed comparison of the two techniques.

Various circuit-specific mining techniques have been proposed by taking into account some special hardware properties. In [32], the authors propose a tool that mines simple likely invariants such as one-hot encodings or fixed-delay pairs. Fey and Drechsler [31] 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. Isaksen and Bertacco [24] propose the use of inferred boundary labels to generate transaction diagrams from a trace. Their methodology is particularly suitable for analyzing protocols. We focus on the mining of temporal properties for large general circuits.

3 Concepts and Definitions

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

We denote S as 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). 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. In this work, we do not define events as assignments of signals across cycles. 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 consider finite traces in this work.

Definition 3.1 (Projection) The projection π of a trace τ over an alphabet Σ, $\pi_\Sigma(\tau)$ is defined as τ with all events not in Σ deleted.

Definition 3.2 (Specification Pattern) [7] 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 $\pi_\Sigma(\tau) \in \mathcal{L}(A)$ i.e. the projection of the trace on the pattern alphabet is in the language of the pattern automaton.

Definition 3.3 (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 ***aRb***.

Our specification pattern miner takes a trace, a set of pattern templates and optionally an event definition as input, and produces all pattern instances that are satisfied over the trace as output. In this paper, we focus on the more challenging problem of mining of specifications without an explicit event definition. 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 piece of software is that a digital circuit can be thought of as a massive concurrent process, in which thousands of events occur at every clock cycle. In traditional software specification mining problems the size of the problem is relatively small; 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 a while until a response is received. We introduce the notion of delta event, which is defined formally as the following.

Definition 3.4 (Delta Event) A delta event, Δe, is an event such that at least one of its constituent signals changes value from the previous valuation, i.e. $\Delta e \triangleq e = \{\vec{s}, \vec{v}, t\}$ s.t. $\exists\, v_{s,t} \in \vec{v} \wedge v_{s,t} \neq v_{s,t-1}$.

Given these definitions, our mined patterns can be expressed succinctly in LTL [21] or as regular expressions. For example, we can express that every request must be eventually followed by a grant as *"G (request → F grant)"* in LTL, where the operator *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 set of binary patterns that our tool mines is illustrated below.

Alternating (*A*) We restrict ourselves to only delta events. An *alternating* pattern between two delta events Δa and Δb can be then succinctly described by the regular expression $(\Delta a\ \Delta b)^*$. Note that this does *not* mean Δb follows Δa immediately in the next cycle. We denote this pattern as ***aAb***. Figure 3 shows the corresponding finite automaton.

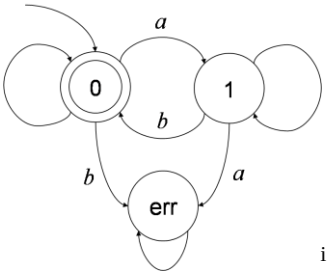

Figure 3: Finite Automaton for the Alternating Pattern

Until (*U*) The *until* pattern can be used to describe behaviors such as "the request line stays high until the corresponding response is received." Figure 4 shows a trace where this pattern is satisfied. Formally, the LTL formula is *"G (Δa → X (a U Δb))."*[ii] We denote the pattern as ***aUb***.

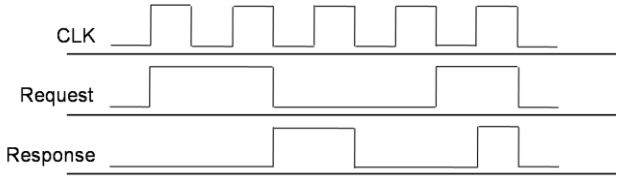

Figure 4: Request stays high until a Response is received.

Next (*X*) The *next* pattern corresponds to the LTL formula *"G (Δa → X Δb)."* We denote it as ***aXb***. Note that ***aXb*** implies ***aUb***. One can also generalize this pattern to fixed-delay pairs.

Eventual (*F*) The *eventual* pattern can be described by the LTL formula *"G (Δa → X F Δb)."* We denote this as ***aFb***. Note that ***aAb***, ***aUb*** and ***aXb*** all imply ***aFb***. We output ***aXb*** from ***aFb*** based on the timing bounds.

Timing information is 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 compute the relevant timing bounds on the fly.

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.

[i] Unlabeled transitions basically include the rest of the events in the alphabet such that the automaton is also deterministic.

[ii] The ***U*** operator is the strong until, which means Δ*b* has to be eventually true for the formula to be true.

4 Mining and Summarization

We exploit the highly modular nature of hardware design to keep the problem tractable. We partition the trace by module into many disjoint sub-traces and analyze them separately. This allows us to scale our technique easily to industrial-size designs. Empirical results also suggest that partitioning the trace does not decrease the quality of the specifications mined even though we have a smaller alphabet to mine from for each trace. 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. We allow only a slight degree of imperfection when handling open-ended traces. This allows us to adapt the analysis to any snapshot of a trace or to an ongoing execution of the system (i.e. online). Informally, this means that a pattern may have its last occurrence non-accepting as long as it is not in an error state. The specification mined can be immediately examined by the user or synthesized as online monitors for other executions. While this greatly reduces false-positives, the tradeoff is that we may lose the potential to debug the current trace since a frequent but non-universal pattern may indicate an erroneous behavior**.**

Gabel and Su [7] formalize the *pattern-based* specification mining problem and show that its general form 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 exponential blow-up. The approach taken in Perracotta [10] requires $O(n^k)$ 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 [7] propose the use of Binary Decision Diagram (BDD) to improve the tractability of the problem. However, while they show significant speedup using the symbolic technique, the practical input alphabet size is still 3. In addition, the performance of BDD-based techniques depends heavily on the good variable ordering, and finding the optimum variable ordering is again NP-hard [15].

Our algorithm mines binary patterns with timing bounds as discussed in Section 3. We adopt the Perracotta approach but extend it to handle traces with multiple events at the same cycle and to mine richer

binary patterns. While we have to allocate a quadratic space and perform linear-time operation per slice, we can use bit-strings and bit-masking to improve the algorithm's efficiency. The algorithm is briefly outlined below. All the events are *delta events* here, which we write just as *e*. The algorithm for mining ***aUb*** is shown below, where *a* and *b* are events that contain a single binary variable.

Algorithm 1: Mine ***aUb***

```
function UNTILMINE (η: slice, t: time)
  foreach (event e in η) // update row
    for (i = 0; i < alphabet_size; i++)
      if (table[ind_e, i] == 0) table[ind_e, i] = 1;
      else if (table[inde, i] == 2) tabe[inde, i] = -1;

      end if
    end for
  end foreach

  foreach (event e in η) // update column
    for (j = 0; j < alphabet_size; j++)
      if (alphabet[j] == ¬e[iii])
        for (k = 0; k < alphabet_size; k++)
          if (table[j, k] == 1) table[j, ind_e] = 2;
          end if
        end for
      else if ((table[j, ind_e] == 1) |
            (table[j, ind_e] == 2))
        table[j, ind_e] = 0;
      end if
    end for
  end foreach
end function
```

The algorithm basically updates the associated pattern automaton in each entry of the pattern table, for each slice of the trace, in an online fashion. The number 0 and -1 correspond to the accepting state[iv] and the error state of the automaton respectively. The algorithm for mining ***aAb*** is similar, provided we avoid using two events from the same slice to update the same pattern. The final algorithm requires $O(n^2)$ space and $O(mnl)$ time, where m is the maximum number of events per slice.

4.2 Specification Summarization

Specification summarization is essential to keep the set of properties to a manageable size. It eliminates redundant specifications and helps specification comprehension. We perform three summarization procedures – event merging, pattern chaining and graph composition.

We first merge the patterns in parallel by matching their time of occurrence. For example, if both *(a b)** and *(a c)** are true, and *b* and *c* always occur at the same time, we can merge them to form *(a b∧c)**. We keep the indices of occurrences for the events in each pattern so that a recursive procedure suffices to merge such patterns maximally by scanning their occurrences as time increases (by moving a pointer pointing at the time of occurrence). The algorithm is outlined below.

Algorithm 2: Merge ***aRb***

```
function MERGE (P: list of pattern instances)
  if (|P|==1)
    return P;
  else if (all instances at last occurrence in P)
    return P;
  else if (some instances at last occurrence in P)
    return append(ended instances in P,
                  MERGE (remaining instances));
  else
    return append(MERGE (instances with earliest
current occurrence in P),
                  MERGE (remaining instances));
  end if
end function
```

This parallel merging procedure is particular useful when no event definition is 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 improve the quality of specifications while reduces their number.

After we merge the patterns in parallel, we repeatedly apply a set of inference rules to the results to obtain even more complex patterns. Some inference rules for chaining binary relational patterns are

[iii] This means the binary variable takes the opposite value.

[iv] 1 is also accepting if we allow open-ended trace.

illustrated below. It is straightforward to generalize them for further composition of similar patterns.

Alternating Pattern Chaining Rule [12]:

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

Next Pattern Chaining Rule:

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

Eventual-Until Pattern Chaining Rule:

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

We further graphically compose the resulting patterns to create an edge from *a* to *b* for patterns ***aRb***. Every disjoint sub-graph represents a complex behavior amongst its constituent events.

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. Alternatively, we can rank patterns according to their frequencies, timing distributions or time of first occurrence.

5 Trace Diagnosis

Verification has been a difficult task and diagnosis is even harder. Dramatic increase in design complexity in recent years is making these activities a very expensive proposition. Such validation process can consume 35% of chip development time on average [30]. Bugs are even more difficult to diagnose in the post-silicon environment due to limited observability, reproducibility and dependence on physical parameters.

In this paper, we also consider the problem of debugging potential multiple errors 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.

A number of diagnosis approaches have been proposed in the literature. As observed by Console et al [16], 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 [18]. 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 of the system fails (a fault model), it requires the complete specification of the correct system.

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 require the RTL description of the system to be known, which makes it especially appealing for any gray-box systems. In addition, we do not need to 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. This means that there exists a mechanism to label a trace with respect to some correctness criteria. Typically, such mechanism relies on checking some end-to-end behaviors or observing whether an exception is raised.

Consistency is defined with respect to the specifications mined from the correct trace. Specifically, if a pattern is only observed in the error trace but not in the correct trace, or if the pattern in the error trace violates the timing bounds in the correct trace, we record it as a potential suspect for error. An error can propagate to other modules and in turn cause more erroneous behaviors later. In light of this, we rank the diagnosis candidates by the time of its first violation. The module which the top candidate belongs to gives the localization result. Since the pattern itself describes a specific erroneous behavior, our approach also produces useful insights into what the error actually is.

We present a case study on a microprocessor in Section 6 to demonstrate the effectiveness of this diagnostic approach. It also serves as a way to evaluate the completeness of the specifications mined. Intuitively, reasonable complete specifications should allow us to catch many different errors.

6 Experimental Results

We experiment with the MIPS core in the extensible MIPS processor developed by Pittman et al [19]. eMIPS is a dynamically extensible processor architecture based on the MIPS R4000 instruction set [20]. The core contains 278 modules and over 20000 signals. We only track signals 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. We use ModelSim to simulate the eMIPS core and the trace is recorded as a VCD dump file. The input to the simulation system is a (binary) C program that runs on the eMIPS core, transmitting packets over an on-chip multiprocessor router. The experiments are run on a netbook with an Intel Atom 1.60 GHz processor and 1.0 GB of RAM.

The first experiment is meant to evaluate both the efficiency of our specification mining algorithm and the usefulness of the specification compaction procedure. The length of the trace is 5 million cycles. We ended up tracking 5945 delta events from the signals. It took a total of 11 minutes and 47 seconds to mine all the specifications, including reading the dump file from disk and merging the specifications afterwards. Figure 5 shows that the specification summarization procedure significantly compacts specifications. The parallel merging procedure reduces the number of specifications by more than 5 times and the chaining procedure further reduces the number of specifications.

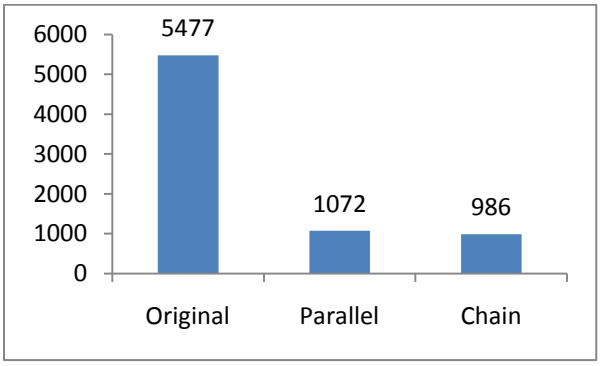

Figure 5: Number of Specifications Mined

In the second experiment, we syntactically injected faults into the MIPS core and then used our mining-based diagnosis approach to localize the fault. We performed three case studies. In the first one, we inverted the *dne_r* signal in the Block_RAM Controller from 1'b0 to 1'b1, as an example of a very common programming mistake that is often hard to locate. In the second, we changed the *we_r* signal in the Block_RAM Interface module from 4'b0 to 4'b6. This is an example of a multi-bit error. In the last, we changed the *rdstate* signal conditionally in the Register_File_Read module from 2'b00 to 2'b10 when it is in state 2'b01. This represents an erroneous state machine transition. We produced from each case an error trace of 1 million cycles, which is also approximately 1 million cycles before the error is first observed. In all three case studies, 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 the error, and these signals in turn violate 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. The time taken to produce the diagnosis candidate for each case was under 20 minutes on the previously described machine. While these results are indicative of the effectiveness of our diagnosis framework, we would like to experiment more with localizing transient faults while observing only the interface signals between modules, which are often the only observables in a post-silicon debug environment.

7 Discussion and Conclusion

The spirit of dynamic specification mining is to quickly generalize recurring patterns in a trace as universal specifications for the system. However, an inherent limitation with this approach is that the quality of the specification mined is only as good as the trace. A large recall rate can seriously undermine the effectiveness of this technique if a lot of the specifications mined turn out to be incorrect. One way to mitigate this problem is to prune out erroneous specifications by taking the intersection of specifications (from a common alphabet) mined from different traces. Although we are still interested in specifications in the disjoint alphabets, the problem is significantly different from the one in software where each path usually entails a unique behavior (and the goal there is to exercise all unexplored branches as fast as possible [34]). In the case of digital circuits, we can leverage works in coverage-directed testing to simulate many behaviors as fast as possible. An alternative is to prototype the circuit on a piece of reconfigurable logic (often a FPGA) and iteratively generate online assertion checkers for specifications mined from each trace. Lu and Forin [33] present the PSL-to-Verilog compiler which efficiently generates hardware monitors from PSL specifications based on logic rewriting. An attractive property of these monitors is that they are in fact software instrumentations realized

in hardware, and therefore do not alter a program's temporal behavior in any way. The monitors execute concurrently with the program and validate the properties that they are synthesized from with zero execution overhead. At each iteration, an invalidated monitor either indicates the existence of a bug or an erroneous behavior.

In conclusion, we have proposed a scalable and extensible specification-mining framework that is suitable for general digital circuits. In addition, we have shown that our specification mining procedure is effective as a subroutine for trace 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 about a factor of 5x, (c) the diagnostic use is effective in pinpointing the error location to the correct module when each of three popular programming mistakes is applied to a very large-scale example. In the future, we would like to experiment our diagnosis technique with different types of faults.

References

[1] R. Alur, P. Černy, P. Madhusudan and W. Nam. Synthesis of Interface Specifications for Java Classes. In *Proc. 32nd ACM POPL*, pages 98-109, 2005.

[2] O. Kupferman, W. Li, S. A. Seshia. A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance. In *Proc. 8th FMCAD*, pages 1-9, 2008.

[3] M. Caplain. Finding invariant assertions for proving programs. In *Int. Conf. on Reliable software*, 1975.

[4] B. Wegbreit. *The synthesis of loop predicates*. Communications of the ACM, 17(2), 1974.

[5] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In Proc. ESEC/FSE, pages 306-315, 2005.

[6] Sankaranarayanan, S., Ivanči, F., and Gupta, A. Mining library specifications using inductive logic programming. In *Proc. of ICSE '08*.

[7] Gabel, M. and Su, Z. Symbolic mining of temporal specifications. In *Proc. of ICSE '08*.

[8] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. In *Proc. of POPL'02*.

[9] W. Weimer and G. C. Necula. Mining Temporal Specifications for Error Detection. In *Proc. of TACAS'05*.

[10] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. Perracotta: mining temporal API rules from imperfect traces. In *Proc. of ICSE '06*.

[11] Christodorescu, M., Jha, S., and Kruegel, C. Mining specifications of malicious behavior. In *ESEC-FSE'07*.

[12] Gabel, M. and Su, Z. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proc. of SIGSOFT '08/FSE-16*.

[13] Ernst, M. D. et al. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 2007.

[14] Lo, D., Khoo, S., and Liu, C. Mining past-time temporal rules from execution traces. In *Proc. of WODA'08*.

[15] Beate Bollig, Ingo Wegener. Improving the Variable Ordering of OBDDs is NP-Complete. *IEEE Transactions on Computers*, 1996.

[16] L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Comput. Intell.*, 1991.

[17] P. Patra. On the cusp of a validation wall. *IEEE Des. Test*, 24(2):193–196, 2007.

[18] J. de Kleer, A. K. Mackworth, and R. Reiter. Characterizing diagnosis and systems. *Artificial Intelligence*, 56, 1991.

[19] Pittman, R. N., Lynch, N. L., Forin, A. eMIPS, A Dynamically Extensible Processor. *MSR-TR-2006-143*, Microsoft Research, WA, October 2006

[20] J Heinrich. MIPS 4000 Microprocessor User's Manual. 1994.

[21] Manna, Z. and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag 1992.

[22] M. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Work. On Dynamic Analysis*, 2003.

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

[24] B. Isaksen, V. Bertacco. Verification through the Principle of Least Astonishment. In *Proc. of ICCAD'06*.

[25] Whaley, J., Martin, M. C., and Lam, M. S. Automatic extraction of object-oriented component interfaces. In *Proc. of ISSTA '02*.

[26] E. Gold. Complexity of automatic identification from given data. *Information and Control*, 37, 302-320, 1978.

[27] D. Engler, et al. Bugs as Deviant Behavior: a General Approach to Inferring Errors in System Code. In SOSP'01.

[28] Csallner, C., Tillmann, N., and Smaragdakis, Y. DySy: dynamic symbolic execution for invariant inference. In *Proc. of ICSE '08*.

[29] S. Shoham, E. Yahav, S. Fink, M. Pistoia. Static specification mining using automata-based abstractions. In *Proc. of ISSTA'07*.

[30] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller. A reconfigurable design-for-debug infrastructure for socs. In *Proc. of DAC'06.*

[31] G. Fey and R. Drechsler. Improving simulation-based verification by means of formal methods. In *Proc. of ASPDAC'04*.

[32] S. Hangla, N. Chandra, S. Narayanan, S. Chakravorty. Iodine: A tool to automatically infer dynamic invariants for hardware designs. In *Proc. of DAC'05*.

[33] Lu, H., Forin, A. The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs. *MSR-TR-2007-99*, Microsoft Research, WA, August 2007.

[34] Nikolai Tillmann, Jonathan de Halleux. In *Proc. of TAP 2008, the 2nd International Conference on Tests and Proofs*, LNCS, vol. 4966, pages 134-153, April 2008.