IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs

Sudheendra Hangal
Sun Microsystems, Bangalore, India
sudheendra.hangal@sun.com

Naveen Chandra
Sun Microsystems, Bangalore, India
naveen.chandra@sun.com

Sridhar Narayanan
P. A. Semi Inc, Santa Clara, CA
sridharn@pasemi.com

Sandeep Chakravorty
Sun Microsystems, Bangalore, India
sandeep.chakravorty@sun.com

ABSTRACT

We describe IODINE, a tool to automatically extract likely design properties using dynamic analysis. A practical bottleneck in the formal verification of hardware designs is the need to manually specify design-specific properties. IODINE presents a way to automatically extract properties such as state machine protocols, request-acknowledge pairs, and mutual exclusion between signals from design simulations. We show that dynamic invariant detection for hardware designs can infer relevant and accurate properties.

Categories and Subject Descriptors: B.5.2 **Hardware** Register-Transfer-Level Implementation *Verification*
F.3.1 **Theory of Computation** Logics and Meanings of Programs *Invariants*

General Terms: Verification, Algorithms

Keywords: Dynamic Invariants, Dynamic Analysis, Formal Specification

1. INTRODUCTION

Traditional verification techniques center around extensive simulation-based testing, where a design is exercised heavily using directed or random test-cases, and the behavior of the design is checked for correctness.

In contrast, using formal verification (or formal property checking) techniques, designers specify properties of their design which must be valid for all legal inputs. Commercial formal verification tools such as 0-IN^{TM} Confirm, Magellan^{TM}, JasperGold^{TM}, or Verix^{TM} can be used to prove that the property either holds for the given design, or to produce a legal input sequence that violates the property.

Formal property checking is often applied to design blocks which have a relatively well-defined high-level specification, such as blocks involving bus interfaces or floating point arithmetic. However, in practice, many important design blocks do not have robust specifications or documentation, and identifying design properties in such cases is a daunting task.

2. HARDWARE DYNAMIC INVARIANTS

We describe a tool called IODINE (Implementation of Dynamic Invariant Extraction) to automatically extract likely properties based on design simulations. Our technique is inspired by prior work on inferring and checking dynamic invariants for software programs [2][3], and the difficulties we have faced in identifying properties for formal verification on real-life designs. Our approach extracts dynamic invariants by hypothesizing a set of invariants across one or more variables in the design and analyzing the behavior of the design over a set of inputs. As potential invariants are falsified during program runs, they are removed from the candidate set, leaving only invariants which are true for the given input set.

Dynamic invariants have the well-known limitation that they may be unsound; a false invariant is nevertheless useful, because it points to functionality not covered by the test-suite. In particular, test suites for hardware designs are expected to be nearly complete. Therefore, dynamic invariants for hardware designs tend to be fairly accurate, and any test coverage deficiency pointed out by a false invariant is particularly valuable.

An important feature of IODINE analysis is that it is *pervasive*, which means that it identifies all invariants within its space of candidate invariants, unlike manual specification which is subject to the author's biases, blind spots and time pressures. A high density of design properties is also helpful for generating shorter counter-example traces esp. for formal verification tools based on bounded model checking [1].

The following sections describe our key contributions; (i) A general purpose framework for dynamic analysis of hardware designs (ii) A set of analyzers which extract useful dynamic invariants for hardware designs (iii) Our experiences on extracting dynamic invariants for one block of a dual-core SPARC^{TM} processor design.

3. IODINE FRAMEWORK

The IODINE dynamic analysis framework (Figure 1) consists of a simulation dump reader, a set of analyzers to process the simulation data, a GUI to view invariants and a comparator to compare two sets of invariants.

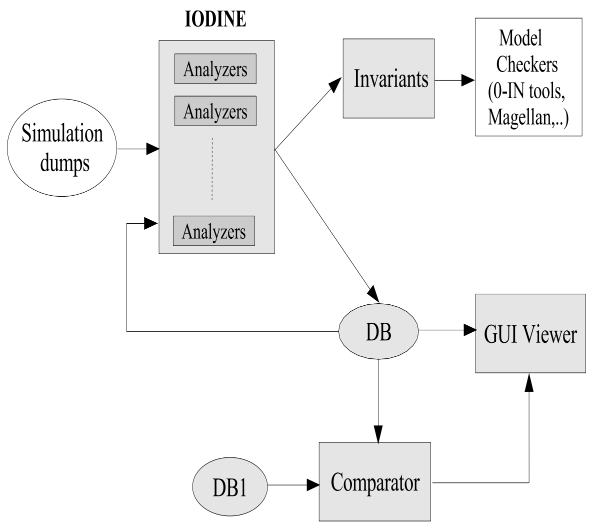

Figure 1: IODINE Framework

Initially, the design is simulated using an extensive test-suite consisting of manually written or randomly generated test-cases. It is assumed that the design passes these tests according to the checks employed in the existing design verification environment; this ensures that invariants are inferred only over correct behavior of the design. Information generated during the simulations is recorded in the form of simulation dumps in industry-standard formats such as VCD or FSDB files. This keeps IODINE agnostic to details such as the hardware description language used and whether the design is modeled at RTL or gate level. The user manually specifies the clocking and reset scheme for the design.

IODINE performs a multi-pass analysis on the simulation dumps by running a set of analyzers in each pass. For every pass, the analyzers for that pass are invoked at every user-specified trigger event (typically, a clock edge) to analyze all the signals in the design. Each analyzer is also given a list of signals which changed during the last clock cycle - using this information often speeds up individual analyzers by an order of magnitude. The user can also supply a list of *derived* variables, which are not directly named in the design, but are a simple boolean function of primary variables.

While the analysis framework is extensible, the specific analyzers we have implemented are oriented towards invariants often encountered in current hardware design practice, especially in the context of microprocessors: for example, mutually exclusive events, request-acknowledge protocols, state machines, scoreboards, etc.

At the end of analysis, each analyzer emits its inferred invariants into a database, or into a variety of output formats suitable for property checking tools. A confidence metric is associated with every invariant based on an invariant-specific policy. This helps us select the most likely invariants first; it tends to eliminate invariants which may be coincidences due to inadequate testing. Invariants with confidence below a user-specified cutoff are discarded. The GUI helps users view, sort and select inferred invariants.

IODINE is implemented as a multi-threaded program written in the Java programming language. IODINE analyzers are usually simple Java classes spanning a few hundred lines of code.

4. IODINE ANALYZERS

Most IODINE analyzers infer invariants on all signals in the design. An initial preprocessing pass removes some signals from the analysis: in this step, a constant signal pruner is used to prune out signals that never change. An Equals pruner identifies sets of scalar and vector signals which are equal at every clock cycle over the entire input set. (For scalars, it also detects signals which are complements of each other.) For equivalent sets, all elements except one are removed from the analysis, in order to avoid the reporting of redundant invariants by other analyzers. After this preprocessing, the following analyzers are run.

OneHot Analyzer: This analyzer identifies vectors which are one-hot or one-cold, and which have properties on the minimum/maximum number of bits in the vector which may be set to 0 or 1.

Mutex Analyzer: This analyzer identifies pairs of scalar signals whose assertion is mutually exclusive at every clock cycle. Each signal and its complement are considered. To avoid reporting redundant invariants, the mutex invariants derived are arranged canonically in an implication graph. The invariant "a and b cannot simultaneously be 1" can be expressed as the implication a $\rightarrow$ ˜b. Only non-redundant edges in this graph are reported.

State Analyzer: This analyzer records each visited state and the observed transitions for each state variable. State variable candidates are simply vector signals with a specified range of widths (between 2 and 32 bits). The invariants reported are counters (up, down or up-down) and state machines with all the possible transitions that can be taken from a particular state. This approach has the advantage that it is independent of state machine coding styles. We usually find that a vector not representing a control element (for example, a data bus) takes on random values which quickly reduce its associated confidence and cause the invariant to be discarded.

Req-Ack Analyzer: This analyzer identifies all scalar signal pairs which follow these request-acknowledge patterns:

1. **R $\rightarrow$ A**: A request followed by an acknowledge.
2. **(R1 | R2) $\rightarrow$ A**: Multiple non-overlapping requests by a single acknowledge.
3. **R $\rightarrow$ (A1 | A2)**: Single request with multiple non-overlapping acknowledges.
4. **R $\rightarrow$ n $*$ A**: A single request with n acknowledges
5. **n $*$ R $\rightarrow$ A**: n requests with single acknowledge.

For efficiency reasons, we find Req-Ack pairs in two passes. In the first pass, a frequency matrix is computed of the number of clock cycles for which each scalar signal is asserted. We make the simplifying assumption that the active polarity of a signal is the one to which it is assigned for less than half the total number of clocks (a control signal is assumed to be active less often than it is inactive). The set of candidate req-ack pairs is computed by identifying (i) pairs of signals which are active for the same number of clock cycles, for $R \rightarrow A$ type candidates (ii) sets of three signals where the sum of frequencies of two of the signals is equal to the third, for $(R1 \mid R2) \rightarrow A$ or $R \rightarrow (A1 \mid A2)$ type candidates (iii) pairs of signals where the frequency of one is a small multiple (say, < 10) of the other, for $R \rightarrow n * A$ and $n * R$

→ A type candidates. This last type of invariant is common in microprocessor designs where, for example, a single data request is acknowledged by a fixed number of data beats in response.

Once the req-ack candidates have been identified, a second pass checks if they actually follow the respective req-ack pattern. The analyzer also accumulates data like the maximum number of unacknowledged requests and the minimum/maximum number of clock cycles between a request and its corresponding acknowledge. Some req-ack invariants have a fixed interval between the req and the ack, and are reported as fixed delay invariants. In some cases, random tests end abruptly with unacknowledged requests after a fixed timeout. To support this, IODINE allows a small degree of tolerance, but assigns such invariants lower confidence levels.

Redundant req-ack invariants are removed before being reported. For example, if A → B, B → C and A → C are req-ack invariant pairs, the A → C invariant is redundant. The remaining invariants are output in a graphical format which clusters related pairs of req-acks together and is very useful for extracting protocol diagrams (see Fig 2).

Scoreboard Analyzer: This analyzer looks for design behaviors similar to in-order (FIFO) or out-of-order (scoreboard) patterns, which are common in microprocessor designs. For example, a processor issuing a series of requests to a high performance cache, tags each request with a unique ID; the cache may respond to requests out of order using the ID to match each request-response pair. The scoreboard analyzer uses the request and acknowledge signals identified by the Req-Ack analyzer as candidates for the enqueue-dequeue control signals. This analysis again runs in two passes - in the first pass, candidate data buses which may be associated with each pair of control signals are identified. In the second pass, the scoreboard analyzer checks which candidates actually follow the FIFO or the scoreboard pattern, subject to a specified maximum depth of outstanding transactions.

Note that for both Req-Ack and Scoreboard invariants, the hardware implementation may not directly mirror the invariant inferred. However, event or data flow protocols often manifest themselves as one of these invariants. For example if event A is followed by event B, then A → B is identified as a req-ack pair, even though B is not explicitly designed as an "acknowledge" for A. Similarly, data flowing down a pipeline is reported as a FIFO (with latch enable signals for each pipe stage acting as enqueue-dequeue signals), even though no explicit FIFO structure is involved. Falsification of such properties is interesting as it indicates leakage of data in the pipeline.

Any invariants involving the inputs of the block under test are emitted as constraints for the block. However, in practice, we find that these constraints can both under-constrain and over-constrain the design's input interface, so they often need manual tuning.

5. RESULTS ON A MEMORY CONTROLLER

In this section we report our experiences with using IODINE on the Memory Controller Unit (MCU) of a high-performance, dual-core SPARC microprocessor [4]. The MCU is common to the entire chip and arbitrated between requests from each core. Due to space constraints, we will restrict our discussion to the most interesting invariants detected, namely the Req-Ack and One-hot invariants.

#Signals	6012
#Verilog Modules	694
Test suite	588 tests
Analysis time	20 hours (with 12 900 MHz UltraSPARC-III CPUs)
Req-Ack Pairs	188
Fixed-Delay Pairs	326
One-hot vectors	22
One-cold vectors	2

Table 1: Results on MCU block of dual-core SPARC microprocessor

Table 1 provides statistics about this design block and IODINE analysis. The test suite used was a nearly complete regression suite for this block, consisting of 588 tests, developed over several man-months. Some properties for this design block had already been written down manually by the designers. While it is difficult to directly compare designer written invariants with IODINE generated invariants, in general, we found that human designers tend to capture protocols at a high level, while IODINE captures many more invariants at a lower level. IODINE detected several invariants on this design which were not explicitly written down by the designers (see below for example); at the same time, IODINE did not capture all the high-level properties that a designer could write. Thus, the use of dynamic invariants appears to be complementary to the process of a designer writing down invariants manually.

Among the 22 one-hot and 2 one-cold vectors, IODINE identified several one-hot/one-cold encoded state machines all of which could be correctly proven one-hot or one-cold by a verification tool. Proving one-hotness of the FSM's provided additional confidence about the state machine logic.

For the 326 fixed delay invariants, many (but not all) were proven - the rest were inconclusive. We have been experimenting with several different tools to reduce the number of inconclusives.

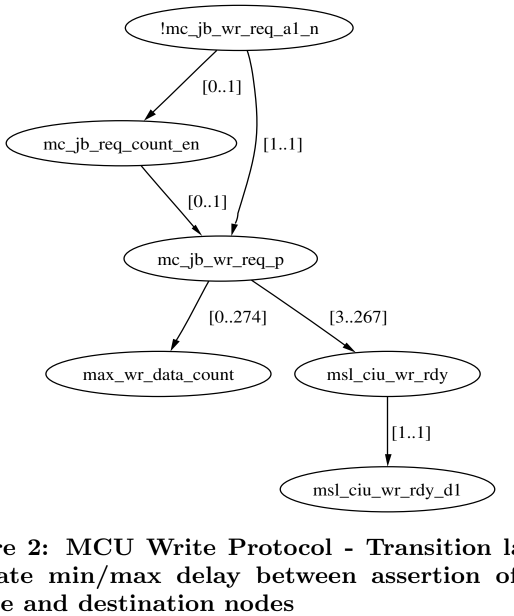

Figure 2: MCU Write Protocol - Transition labels indicate min/max delay between assertion of the source and destination nodes

Of the 188 Req-Ack invariants detected, Figure 2 illustrates one of the subgraphs generated by the Req-Ack analyzer for events in the MCU write protocol. None of the relations in this protocol was captured in the existing, manually written invariants. The edges in this figure are annotated with the minimum and maximum delay between assertion of the source and the destination. To start a write transaction, the signal *!mc_jb_wr_req_a1_n* is asserted as an early version of the start of a write request from one of the processors to the DRAM. The actual write request is *mc_jb_wr_req_p* which arrives exactly one clock cycle later (a fixed delay invariant). The memory controller responds to this request when it is ready to accept the write data, between 3 and 267 clock cycles later, by asserting its response ready signal *msl_ciu_wr_rdy.* The wide response time range indicates the variability in memory controller processing due to other actions like precharge, refresh, buffer flush, etc. This is followed by assertion of a delayed version of the response ready *msl_ciu_wr_rdy_d1* exactly one clock cycle later. Every time a write request is issued, two performance counters in the chip are also incremented, for which *!mc_jb_req_count_en* is asserted within one clock cycle of *mc_jb_wr_req_a1_n*, and *max_wr_data_count* is asserted a variable number of clocks after the request *mc_jb_wr_req_p.*

This example illustrates the power of automatic invariant discovery. Even though the req-ack invariant from *mc_jb_wr_req_p* to *msl_ciu_wr_rdy* is the most important, there is an entire family of related invariants involved in the correct working of the protocol. A human designer specifying invariants manually can easily overlook one or more of these invariants - in this case, the entire protocol was overlooked.

Another interesting type of invariant we found is the $(R1 \mid R2) \rightarrow A$ and $R \rightarrow (A1 \mid A2)$ type of req-ack invariant. For example, the following is a true invariant: A read request from a core (identified by assertion of *jba_mc_rd_req*) is followed by the request either being committed (*mc_jba_jadtype_vld_p* asserted) or cancelled (*jba_mc_xact_cancel* asserted). The following expresses the invariant in 0-IN format:

```
// 0in req_ack -single_req off -req jba_mc_rd_req
     -ack (mc_jba_jadtype_vld_p | jba_mc_xact_cancel)
```

6. OTHER APPLICATIONS

We now present uses of IODINE other than in property checking. IODINE can be useful as a tool to automatically extract some types of functional coverage information. Consider the following invariant (expressed in 0-IN format) which was discovered by the scoreboard analyzer on the same design discussed in the previous section.

```
// 0in fifo -depth 8
            -req mrq_war_chk_a2     -enq_data latest_pa
            -deq dispatch_write1_st -deq_data mwq_pa_out
```

This invariant captures an in-order FIFO which queues physical addresses (PAs) while a write-after-read check is performed in the memory controller. The address on *latest_pa* is queued upon assertion of *mrq_war_chk_a2* and after the check is performed, the same PA is issued on *mwq_pa_out* with the assertion of *dispatch_write1_st.* Interestingly, this is a real FIFO structure in the design, which IODINE was able to infer automatically; however, the designer knew that the actual depth of the FIFO as designed was 9, not 8. Therefore, this invariant pointed to a functional coverage deficiency: the test suite never exercised the FIFO full condition.

The IODINE comparator module in Fig. 1 can be used to compare dynamic invariants found on different versions of the design. This is especially important for late-stage fixes as it can highlight unintended falsification of an existing invariant.

7. RELATED WORK

Dynamic invariants were proposed by Ernst et al as a way to extract invariants for software programs [2]. Nimmer and Ernst used invariants inferred by their dynamic invariant detector Daikon in conjunction with the ESC Java static checking tool on small software programs of less than 100 lines of code[5]. Yang and Evans report experiences with inferring temporal properties between 2 variables, using QREs (Qualified Regular Expressions) and target applications related to program evolution[8]. IODINE recognizes a subset of request-acknowledge pairs their system does, but handles relations between up to 3 variables, and scales the analysis to many thousands of variables.

Real-Intent's Implied Intent tool [6] extracts simple predefined invariants based on static analysis of the RTL, which are generic and unable to capture design-specific properties. Static verification tools have often used inference of auxiliary properties to help in proving target properties [7].

8. CONCLUSIONS AND FUTURE WORK

IODINE is an extensible analysis framework for extracting detailed, low-level dynamic invariants for hardware designs. We have shown that dynamic invariants can be relevant and useful for hardware designs and can be used to help solve a thorny practical problem in formal property checking, namely the identification of properties to be statically proved or disproved. In future, we expect to explore several other applications of dynamic analysis in hardware designs.

9. ACKNOWLEDGMENTS

We thank Monica Lam for early discussions, Michael Ernst for comments on a previous version of this paper, and Ajit Pasi for implementing the IODINE FSDB interface.

10. REFERENCES

[1] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *In Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, 1579:193–207, 1999.

[2] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, Feb 2001.

[3] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[4] S. Kapil, H. McGhan, and J. Lawrendra. A chip multithreaded processor for network-facing workloads. *IEEE Micro*, pages 20–30, March-April 2004.

[5] J. W. Nimmer and M. D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, pages 11–20, 2002.

[6] S. Pollock and J. Zhang. Early and automatic error detection with Verix formal functional verification.

[7] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Proceedings of Computer Aided Verification.* Springer Verlag, 1996.

[8] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. *Fifteenth IEEE International Symposium on Software Reliability Engineering*, November 2004.