Verification Through the Principle of Least Astonishment

Beth Isaksen Valeria Bertacco
Advanced Computer Architecture Lab, The University of Michigan – Ann Arbor, MI
{*bisaksen,valeria*}*@umich.edu*

ABSTRACT

Assessing the correctness of a digital design is a challenging task hampered by extremely large circuit netlists, counter-intuitive property descriptions and ill-defined specifications. In this paper we propose a new verification methodology, inspired by the principle of least astonishment. The underlying idea is to provide an automatic assessment of what constitutes "common behavior" for a system, and use this to detect any anomaly in the design. Deviant behavior is presented to the verification engineer through intuitive, compact diagrams which lend themselves to quick inspection for correctness. To enable this methodology we introduce Inferno, a new tool which can analyze the results of a logic simulation trace and automatically extract high-level diagrams representing the design's transaction activity across any user-defined interface. In addition, Inferno can automatically generate a checker module corresponding to a given transaction, suitable for use in a wide range of verification methodologies. We envision the deployment of Inferno in a closed-loop constraint-random simulation methodology where any new transaction detected on the interface is presented to the user for analysis and, once deemed legal, it is merged into an "approved transactions" checker, which flags the detection of any new type of transactions. We provide a series of examples and experimental results to show the effectiveness of Inferno and some of its possible uses.

1. INTRODUCTION

As hardware designs continue to grow in complexity and time-to-market pressure intensifies, hardware designers and verification engineers must respond with ever-higher standards of productivity and quality. In fact, in many cases the success of the project depends on producing a sufficiently correct system at first tape-out. Formal and semi-formal verification can greatly reduce the risk of bugs escaping to silicon, but they require a significant investment of time and effort on the part of the verification team. Moreover, they typically require the user to manually specify the properties of the design which are to be proven, by deriving them from a high-level, qualitative specification document, complemented by their personally understanding of the system's functionality. Writing such properties – or checkers, when used in a simulation-centered methodology – is non-trivial in the best case, and, when the verification engineer is not completely familiar with the details of the implementation, it can verge on the impossible. Designers, who understand the inner workings of their own component, lack knowledge of the system at a global level and of the complex interactions between components; hence they are not in a position to describe complex properties, which usually affect multiple components. Recent literature provides a number of real world experiences reporting on the complexity of running a formal verification methodology in an industry context even on a very small part of the design, and on the challenges of verifying a complex system in general [12, 4]. Many factors contribute to make the verification task so error-prone and complex; however, more often than not these factors relate to the models used to describe the system:

• Designs are commonly described at the register-transfer level, which flattens the design into an amorphous structure from which it is difficult to distinguish control signals from data busses and to discern the role of different control nets.

• The most common software tools supporting the understanding of a design's behavior are waveform viewers. Their use is very cumbersome, requiring an engineer to visually inspect complex waveforms over millions of cycles in the attempt to understand the dependency chain between events and detect a design error.

• Property languages are usually declarative, making it particularly hard for a verification engineer to express a desired functionality to be verified. This is particularly true for cross-module properties, whicj may involve many signals in complex expressions, with the result that often the property is harder to debug than the design itself.

Clearly, any way of reducing the user effort required by the formal verification process would be of great use. In particular it would be greatly beneficial if abstract design behavior could be extracted automatically and presented for analysis to the verification engineer through structures which are intuitive and compact, thereby avoiding the need to inspect a large quantity of code or waveforms. Verification could then focus on checking the correctness of the high-level transactions, thus eliminating cumbersome activities and error-prone inspections such as those described above.

Contributions. The goal of the solution presented in this paper is to i) lower the barrier for the understanding of the activity of a system across a user-specified communication interface, and ii) provide a mechanism to automatically detect when anomalous activity is observed at that interface. In this context, anomalous activity stands for interactions that engineers have not yet understood or seen, which are as such potential indicators of hidden bugs. We achieve the first goal by providing a novel algorithm and a tool to automatically extract the high-level communication activity at any user-defined interface within a design (it could be the interface between two design components, or a set of signals within a single module). By raising the level of abstraction of the interaction protocol, it is easier to inspect and evaluate the correctness of the communication activity. It is also easier to detect any anomalous transaction, possibly indicative of a hidden bug. This second observation suggests a new verification methodology where the correct behavior of a system is not defined *a priori* through a set of complex properties, but instead it is surmised by analyzing the set of transaction activities detected by our tool, which can also be checked against a specification document. The second goal of this work is to provide a means to automatically detect when, in a constraint-based random simulation setting, a new "uncertified" transaction is observed. We achieve this by means of an automatic transaction checker generator which can be embedded in the system during simulation. This methodology is inspired by the informal principal of *least astonishment* which, applied to the world of design verification, says that bugs are likely to hide in the anomalous, or uncommon, behavior of a design. To support these two goals we developed Inferno, a software tool which can analyze the activity of a design interface during logic simulation and summarize the transactions observed through simple and intuitive diagrams. In addition, Inferno has the ability to automatically generate a checker flagging any uncertified transaction, making it suitable for deployment in a closed-loop random simulation environment where each new transaction is flagged and presented to the user in diagram form. If deemed correct, it is merged into the transaction-checker, which becomes incrementally complete. Otherwise, the design is updated and the process can restart.

The remainder of this paper is organized as follows: Section 2 provides an overview of related solutions both in the hardware and software verification domains. Section 3 gives an overview of the Inferno architecture and its use in the context of current checker-based and coverage-driven verification methodologies. Sections 4 to 6 present the extraction, analysis and checker-generation algorithms which enable Inferno to provide the high-level transaction models mentioned above. Finally, Section 7 and 8 provide experimental results and outline future research directions.

2. PREVIOUS WORK

A number of previous works have dealt with the problem of automatically generating properties and specifications for both software and hardware. On the hardware front, Hangal *et al.* [10] have proposed a tool to extract simple "probable" properties (*e.g.*, one-hot or mutually exclusive signals) through simulation trace analysis, which can then be fed to a formal property checker for verification. In [9], the authors propose a more general approach to automatic property extraction, by evaluating a wide range of possible "time relations" between group of signals. Our solution shares with this line of research the idea of extracting design behavior automatically from a simulation context; however, we attack the problem as a high-level modeling one, by extracting transactions observed at a user-selected interface. Moreover, although the authors of [9] attempt to generate all possible properties, they do not differentiate between data and control, thus a common control sequence (or transaction) will likely go undetected unless it is observed many times with different data. In contrast, we are able to recognize a transaction even if it occurs only once in the execution trace. The software verification community, faced with similar challenges, has arrived at some solutions which are also relevant. Ernst *et al.* [8] have proposed Daikon, which analyzes software execution traces to suggest a list of possible properties (or annotations) for use with the static checker ESC/Java. Properties can also be generated using static analysis, as in [3], where, however, the approach requires that the program be first translated into a state machine. Ammons *et al.* perform analysis to generate a specification of the API in the form of "scenarios" describing common sequences of instructions [1], while Yang derives constraints on the order of occurrence of instructions [13]. The value of such scenario- or transaction-based simulation and analysis is well recognized. Brahme *et al.*, for instance, have developed a system to allow verification engineers to write testbenches and analyze results at the transaction-level [5]. Our tool brings the benefits of transaction-based analysis to register-transfer level testbenches. Two additional important differences of our contributions with respect to the solutions outlined above are that i) we restrict our focus to control signals, so that we can abstract away the data and determine the set of transactions of the system, and ii) Inferno is not restricted to identifying a predetermined set of properties, but can analyze any control sequence observed in simulation.

A key idea driving the development of our approach lies in the empirical observation that bugs are more likely to be hidden in behavior which deviates from the norm; this is especially true as the verification process progresses and bugs become harder and harder to locate. In the verification methodology which we envision for Inferno, transactions which have not been previously seen are suspected to be faulty, and hence must be inspected by a user. This observation has also been explored in the context of software verification, for instance by Engler [7]. An example reported in [7] suggests that if a pointer dereference is normally associated with a null-check, then the one location where the null-check is missing might be an oversight. Again, in order to detect such aberrant behaviors, one must first have a good picture of what the expected behaviors are.

Finally, we have the additional objective over the previous literature of presenting the user with a high-level, intuitive model of the activity observed, to improve the understanding of the design and its correct behavior in the form of transaction diagrams. This contribution has similarities to the work of Arts *et al.* in the software world [2].

3. INFERNO ARCHITECTURE

In this paper we present Inferno, a software tool which analyzes a simulation trace to "learn" about the behavior of a design under verification. It then presents the user with intuitive diagrams describing the observed behavior as a high-level model, involving only key control signals. Inferno can automatically split the sequence of signal activity observed during simulation into *transactions*, basic sequences of activity which, when composed together, form possible execution scenarios. The user can use the transaction diagrams to learn about the design's behavior (when Inferno is run over a set of known-correct testbenches) or to quickly evaluate the correctness of the communication protocol across the interface of interest (for instance, when the trace is obtained from constrained-random simulation).

In addition, Inferno can automatically generate property checkers, in the form of Verilog assertions, corresponding to the transactions observed (and approved by the user) for direct use in any verification methodology:

- In a coverage-driven verification methodology, the distinct transactions observed, and the number of observations at a given interface provides a valuable metric to track the progress of verification.
- In constrained-random simulation, Inferno can maintain a set of "approved" transactions, and the checkers can flag the detection of any newly observed ones. A new transaction is then transformed into a high-level diagram for user inspection. If deemed correct, a corresponding checker is generated, merged into the set of approved transactions, and the simulation can continue. Otherwise, the design is modified to correct the exposed bug. A possible terminating condition can be set to a fixed number of simulation cycles with no new transactions detected. A high-level flow of this verification methodology is illustrated in Figure 1. Note that the known pool of transactions can be re-used across multiple versions of the design to quickly re-approve the correctness of an interface.
- Finally, once the user believes that the full set of possible distinct transactions has been observed, the checker generated by Inferno can be used in a formal verification context, to prove that, in fact, no other transaction can be generated at the interface under analysis.

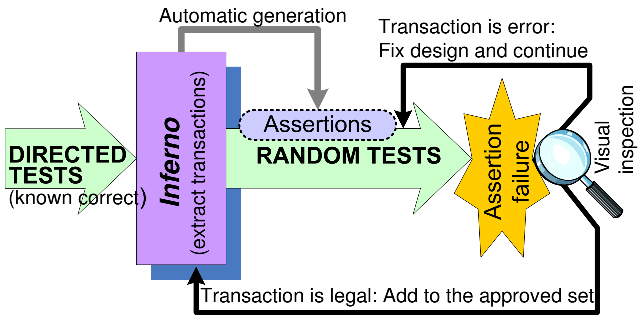

Figure 1: Closed-loop simulation with Inferno. Inferno can be used in a simulation methodology to automatically detect potential bugs. An initial set of approved transactions is extracted from a known-correct direct-simulation trace and the corresponding set of generated checkers is placed with the design. If, during the following constrained-random simulation, any new transaction is detected, it is submitted in diagram form to the user. If approved, the corresponding checker will expand the initial known set and the process continues; if not a bug has been exposed and the design modified.

From a structural standpoint (see also Figure 2) Inferno takes as input i) a small configuration file listing either a design module whose I/O interface is the target of the analysis or the specific signals to consider, ii) a simulation trace, and iii) the design under verification. This last input is only needed to determine the signals' directions at the interface of choice. The interface which Inferno considers for its analysis is a very flexible concept: it can be a communication interface of the design (such as a module I/O) or it can be custom-crafted from any combination of signals within the design. When the interface is specified as a module I/O, additional filters can be applied: for instance Inferno can automatically disregard all "busses", that is interface signals whose bit width is more than a user-specified constant.

Inferno then proceeds to process the simulation trace to extract and record the values observed over time on the interface signals. Each distinct "snapshot" of values constitutes a new vertex in the high-level diagram describing the interface protocol. Two vertices are connected by an edge to indicate that the two snapshots are subsequent. At this point Inferno can proceed in multiple ways: i) it can present the structure it has already learned as a monolithic diagram showing all the distinct activity observed at the interface under study along with its observed time dependency, called a *Protocol diagram*, or ii) it can further analyze the snapshot sequence and break it into distinct transactions presented as *Transaction diagrams* (for instance, in the case of a bus protocol, possible transactions are read, write, burst read, etc.). Section 4 describes in detail the algorithms involved in this analysis. The resulting diagrams are presented to the user for visual inspection and can be automatically converted (and simplified) into Verilog checkers.

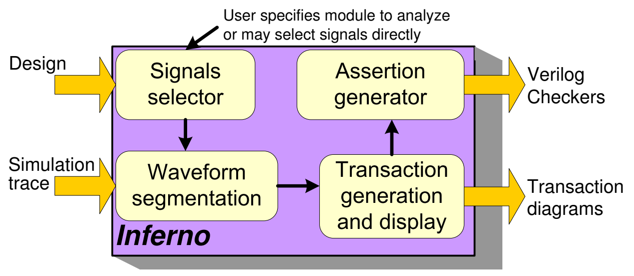

Figure 2: Inferno Architecture. The user selects a design module I/O or a specific list of signals to monitor. Inferno observes the values assigned to these signals over the course of a simulation run. It then analyzes the trace, extracting a list of transactions and presenting them in the form of high-level diagrams. In addition, it generates from the diagrams a set of optimized checkers, which can be used to detect any new transaction.

4. EXTRACTING TRANSACTIONS

This section presents the algorithms we have developed to extract the protocol and the transactions observed at the interface under observation. To build these diagrams, Inferno extracts the direction of each signal which is part of the interface under observation and processes the simulation trace to detect any value change on the signals of interest.

4.1 Generating Protocol Diagrams

Protocol diagrams are generated by building a directed graph with a vertex for each distinct pattern of values observed at the interface under study. Edges connect vertices which are consecutive in time; that is there is an edge from vertex A to vertex B if the interface transitioned from configuration A to configuration B during simulation. Practically speaking, vertices in the Inferno diagrams represents "snapshots" of values observed at the interface under analysis. Note that, due to our construction technique, there are no two vertices corresponding to the same set of interface values in a protocol diagram. Each vertex is labeled with the corresponding values observed at the interface, separating the two possible signal directions. In addition, we label each edge with the value changes which lead from the source vertex to the sink one.

Example 1. Consider a bus interface with the following I/O signals: `ack[1:0]` as input, and `cyc` and `stb` as outputs. During simulation, we observe the following interface sequence: $(00, 00)_{@0}$, $(00, 00)_{@1}$, $(00, 10)_{@2}$, $(00, 11)_{@3}$, $(00, 11)_{@4}$, $(00, 11)_{@5}$, $(10, 11)_{@6}$, $(10, 11)_{@7}$, $(00, 10)_{@8}$ and $(00, 00)_{@9}$ (subscripts indicate simulation times). Inferno's protocol diagram generator will produce a diagram with five vertices: A:00, 00, B:00, 10, C:00, 11, D:10, 11 and E:10, 00. The directed edges in the diagram are: A→B, B→C, C→D, D→B and B→A. Note that the protocol diagram abstracts away the absolute time, and only tracks time dependencies between events. □

This relatively simple procedure is already quite useful: it reduces a trace, possibly tens of millions of cycles long, to one compact image showing transitions at the interface. If an undesired behavior occurs only a few times over the course of a long regression suite, chances of identifying it through a waveform viewer are very slim, however it stands a much better chance of being detected as an anomalous vertex or edge in the corresponding protocol diagram. The case study of Section 6 shows an example of this situation.

4.2 Generating Transaction Diagrams

The analysis to extract transactions starts by processing the entire trace, labeling each distinct combination of values observed during simulation, and hence generating a chain of labels corresponding to the sequence of distinct configurations observed at the interface. Transactions are then identified by properly partitioning this initial chain of labels. In informal terms, a transaction is simply a sequence of events corresponding to a particular high-level operation, for instance a read or write to a bus, a cache, or a memory. In general a transaction is a sequence which can be easily understood as a high-level operation. On the other hand, given a sequence of events, there are many different ways to group them in transactions. Our objective is to define ways of grouping events which lead to many repetitions of few distinct transactions. We found that a technique which gives good results exploits the fact that most interfaces are designed by engineers reasoning through high level transactions, hence they tend to create very well-defined transaction boundaries in the design behavior. A key aspect in the concept of transaction is that a system generates or processes only one transaction at a time. Hence, there must be one or more signal values or transitions indicating the end of a transaction and the beginning of the next. In the simplest scenario, the user can specify which signal values or transitions correspond to transaction boundaries (for instance, in a Wishbone protocol, the assertion or de-assertion of the `stb` signal). However, in general, Inferno must infer the transaction boundaries from the trace itself. We proceed by refining the partitioning of the chain of labels into transactions through multiple passes. Below, we describe all phases of this process, and report the related pseudocode in Figure 3.

Boundary label. (lines 2, 4-5) The first pass identifies the first label of the chain that is repeated (called the boundary label). Specifically, we consider the first label to be repeated in a trace to mark the end of a transaction. Hence, we proceed in breaking the chain of labels at each occurrence of the boundary label. While this is not the only viable technique to identify transaction boundaries, we found experimentally that it works well in practice. It can be justified by noting that the stable interface value at the end of a reset sequence almost certainly marks a transaction boundary (though not necessarily the only one), and frequently it is also repeated at the completion of each transaction, hence it is observed in simulation as the first repeated label.

We also considered an alternative approach of setting the boundary label to the label with the highest number of occurrences in the trace. Intuitively, this suggests that we should extract the highest number of simple transactions. However, in practice, we have not found this second approach to work well. We believe the reason may lie in the fact that the first solution creates a better correspondence with the transaction design intent.

Loop folding. (lines 6-11) The second refinement identifies loops within a transaction. A typical scenario is a burst-read transaction, where a read sequence is repeated multiple times within a transaction. Clearly all burst-reads should be matched as the same transaction, regardless of the specific number of read operations. We can do so by identifying loops and then matching transactions which are identical except for the number of loop repetitions.

Boundary refinement. (lines 3, 13-17) The transaction extractor algorithm described so far works well in the case where all transactions end with the same label. When this is not the case, it fails to detect some of the boundaries, and two or more transactions may be clustered into a single one. The last refinement phase addresses this problem. It consists of one final pass through all the transactions identified so far checking if any transaction A is the suffix of another transaction B. That is, if B has a preamble after which it matches A completely. In this case, we can reasonably conclude that the boundary between them constitutes a new transaction boundary. At this point, we repeat the extraction process using the new boundaries in addition to the original one (and refine them through loop folding). The process can be repeated multiple times until it converges. However, we found that usually one additional pass is sufficient.

```
1  TransactionExtractor (labels_chain) {
2    new_boundary_set = first_repeated_label
3    do {
4      boundary_set = new_boundary_set
5      partition chain into segments,cut at boundary set
6      for each segment {
7        identify all repeated sub-segments
8        for each repeated sub-segment {
9          count number of occurrences
10         modify transaction to only one repetition
11       }
12     }
13     for each distinct segment pair (i,j) {
14       if (j is contained in i) then
15         new_boundary_set += label ending (i-j)segment
16     }
17   } while (new_boundary_set != boundary_set)
18 }
```

Figure 3: Transaction extractor algorithm.

At the end of this process we automatically generate a set of graphs, each showing a distinct transaction with vertices corresponding to the labels and edges corresponding to transitions between them. For each transaction we indicate the number of times it occurred, and the initial simulation time at which it was observed.

Example 2. Consider a scenario where the initial chain of labels generated for the transaction diagrams is A, B, C,

A, B, C, B, C, A, D, B, C, A. The first repeated label is A, hence the initial boundary segmentation produces 3 transactions: (B, C, A), (B, C, B, C, A), (B, C, D, A). The loop folding algorithm will then fold the second transaction and leave only two transactions: (B, C, A) and (D, B, C, A). Finally, the last refinement discovers that D is also a transaction boundary since the first transaction is a suffix of the second one. The final set of transactions is then: (B, C, A) and (D). □

While we have emphasized the usefulness of these diagrams for an engineer interested in learning more about the operation of the design, they have other benefits as well. For instance, similar to what Ammons [1] suggests, we can compare several instantiations of the same module to check if they generate the same diagrams. If not, the test coverage for some of them may be insufficient. Alternatively, we can produce a new set of diagrams for each revision of the design, enabling easy detection of unintended changes. We can also use Inferno to compare different modules which are intended to follow the same protocol. If they do not appear to correspond, either one or both of the designs is incorrect or the testbench stimulus is inadequate.

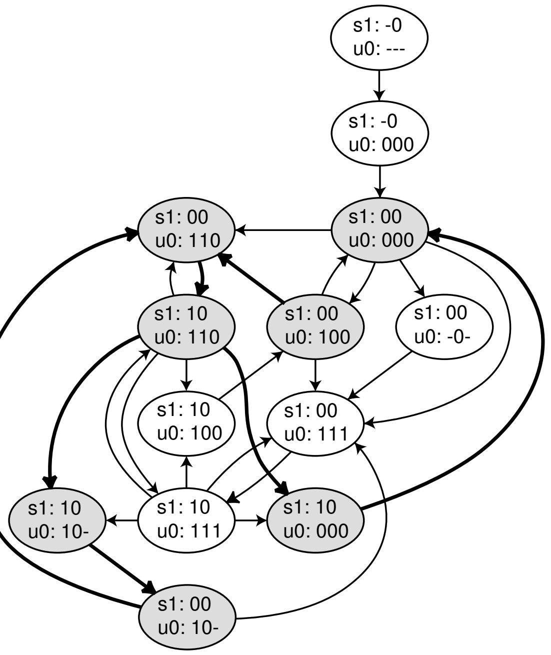

Figure 4: Protocol diagram for a Wishbone DMA. Each vertex represents a distinct combination of interface values. Edges corresponds to transitions from one combination to another. Vertices are labeled with the corresponding interface values. Grey vertices and bold edges show the correspondence with the transaction shown in Figure 5.

4.3 Example: Wishbone DMA

Examples of both protocol and transaction diagrams for an interface following the Wishbone protocol are shown in Figures 4 and 5. The protocol is primarily intended to allow communication between separate modules in a system-on-a-chip design, where a number of IP cores, possibly of different origins, must interface with each other. The design from which this diagram was extracted is a DMA (direct memory access) controller but we focus only on the Wishbone interface. Although the protocol provides for more complex cases, the only signals active in this design are `cyc`, `stb`, and `we` from the master, *u0*, and `ack` and `err` from slave *s1*. `cyc` is asserted and kept high throughout the course of each transaction, while `stb` is raised at each operation within the transaction. The `we` signal is asserted for writes and de-asserted for reads, `ack` indicates that the slave has finished processing the operation on its end, and `err` flags error conditions. The protocol diagram shown in Figure 4 includes all states and transitions observed over the course of an (extensive) regression test. Figure 5 shows the burst-read transaction, one of eight transactions extracted from the same run. The diagram shows that the transaction starts with `cyc` being asserted, followed by the assertion of `stb` to start the first operation. Since `we` is de-asserted, read is performed. When the `ack` is received, `stb` is lowered to end the operation. The burst-read transactions observed during simulation may repeat the read sequence up to three times. At the end of the transaction, `cyc` is finally lowered.

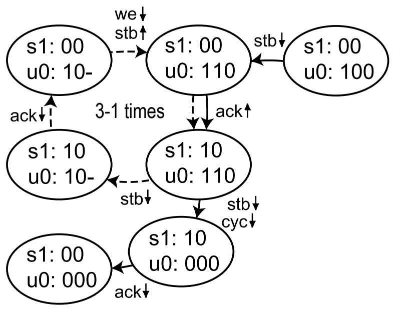

Figure 5: Transaction diagram for burst-read. The graph shows each stage of the transaction through a vertex corresponding to the values combination at the interface under study. The dashed edges highlight the loop, which is to be repeated 3 times as specified. From the diagram, it is easy to recognize the transaction's preamble, core part and trailer. Edges are labeled by the corresponding signal transitions, where a rising arrow represents a rising edge and a falling arrow a falling edge.

5. TRANSACTION CHECKERS

Once a user has inspected and approved the transaction diagrams, Inferno can automatically generate checkers in register-transfer level (Verilog) form corresponding to each transaction. These checkers can be used in a range of different verification contexts to improve the confidence in the correctness of the design, as discussed in Section 3. It is not unreasonable to say that by extracting transaction diagrams from a simulation trace, Inferno develops an "understanding" of the design, which it can use to generate transaction checkers. In practice, we attain this goal using the set of vertices and edges of a transaction to establish the legal behavior of the system. For each vertex, we generate an expression corresponding to all legal outgoing edges from it; then we build the transaction checker as the disjunction of all these expressions. The initial vertex is used to activate the checker. If the design performs a transition not described in the checker, the `fail` output signal is raised to flag a potential problem. Our initial automatically generated Verilog description is then sent through a synthesis tool and optimized. In our experiments we found that SIS [6] was sufficient to handle the complexity of the netlists generated. Alternatively, commercial synthesis tools, such as Synopsys's Design Compiler, can be used.

A checker accepting a number of separate transactions can be easily generated by combining the "fail" outputs of the individual checkers. This divide-and-conquer approach to describe a complex set of transactions has proven to be helpful in reducing the checker complexity. We wanted to verify that no un-approved transaction was occurring in the Wishbone example of Section 6. The checker describing all possible activity of the interface was too complex to be manageable in synthesis and simulation. However, when we expressed the checker as a composition of multiple transaction checkers, we were not only able to verify that the checker was an invariant across the entire simulation, but also observe which transactions was occurring. The checker generator can also be used to verify coverage. By expanding a checker to detect when the final vertex of a transaction has been observed and to include a "complete" signal, we can easily count the occurrence of each type of transaction in a coverage-driven verification context. Additional uses of the transaction checkers have been described in Section 3.

6. LEARNING FROM TRANSACTIONS

We discussed in Section 3 a number of verification methodologies where Inferno can be effectively deployed. In the context of design development, situations may arise when a clear specification of the interface protocol is missing. In this scenario Inferno can be key in developing a common understanding of the protocol through visual inspection of the diagrams, hence eliminating potential complex interface bugs. A typical example is that of interfaces between in-house components and third-party IP cores, where details of the interface may be ambiguous and access to the designers scarce. The benefits of high-level transaction modeling in Inferno are illustrated by the following case study.

While analysis results of stable designs provide interesting examples, it is ultimately more valid to observe the possible uses of a tool in a real world setting. By lending our assistance to a group of student designers, we were able to observe the beneficial effects of the protocol and transaction diagrams, both in terms of direct bug-finding (*i.e.*, identifying bugs simply by examining the diagram), and in terms of checking the equivalence of two instantiations of the same module over the course of a single test (which revealed a major difference in the coverage of the two modules).

The students were engaged in a project to design a dual-core Alpha processor with independent voltage and frequency scaling incorporated into the two cores. Each core had its own L1 instruction and data caches, and they shared a L2 cache using the MESI protocol. Only one concurrent access to the L2 cache was allowed, leading to the need for arbitration. Moreover, since the L2 cache always operated at the maximum frequency and voltage, while the cores could operate at lower frequencies and voltages, the interface between the L1 and L2 caches was asynchronous. The initial division of labor vastly underestimated the complications of the cache protocol and allocated only one of the four engineers to design all of the cache controllers and arbitration logic. As the deadline approached and the caches still could not pass simple tests, it became clear that this had been a mistake. The other three designers were recruited to aid in the debugging of the caches. However, they had very little familiarity with that part of the design. The original designer and one other engineer began together to analyze the system with a waveform viewer. At the same time, the other two team members (neither of whom was familiar with this part of the design) decided to put Inferno to use. The design was so far from being operational that it was impossible to generate a trace long enough to reliably determine the transactions; however, they could obtain a protocol diagram as the one shown in Figure 6, which they proceeded to analyze. Over the course of the same night, despite the significant difference in their background knowledge, both teams independently discovered the same bug. As indicated in Figure 6, the protocol diagram contains a transition from the idle state to a state where the signals indicate that there is data ready from the L2 cache. The most naïve examination suggests that this is suspicious, since there has obviously been no request for data from the L1 cache. Further investigation revealed that it was, indeed, a bug. The diagram for the corrected version is the same except that the marked transition is missing.

This case study also exposed the benefit of Inferno in evaluating the quality of a testbench suite. We set up an experiment which compared the transactions detected at the interface of two separate instantiations of the same module – since there were two cores, almost every module was duplicated. Once the design was sufficiently stable to execute simple programs, transactions were generated for the interfaces of the arbiter with each of the cores. The results were dramatically different, with only one transaction in common. Further analysis showed that this was because the two cores were executing the same program on the same data; hence, the one which accessed the L2 cache first would always be the first to request new data, while the other would always find the cached values ready. Consequently, the transactions were very different at the two interfaces: one would have all misses and the other all hits. This result was key in driving the development of more varied testbenches, leading to diversified coverage on both instances.

7. EXPERIMENTAL RESULTS

To evaluate the quality of our solution, we have exercised Inferno on a number of widely varying designs: the Wishbone DMA described in Section 4.3, the PCI interface of a PCI bridge, one of the cache arbiters in the dual-Core Alpha processor from Section 6, a Serial Parallel Interface (SPI) interface, the interface between the execute and memory stages of a Z80, a set of FIFO queues, a Reed-Solomon decoder, and a USB protocol design. All but the cache arbiter testbench are available in [11].

Testbench	gates	FF	cycles	if.nets
Wishbone DMA	1,972	672	1,759,678	5
PCI protocol	334	95	8,298,177	13
Dualcore Alpha	109,441	22,608	9,310	10
SPI protocol	4,578	1,345	1,999	9
Z80 μp	3,628	277	14,436	16
FIFOs	4139	2091	2612259	12
R.-S. Decoder	987	231	272	19
USB	292	98	1517565	13

Table 1: Testbenches characteristics and setup. The table shows for each testbench: number of gates and flip-flops, the number of simulation cycles in the input trace for Inferno, and the number of nets (signals) which are part of the interface.

Table 1 provides a brief characterization of the experimental testbenches and Table 2 lists the results of our analyses. As shown in Table 1, the number of bits monitored on any one interface ranges from 5 to 16, and the lengths

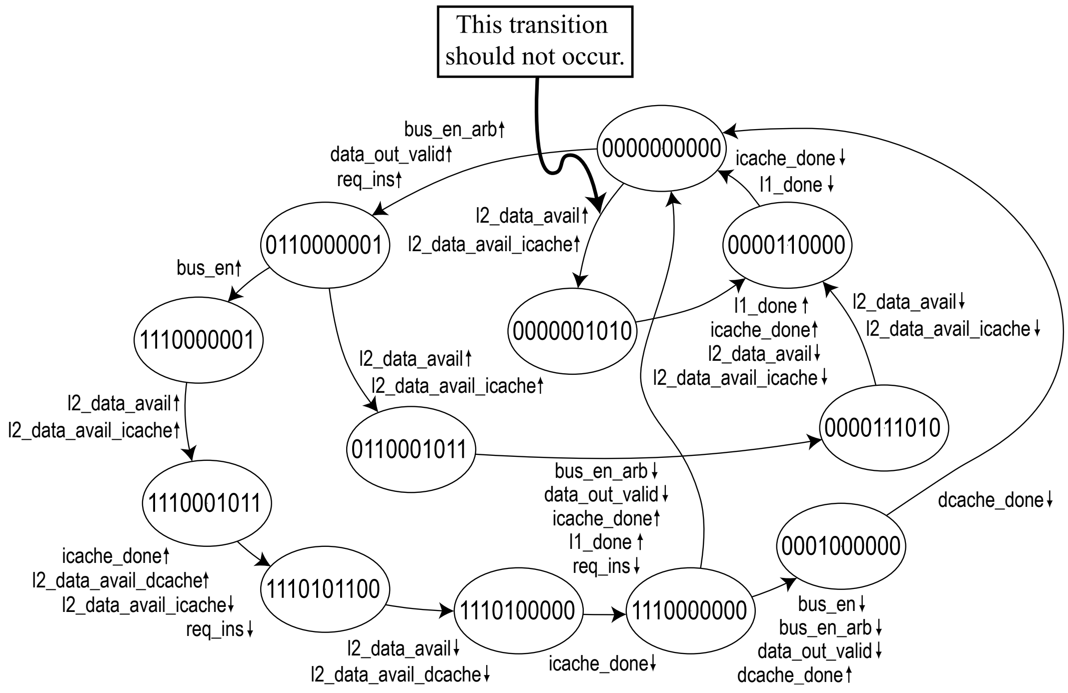

Figure 6: Protocol diagram for the cache arbiter interface in the dual-core Alpha processor. Notice that the labeled edge shows the interface leaving idle state and entering a configuration where the arbiter signals data ready from L2 cache. This is clearly a bug, since no one has requested any data, and yet it is there. This case study presents a situation where direct inspection of a simple protocol diagram yields insightful understanding and debugging support.

of the simulation traces to be analyzed vary up to 8 million cycles. Table 2 shows that the bus interfaces and cache arbiter have a number of well-defined transactions which we are able to detect, while the pipeline stage interface of the Z80 does not follow a clear transaction pattern, as one would expect. Only three potential transactions could be identified for the Z80 interface, and inspection of the results suggests that these are not actual transactions, but simply coincidental loops of instruction sequences in the testbench. Surprisingly, only three transactions were identified for the SPI interface. Further inspection of the resulting diagrams reveals that these do indeed correspond to behavioral transactions, and the small number just reflects the simplicity of the protocol: they are really the only transactions which can occur. The other protocols all respond well to the analysis, yielding a number of transactions.

Testbench	prot. nets	distinct transact.	total states	repeat states
Wishbone DMA	5	8	10	10
PCI protocol	13	208	310	60
Dualcore Alpha	10	6	13	7
SPI protocol	9	3	28	1
Z80 μp	16	3	23	1
FIFOs	12	4	59	54
R.-S. Decoder	19	3	22	3
USB	13	40	25	20

Table 2: Transaction analysis. The table reports for each test the number of signals in the interface, the distinct transactions which Inferno could extract from the simulation trace, the vertices in the protocol diagram, and the number of interface configurations which are part of more than one transaction.

We also note that in all cases, including the Z80, the number of distinct configurations actually observed at the interface (and therefore assumed to be legal) is far smaller than the number of possible states, that is, $2^{(\#signals\_monitored)}$, supporting the argument that protocol and transaction diagrams are actually a much more compact way to evaluate the behavior of a system compared to a waveform viewer. It also indicates that the checkers generated are fairly compact, since they only span the small set of configurations actually observed. Moreover, the protocol diagram has few transitions compared to a clique, suggesting that the extraction of transaction is indeed beneficial in understanding the transition patterns between distinct interface configurations. This point is also supported by the observation that the configuration is frequently repeated in more than one transaction, suggesting that it is not only the combination of signals itself, but the context in which it appears, which should interest us.

Finally, we have plotted the detection of transactions over time (measured in simulation cycles) for three designs with significant numbers of transactions. In Figures 7, 8 and 9 we marked the simulation cycle of first discovery for each transaction. Note the periodic bursts in new transactions observed, which could correspond to when the regression testsuite switches to a new phase, stimulating different aspects of the design.

8. CONCLUSION

In this paper we presented Inferno, a software tool to automatically extract, or infer, transactions from a simulation trace over a user-selected interface. Inferno presents the results of its analysis through intuitive "protocol diagrams", reporting the overall protocol observed at the interface, and "transaction diagrams", showing the flow of each distinct type of transaction detected. In addition, Inferno can automatically generate transaction checkers in Verilog, that is, checkers which monitor the execution of a transaction. Checkers can be used in the context of a constrained-random simulation, or a coverage-driven methodology, and also in a formal verification setting.

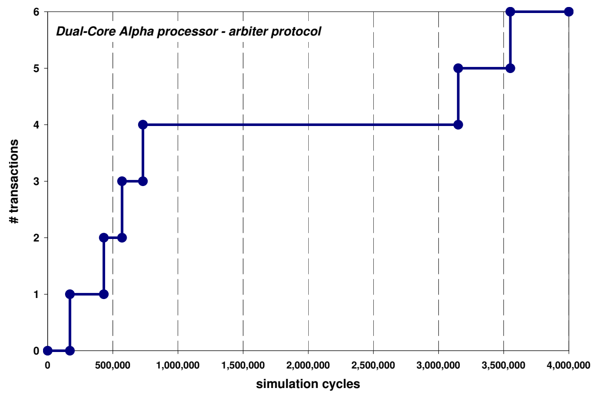

Figure 7: Cache arbiter. Number of distinct transactions detected over time for the cache arbiter of the dual-core Alpha.

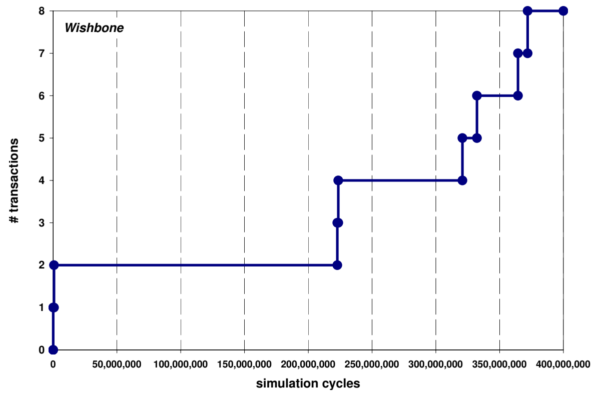

Figure 8: Wishbone DMA. Number of distinct transactions detected over time for the Wishbone DMA design.

In the context of random simulation-based verification, Inferno enables a novel verification methodology, inspired by the principle of least astonishment, by automatically extracting any new transaction observed at the selected interfaces, and presenting it to the user for evaluation through simple, high-level diagrams, thus allowing a verification engineer to focus in on the uncommon aspects of a design's behavior, in the hope of uncovering hidden bugs. The case study in debugging a dual-core Alpha processor demonstrates Inferno's usefulness in practice, even for large designs. In addition, experimental evaluation on a range of designs indicates that Inferno is effective in summarizing the common behavior of a system and presenting it to the user through simple and intuitive diagrams. We plan to explore this methodology further by developing additional techniques to decompose the inherent structure of an interface activity into simple components and by investigating more scalable solutions for our assertion generation engine.

9. REFERENCES

[1] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[2] T. Arts and L.-A. Fredlund. Trace analysis of erlang programs. In *ACM Sigplan Notices*, pages 18–24, 2002.

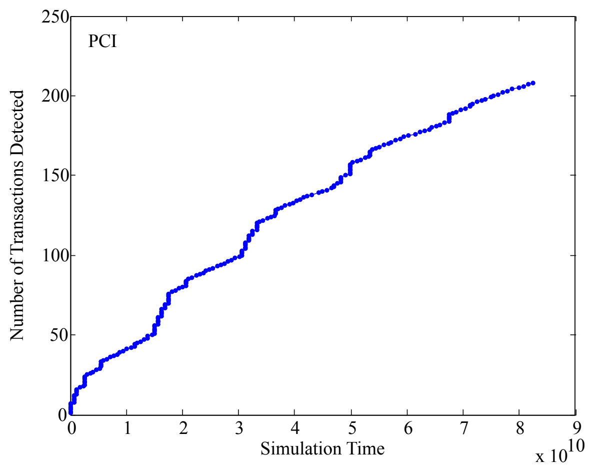

Figure 9: PCI bridge. Number of distinct transactions detected over time for the PCI bridge design.

[3] S. Bensalem, Y. Lakhnech, and H. Sadi. Powerful techniques for the automatic generation of invariants. In *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 323–335, Aug. 1996.

[4] B. Bentley and R. Gray. Validating the Intel Pentium 4 Microprocessor. *Intel Technology Journal*, pages 1–8, 2001.

[5] D. S. Brahme, S. Cox, J. Gallo, W. Grundmann, C. N. Ip, W. Paulsen, J. L. Pierce, J. Rose, D. Shea, and K. Whiting. The transaction-based verification methodology. Technical report, Cadence Design Systems, Inc., Aug. 2000. Technical Report No. CDNL-TR-2000-0825.

[6] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, 1992.

[7] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.

[8] M. D. Ernst. Verification for legacy programs. In *Verified Tools: Theories, Tools, Experiments*, Zürich, Switzerland, October 10–13, 2005.

[9] G. Fey and R. Drechsler. Improving simulation-based verification by means of formal methods. In *ASPDAC, Proceedings of the Asia South Pacific Design Automation Conference*, pages 640–643, Jan. 2004.

[10] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 775–778, New York, NY, USA, 2005. ACM Press.

[11] http://www.opencores.org.

[12] T. Schubert. High-level formal verification of next-generation microprocessors. In *Proc. DAC*, pages 1–6, June 2003.

[13] J. Yang and D. Evans. Automatically inferring temporal properties for program evolution. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 340–351, Nov. 2004.