Inferno: Streamlining Verification With Inferred Semantics

Andrew DeOrio, *Student Member, IEEE*, Adam B. Bauserman, *Member, IEEE*, Valeria Bertacco, *Member, IEEE*, and Beth C. Isaksen, *Member, IEEE*

***Abstract*—Understanding designers' intentions and accurately verifying a design are major obstacles for verification engineers today. Currently available debugging tools, such as waveform viewers, are unwieldy, often requiring the user to search through millions of cycles of logic simulation data to locate a problem. In this paper, we present Inferno, a novel solution capable of automatically extracting semantic information from a design's interface from simulation information. The semantic structure of an interface's communication protocol is presented to the user as a set of transactions, that is, monolithic communication units that have typically been observed several times during the logic simulation. Transactions can graphically be presented to the user and used as an aid to understand and validate the communication protocol of a design's interface. In addition, approved transactions can also be encoded as assertions expressed in a hardware description language (HDL) and used in constrained-random simulation to certify that the interface protocol adheres to the set of observed (and user-approved) transactions. Moreover, we developed a new closed-loop verification methodology based on Inferno, called transactional verification, which leverages approved transactions to describe correct design behavior. In our methodology, transactions are concurrently extracted during a constraint-based random simulation: the anomalous ones are flagged as potentially buggy and presented to the user for inspection. In the experimental results, we evaluate the performance and the quality of the results of Inferno on a broad range of testbench designs and several of their interfaces, including a number of communication intellectual properties and the OpenSPARC T1 8-core processor from Sun.**

***Index Terms*—Finite-state machines, formal verification, hardware description language (HDL), logic simulation, simulation.**

Manuscript received March 28, 2008; revised July 31, 2008 and October 13, 2008. Current version published April 22, 2009. An earlier version of this paper was published in [18]. This document includes the following additional contributions: 1) a new dynamic closed-loop verification methodology based on Inferno; 2) extended experimental evaluation on additional testbenches, including OpenSPARC T1 and more in-depth experimental analysis of all testbenches; and 3) several optimizations and extensions on Inferno, including statistical line weighting (to highlight the most common activity), algorithm optimizations based on intersignal correlation, and simulation trace sectioning to enable our dynamic verification methodology. This paper was recommended by Associate Editor R. F. Damiano.

A. DeOrio and V. Bertacco are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2121 USA (e-mail: awdeorio@umich.edu; valeria@umich.edu).

A. B. Bauserman was with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2121 USA. He is now with NVIDIA, Santa Clara, CA 95050 USA (e-mail: adambb@umich.edu).

B. C. Isaksen was with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2121 USA. She is now with Jasper Design Automation, Mountain View, CA 94041 USA (e-mail: bisaksen@umich.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2009.2013995

I. Introduction

THE FAST growing complexity of digital hardware designs is exacerbating the challenges of validation and debugging. Several factors contribute to the time-consuming nature of these incomplete and *ad hoc* tasks, demanding more engineering resources than the design itself. First, debugging is commonly done through a waveform viewer, an arduous task due to the low-level models used to describe a system [i.e., register–transfer level (RTL)], as well as extremely long simulation traces. Second, the growing adoption of semiformal verification methodologies to complement simulation-based verification requires engineers to express the correct behavior of a system through a set of properties. However, property description languages are often declarative; hence, their use is error-prone and adds further challenges for designers, who are commonly trained to use imperative languages. Even worse, these properties are often the result of the personal understanding of a verification engineer who has studied a high-level specification document of the system.

In this landscape, it is clear that there is a critical need for novel verification solutions that automate the verification of the correspondence between a high-level description and a low-level model of a system to relieve designers from the cumbersome and time-consuming tasks described above, and let them refocus their efforts on verifying the correctness only of a high-level description of the whole system.

A. Contribution of This Paper

In this paper, we propose a novel approach to streamline the engineering effort dedicated to verification by presenting a technology to automatically extract the semantic protocol of any communication interface in a design from simulation data. The protocol is then summarized in a compact abstract diagram, which retains all the key behavioral aspects observed while removing the low-level timing information. This form, which is called a *transaction diagram*, represents the semantic behavior of the communication interface.

Transaction diagrams are then presented to verification engineers who can inspect them to spot potential design errors or to approve the design's functionality based on their knowledge of the specification document. Transaction diagrams are extremely compact and summarize the design's behavior in an intuitive fashion, leading to a much more efficient verification methodology than that enabled by waveform viewers and RTL descriptions. Our solution accomplishes this goal by analyzing the simulation traces at the target interface and distilling the

relevant information, resulting in a high-level description of the system.

We implemented our proposed solution in a software tool called Inferno—because it "infers" the design's behavior—and evaluated its quality and performance on a wide range of testbench designs and interfaces, both communication-centered and not. Moreover, we developed a new closed-loop verification methodology based on Inferno, whereby our tool continuously monitors an interface during constraint-based random simulation; any new transaction detected is presented to the user, and it can either be approved—and added to a database of acceptable activity—or tagged as buggy, thus leading to a new design's revision. In addition, Inferno can generate design assertions in several hardware and verification languages, including Verilog, based on the approved transactions. These assertions can flag any unapproved activity in simulation or be used as properties in a formal or semiformal verification methodology, eliminating the challenging effort of property development by the engineering team.

Our proposed approach greatly streamlines the resources needed for the verification and debugging of communication-centered designs by attacking the problem from several directions.

1) It greatly reduces the need to rely on waveform analysis to debug a design.
2) It lowers the barrier to understanding the design's RTL implementation.
3) It eases the adoption of formal verification methodologies by automatically generating properties related to the protocol's approved behavior.

Finally, understanding and verifying a design at the transaction level lead to a new verification methodology, where the correct behavior of a system is not defined *a priori* through a set of complex properties, but rather by inspecting and approving the set of transactions observed in simulation, and where bugs are spotted by anomalous transactions. Inferno extracts these transactions as simulation progresses, automatically matching previously approved ones, and only presenting new activity to the user.

II. Related Work

A. Hardware Verification

A number of previous works have focused on the problem of automatically extracting properties and specifications in both the software and hardware contexts. On the hardware front, Hangal *et al.* [17] 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. Fey and Drechsler [12] proposed a more general approach to automatic property extraction by evaluating a wide range of possible "time relations" between groups of signals. Our solution shares with this line of research the idea of automatically extracting design behavior from a simulation context; however, we attack the problem from a high-level modeling standpoint by extracting transactions observed at a user-selected interface. Moreover, although Fey and Drechsler [12] attempted to capture into properties all the possible relations between the design's signals, they do not differentiate between data and control; hence, even a common control sequence (or transaction) may go undetected, unless it is observed several times operating on different data. In contrast, we are able to recognize a transaction even if it occurs only once in the execution trace. An earlier version of this paper was published in [18], proposing similar core algorithms. However, that work did not include any solution for integrating the approach into a dynamic closed-loop verification methodology. We address this problem here along with several performance boosting and usability improvements. Rogin *et al.* [22] also leveraged some of the ideas in [18] to extract complex candidate properties from a simulation trace and a baseline set of simple properties known to hold up front. The inclusion of data signals in the analysis limits the scalability of this solution because of the large body of signals to be considered.

B. Software Verification

The software verification community, faced with similar challenges, has also developed a number of relevant solutions. Ernst [11] has proposed Daikon, a tool that 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 [1] and [5]; however, in [5], a program must first be translated into a state machine, a step that may add notable complexity to the process. Ammons *et al.* [2] performed an analysis to generate a specification of a given application program interface (API) in the form of "scenarios" describing common sequences of instructions, whereas Yang and Evans [26] derived constraints on the order of occurrence of instructions. The value of such scenario- or transaction-based simulations and analyses is well recognized. Brahme *et al.* [7], for instance, have developed a system to allow verification engineers to write testbenches and analyze results at the transaction level. Our tool brings the benefits of transaction-based analysis to RTL testbenches, and, when restricting our focus to control signals, we can leverage common patterns of behavior to recognize more general data-independent transactions.

A key concept driving the ideas developed in Inferno is the empirical observation that bugs are more likely to be hidden in behavior that deviates from the norm; this is particularly true as the verification process matures, and bugs become harder to locate. In the verification methodology, which we propose in this paper, transactions that have not been previously observed are suspected to expose design flaws and, hence, must be inspected by a verification engineer. This observation has also been explored in the context of software verification, for instance, by Engler *et al.* [10]. An example reported in [10] 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. Anomaly detection strategies that are used to detect intrusions through abnormal behavior such as in [14] leverage similar techniques to provide software security. Again, to detect such aberrant behaviors, one must first have a good picture of the typical behavioral pattern.

C. Regular Language Recognition

The techniques we use to derive transactions from a simulation trace build on the body of knowledge in regular languages and finite automata. Given a design's interface that we want to analyze, it is possible to view the distinct values observed at the interface during simulation as characters of a language's alphabet. Then, the protocol diagram that we derive would be a nondeterministic finite-state automaton (NFA) that partly recognizes the regular language expressed at the interface under analysis. Specifically, our protocol diagram is learned only from the strings of the languages that have been observed in simulation. Moreover, the complete language expressed at the interface may even be irregular; however, our diagrams would only interpret the strings observed as part of a regular language.[1] The classic work by Gold [15] showed that it is not possible to recognize a regular language by simply observing strings in the language, not even if the strings span the full set of strings that can be generated in the language (i.e., a text). Indeed, we only recognize the portion of the language that can be inferred by the strings observed during simulation. Our extractor algorithm closely corresponds to the k-tails algorithm proposed by Biermann and Feldman [6], which builds an NFA recognizing a portion of a regular language through the pool of strings observed. The algorithm has various degrees of nondeterminism based on a user parameter k; our protocol diagram can be viewed as having a correspondence with the one-tail instance of the algorithm. We apply these classic results in theoretical computer science to the task at hand, simplifying the interpretation of the simulation activity at a design interface and automatically detecting when anomalous behavior occurs. Moreover, after extracting the protocol diagram, i.e., the baseline NFA recognizing the observed interface behavior, we derive from it a set of transaction diagrams. The purpose of the transaction diagrams is to interpret the activity at a high level of abstraction, hopefully corresponding to the designer's intent in developing the interface's communication protocol.

D. Intuitive Visualization

The use of graphs to visualize design functionality has been explored by both the hardware and software camps. On the software side, Arts and Fredlund [4] use a directed graph to represent a program's execution trace for visualization and model checking. Inferno can generate a similar result when directed at the program counter of committing instructions, with the additional capability of fragmenting the trace graph into transactions representing frequent behavior. Clark *et al.* [8] use data flow graphs to select a set of instructions that are good candidates for hardware acceleration. Also, working at the instruction level, Fields *et al.* [13] use a dependence graph model to find the critical path of high latency instructions, using this information for hardware optimizations. Inferno's transactions could be leveraged to make decisions about hardware optimizations in a similar fashion; those that exhibit a high number of repetitions would be particularly good candidates. Inferno can also convey the frequency of occurrence of each activity at the interface of interest through the use of weighted edges. Reference [25] presents a similar approach to convey hardware resource utilization in the context of processor design. This is useful for identifying bottlenecks and considering architectural tradeoffs.

[1] An example of an irregular language is an interface that transmits encoded values, and the coding scheme balances the number of zeroes and the number of ones for security, power, or clock extraction purposes.

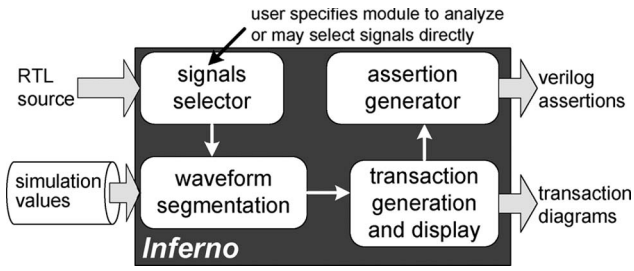

Fig. 1. Inferno architecture. The user selects a design module I/O or a specific list of signals to monitor. Inferno observes the values assigned to these signals over the course of a simulation run. It then analyzes the trace, extracting a list of transactions and presenting them in the form of high-level diagrams. In addition, it generates a set of optimized assertions from the diagrams, which can be used to spot any new transaction.

III. Inferno Overview

We present Inferno, a software tool that analyzes a simulation trace and its corresponding source code to learn about the behavior of the design under verification and to highlight aberrant behavior. Inferno presents the user with intuitive diagrams describing the behavior observed at the target interface. The target interface is generally a meaningful subset of the signals in the design characterizing an interface or component to be verified. Transactions that are automatically inferred by Inferno are called *transaction diagrams*; they represent basic sequences of activity that form a high-level picture of the design's functionality. The goal of Inferno is to represent all the behavior contained in the simulation trace using as few transactions as possible, while clearly separating unrelated events.

An example of a transaction is a burst read, which often manifests itself as a sequence of signal configurations repeated several times, but with a varying number of repetitions. Inferno is capable of automatically ascertaining that such an event is significant and presents it as a single transaction. Transaction diagrams can be used to learn about a design's behavior or to quickly evaluate the correctness of a protocol across the interface of interest, for instance, when a trace is obtained from constrained-random simulation.

From a structural standpoint (see Fig. 1), Inferno takes as input a small configuration file, a simulation trace, and the design under verification. The configuration file lists either a design module whose I/O interface is the target of the analysis or the specific signals to consider. The source code of the design under verification is used only to determine the signals' directions at the interface of choice. The range of design interfaces that Inferno may consider for its analysis is very flexible, ranging from any communication interface of the design (such as a module I/O) to any custom-crafted set of signals within the design.

```
1:ProtocolExtractor (sequence) {
  create_vertex (s_0)
2:for i=1 to t_max {
3:    if !vertex_exists(s_i) create_vertex (s_i)
4:    if (s_i ≠ s_{i-1}) create_edge (s_{i-1}, s_i)
5:}
```

Fig. 2. Pseudocode for the protocol diagram generation algorithm. The algorithm generates a graph representing all behavior observed at the target interface, abstracting away time.

Inferno generates two types of directed graphs—protocol diagrams and transaction diagrams. A protocol diagram includes a vertex for each unique combination of signal values observed at the target interface. Vertices are connected by an edge if the corresponding signal combinations follow each other in the simulation trace. Transaction diagrams attempt to grasp the high-level semantic behavior of the design by partitioning the simulation trace into time intervals, with each interval corresponding to a transaction (any transaction can occur multiple times in a trace). Moreover, we deploy several techniques to recognize similarities among different intervals in an attempt to reduce them to a small set of transactions.

A. Protocol Diagrams

Inferno begins its analysis by generating a protocol diagram—a time-independent graph that describes all the behavior observed at the interface under analysis. Each vertex of the graph represents a unique observed signal combination, whereas the edges show transitions between these signal combinations. An edge from vertex A to B indicates that, at some point in the simulation, the interface signals transitioned from the values in A to the values in B. As a result of time abstraction, all vertices have implicit self-transitions (see Fig. 2).

Example—Protocol Diagram Generation: Consider a bus interface with I/O signals `in[1 : 0]` as input and `out[1 : 0]` as output. During simulation, we observe the interface sequence shown on the left part of Fig. 3, where (00,00) indicates (*input bits*,*output bits*) at a given time step. Inferno's protocol diagram generator will produce a diagram with four vertices: $A : (00, 00)$, $B : (00, 10)$, $C : (00, 11)$, and $D : (10, 11)$. The directed edges in the diagram are $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, $D \rightarrow B$, and $B \rightarrow A$. Note that the protocol diagram abstracts away absolute time and only tracks time dependencies between events. The resulting protocol diagram is shown on the right in Fig. 3. 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.

B. Transaction Extraction

After generating a protocol diagram, where each new signal combination observed at the interface triggers the generation

Cycle	Value
1	(00,00)
2	(00,00)
3	(00,10)
4	(00,11)
5	(00,11)
6	(00,11)
7	(10,11)
8	(00,10)
9	(00,00)

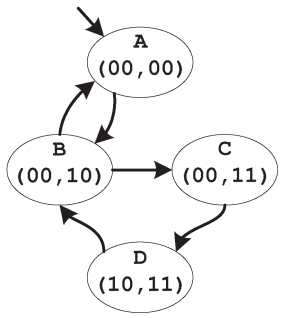

Fig. 3. Protocol diagram example. The table shows values observed at the interface under study for nine consecutive cycles; (00,00) represents (*inputs bits*, *output bits*). The resulting protocol diagram is shown on the right. The diagram includes a vertex for each distinct signal combination observed during simulation. Edges indicate transitions between two vertices. For instance, the signal transition between cycles 2 and 3 corresponds to the edge between A and B.

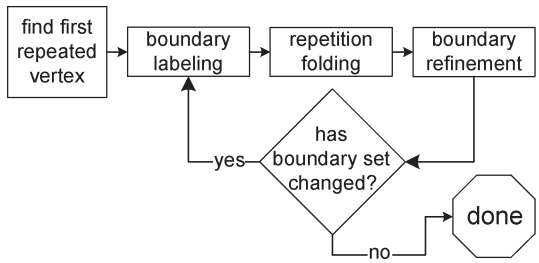

Fig. 4. Flowchart of Inferno's transaction extraction algorithm. The algorithm is composed of three major stages: boundary identification, repetition folding, and boundary refinement.

of a new vertex, tagged by a unique ID, Inferno continues with transaction extraction. The entire simulation can now be encoded as a sequence of these IDs, and transactions can be extracted by analyzing this sequence. The transaction extraction algorithm breaks the sequence of IDs down into subsegments called prototransactions, which are refined by the algorithm until they reach their final shape as transactions.

Note that transaction diagrams are subgraphs of the protocol diagram by construction since they are both extracted by folding the linear sequence of unique IDs. Both diagrams have the same semantic meaning for vertices and edges; however, the transaction extraction algorithm partitions the sequence into segments, and, thus, transactions correspond to subgraphs of the protocol diagram. Moreover, this algorithm implements additional folding and reductions on the sequence compared with the protocol extraction algorithm. It is applied to the complete sequence of IDs and operates in three steps (see Fig. 4):

1) boundary identification and labeling;
2) repetition folding;
3) boundary refinement.

The extraction algorithm starts with the ID sequence, partitions it into prototransactions, and then proceeds to refine these until they are completely distilled down to what we call transactions. A prototransaction is a segment of our initial ID sequence that will be transformed through the algorithm to obtain an instance of a transaction. The following three sections expand on each of these components; line numbers refer to the pseudocode of the algorithm, shown in Fig. 5.

```
 1:TransactionExtractor (sequence) {
 2: new_boundary_chars =
 3:       first_repeated_vertex(sequence)
 4: do {
 5:   boundary_chars = new_boundary_chars;
 6:   segments = split(sequence,boundary_chars)
 8:   for each segment {
 9:     sub-segs = substr(segment,length > 1 &&
10:                 occurring > 1 consecutively)
12:     for each unique sub-seg {
13:       count_repetitions(sub-seqs, sub-seg)
14:       remove_all_but_first(sub-seg,segment)
15:     }
16:   }
17:   for each pair of segments (i,j) {
18:     if (j is a suffix of i)
19:       new_boundary_chars += last_char(i-j)
20:   }
21: }while(new_boundary_chars!=boundary_chars)
22:}
```

Fig. 5. Pseudocode for the dynamic transaction extraction algorithm. The algorithm progresses in three stages: boundary identification and labeling, repetition folding, and boundary refinement.

Boundary Labeling (Lines 2–3): The very first goal of the transaction extraction algorithm is to identify the boundaries between prototransactions. At this stage, we only perform an approximate boundary identification, which is further refined in the subsequent phases of the algorithm. Initially, the boundary marker is defined as the first repeated ID tag. This ID is identified by analyzing the trace from the beginning of the simulation, and it indicates the completion of a prototransaction. We use this ID to partition the simulation trace into multiple prototransactions; each of them will eventually become an occurrence of a transaction. The intuition behind this heuristic approach is that the stable interface value at the end of a reset sequence almost certainly marks a prototransaction boundary (although not necessarily the only one), and, frequently, it is also repeated at the completion of each prototransaction; hence, it is observed in simulation as the first repeated label. While this is not the only viable technique to identify transaction boundaries, we experimentally found that it works well in many practical cases. We have also considered an alternative approach of setting the boundary to the label with the highest number of occurrences in the trace. Intuitively, this suggests that we should extract the highest number of transaction instances. However, we have not found this second approach to produce good results. We believe that the reason may lie in the fact that the former approach creates a better correspondence with design intent. In addition, in our implementation, we provide the option for a user to specify a signal in the design whose rising or falling edge indicates the start of a transaction. Some designs have such a signal readily available, for example, the request signal in an arbiter.

Repetition Folding (Lines 8–14): In this phase, we identify repetitions within each prototransaction identified during boundary labeling. A typical scenario is a burst-read transaction, where a read sequence is repeated multiple times within the same segment. Clearly, all variants of burst-read operations should be matched as the same type of transaction, regardless of the specific number of reads in each prototransaction. We achieve this by identifying repetitions within each segment and then matching segments that are identical except for the number of repetitions. At this point, we have obtained a baseline set of prototransactions.

Boundary Refinement (Lines 17–19): The transaction extractor algorithm described so far works well in the case where all prototransactions do actually terminate with the same ID. When this is not the case, it fails to detect some of the boundaries, with the result that a chained sequence of several transactions may be clustered together in one single prototransaction. The last refinement phase addresses this problem by refining the boundary set. It consists of one final pass through all the prototransactions identified so far, checking if any prototransaction A is the suffix of another prototransaction B, i.e., if B is composed of a preamble followed by A. In this case, we can reasonably conclude that the boundary between them constitutes a new transaction boundary. At this point, we would repeat the extraction process using the new boundaries in addition to the original one and regenerate the prototransactions through the repetition folding phase. The process is repeated until we reach convergence. Now, all prototransactions have been refined to final transactions. We found in practice that one refinement pass is usually sufficient: All of our testbenches converged with one single refinement step.

Upon completion of this process, Inferno has generated a set of diagrams, each representing a distinct transaction. For each transaction, we display the corresponding diagram and report the number of times it occurred during the simulation and the simulation time of its first occurrence. Inferno's inference algorithm does not track the dependence of a transition from past events (that is, past signal combinations that occurred at the interface under study), in that it is similar to a Markov model that only grasps the relation between the present and the immediately previous state. We experimentally found that the simplicity of this algorithm works well in hardware designs. We believe that this is due to the fact that designs frequently carry information relating to past events in storage elements, whose values we can simply monitor with Inferno.

Algorithm Complexity: In the worst case, the complexity of Inferno's transaction extraction algorithm is $O(v^3)$, where v is the number of IDs (vertices) in the initial ID sequence representing the simulation trace. In fact, boundary labeling may require, in the worst case, a pass through the entire sequence $O(v)$. Repetition folding checks for a substring within each prototransaction, and boundary refinement checks for substrings between distinct prototransactions; hence, the worst-case complexity of both of these steps is $O(v^2)$. Finally, note that we may need to iterate these three steps several times, as shown in Fig. 5, in the worst case, v times. Thus, the overall complexity of the algorithm is bound by $O(v(v + v^2 + v^2)) = O(v^3)$.

Example—Transaction Extractor: Consider an initial chain of IDs generated after reading the simulation trace: A, B, C, D, B, C, D, C, D, B, C, E, C, D, B. The first repeated label is B; hence, the initial boundary segmentation produces four prototransactions: AB, CDB, $CDCDB$, and $CECDB$. The repetition folding algorithm will then fold the repetition of CD in the third prototransaction. This leaves three prototransactions: AB, $(CD)^{1,2}B$, and $CECDB$. Boundary refinement discovers that E is also a transaction boundary since CDB is a suffix of $CECDB$. Thus, we arrive at the final set of transactions: AB, $(CD)^{1,2}B$, and CE.

Example—Wishbone DMA: The Wishbone protocol enables the communication between independent modules in a system-on-a-chip design, where a number of IP cores, possibly of different origins, must interface with each other. Examples of the protocol and transaction diagrams that we extracted from an interface following this protocol are shown in Figs. 6 and 7. The design from which this diagram was extracted is a direct memory access (DMA) controller. Although the protocol provides for more complex cases, the only signals active in this design are `cyc`, `stb`, and `we` from the master M, and `ack` and `err` from the slave S. The `cyc` signal is asserted and kept high throughout the course of each transaction, whereas `stb` is raised for each operation within the transaction. The `we` signal is asserted for writes and deasserted 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 Fig. 6 includes all signal configurations and transitions observed over the course of an extensive regression test (millions of cycles). Fig. 7 shows the burst-read transaction, which is one of the eight transactions extracted from the same run. The diagram shows that the transaction starts with `cyc` asserted, followed by the assertion of `stb` to start the first operation. Since `we` is deasserted, a 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 (the longest burst observed). At the end of the transaction, `cyc` is finally lowered.

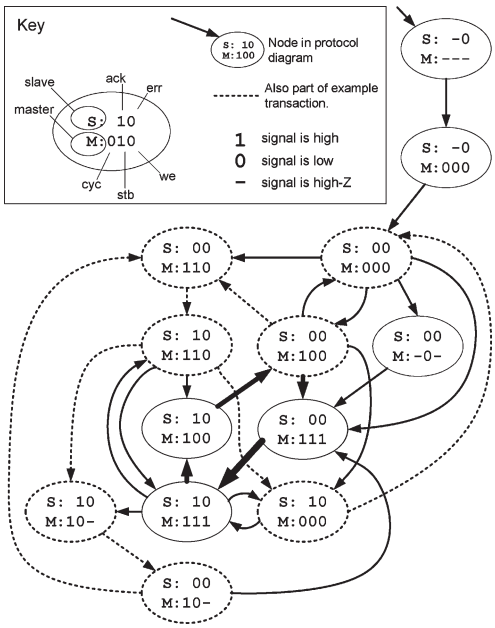

Fig. 6. Protocol diagram for Wishbone DMA. Each vertex represents a unique combination of interface signal values. Edges correspond to transitions from a combination to another. Vertices are labeled with the corresponding interface values. Dotted edges and vertices show correspondence to the transaction in Fig. 7. Note that S indicates signals from the slave, and M indicates signals from the master.

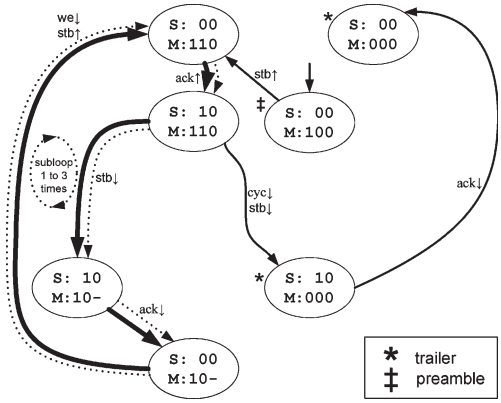

Fig. 7. Transaction diagram for a Wishbone DMA burst read. Each vertex corresponds to a distinct combination of values at the interface under study, and each edge corresponds to a transition from one set of values to another. The transaction diagram is a subgraph of the protocol diagram, and, in general terms, it corresponds to a single high-level type of activity. Dashed edges highlight the cycle of the read, which is repeated up to three times. The cycle is preceded by a preamble and followed by a trailer. Edges are labeled by the signal transitions that activate the corresponding transfer between vertices.

IV. Visualization, Assertions, and Optimizations

Protocol and transaction diagrams can sometimes become large, particularly when dealing with a complex design. To simplify the visualization of the diagrams presented by Inferno, we have added a number of graphical enhancements, including edge labels to show changing signals, line weights to show additional information, and dotted lines to show cycles. In addition, Inferno automatically generates assertions that specify approved transactions. These can be used in several traditional verification methodologies, as well as in our proposed closed-loop verification flow described in Section V. Finally, we extended our algorithm to handle very long simulation traces and concurrent analysis. We also optimized our algorithm to compress the number of signals that must be monitored by Inferno.

A. Visualizing Protocol and Transaction Diagrams

In presenting protocol and transaction diagrams, we strive to provide intuitive and compact graphs from which a user can quickly gather the relevant information. Each vertex of a diagram is labeled with the corresponding signal values observed at the interface under study. The signals are separated based on the signal direction across the interface. For instance, in the case of the Wishbone interface of Fig. 5, signals are partitioned between those driven by the master and those driven by the slave. Moreover, we label each edge with the signal changes that lead from the source vertex to the destination one. For instance, the transition from the preamble vertex of Fig. 7 to the *00110* vertex is due to the fourth signal going from *0* to *1*. Since the fourth interface signal corresponds to `stb`, we label the edge with `stb` rising. In addition, we indicate cycles with dashed lines along all the edges involved, and we indicate the range of repetitions observed for each cycle during the simulation.

We also show path usage information in each diagram using heavy lines to indicate frequently traveled paths. This technique, called statistical line weighting, offers a unique way to make large complex diagrams more readable. It offers the opportunity for quick concise qualitative analysis of the most important aspects of an interface or a program flow. For example, bugs are frequently represented by a sequence of vertices connected by thin lines, differing from normal

execution. Alternatively, if the user wishes to narrow down the subset of signals he is analyzing, selecting the signals causing many transitions is an effective method.

Edge weights are computed as the simulation trace is processed by Inferno. A tally t_i is kept for each signal transition. At the end, each tally is normalized with respect to the total; therefore, the weight associated with edge k is given by

$$\text{edge\_weight}_k = \frac{t_k}{\sum_{i=1}^{n} t_i}. \quad (1)$$

Finally, the weights are linearly fit for conversion to actual line weights in the protocol diagram representation. Statistical line weighting improves the readability of complex protocol diagrams because it automatically highlights the most common activity. Moreover, it hints at potential incorrect behavior since bugs are often related to activity that occurs infrequently, which is represented by thin lines in the diagrams. Figs. 6 and 7 show examples of statistical line weighting.

B. Automatic Assertion Generation

Inferno can automatically generate assertions corresponding to each transaction. Different engines can be linked to generate assertions in various languages, such as Verilog, Property Specification Language, or System Verilog Assertions. These assertions can then be deployed in a range of different verification methodologies to improve coverage and/or prove a design's functional properties.

1) In a *coverage-driven* verification methodology, the distinct transactions observed and the number of observations at a given interface provide a valuable metric to track the progress of verification.
2) 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 Fig. 8. Note that the known pool of transactions can be reused across multiple versions of the design to quickly reapprove the correctness of an interface. This closed-loop methodology is also directly implemented into Inferno and described in detail in Section V.
3) In the context of *formal verification*, assertions can be employed once the user believes that the full set of possible distinct transactions has been observed in simulation. The checker generated by Inferno can be used to prove that no other transaction can be generated at the interface under analysis or to expose undetected corner-case scenarios.

Transaction checkers are generated based on the set of vertices and edges of a transaction, forming the basis upon which the legal behavior of the system is established. For each vertex, we generate an expression corresponding to all legal outgoing edges from it; from here, we build the transaction checker as the disjunction of all these expressions. The initial checker expression can then be optimized by synthesis tools. In our experiments, we found that SIS [23] was sufficient to handle the complexity of the netlists generated; ABC [19] is also supported by Inferno. Alternatively, commercial synthesis tools such as Synopsys's Design Compiler can be used. If the design performs a transition not described by the checker's logic, the fail output signal is raised to flag a potential problem.

A checker accepting a number of separate transactions can easily be generated by combining the fail outputs of the individual checkers in a disjunction. This divide-and-conquer approach to describe a complex set of transactions has proven to be helpful in reducing checker complexity. We wanted to verify that no unapproved transaction was occurring in the Wishbone example of Section III-B. The checker describing all possible activity 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 able to not only verify that the checker was an invariant across the entire simulation but also observe which transactions were occurring. The checker generator can also be used to evaluate 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.

C. Optimizations

We implemented two optimizations to boost the performance of our algorithms in extracting protocols and transactions. To enable the processing of extremely long traces or simulations concurrently running with Inferno, we optimized our algorithm to incrementally read simulation traces. This allows Inferno to process arbitrarily long simulation traces, as well as to process a trace as it is being generated. The later benefit enables the new verification methodology we propose with this paper, discussed in Section V. Incremental trace processing introduces several modifications to the transaction extraction algorithm. Because the vertex sequence grows with every additional section read from the simulation trace, the sequence currently under consideration may terminate in the middle of a transaction. Thus, incomplete or spurious transactions may be found at the end of the trace section under study. Hence, Inferno discards the last portion of the trace file representing an incomplete transaction. Incomplete transactions are those transactions that do not terminate with one of the boundary IDs. Note that the dropped incomplete transaction is restored and completed when the next trace section is processed.

Intersignal correlation is an optimization that compresses the signals monitored by Inferno. Performed before the transaction-finding algorithm commences, intersignal correlation measures the relationships between pairs of signals to determine if they can be treated as a single entity. When two signals are found to match at every simulation cycle, or be always in an opposite phase, we can eliminate one of them and reduce the number of signals to analyze and store. If the correlation is sufficiently

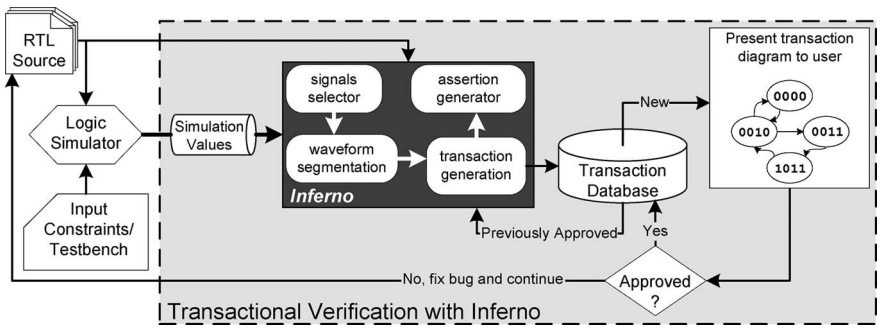

Fig. 8. Transactional verification methodology with Inferno. Inferno concurrently runs with a logic simulator and dynamically extracts relevant transactions. When a transaction is detected, it is first checked against a database of approved transactions. Only those transactions not found in the database are presented to the user to be further inspected against the specification.

close to either of these extreme values, signals may optionally be removed (with some loss of information) to simplify a large protocol diagram. Intersignal correlation reduces the complexity of a diagram while still accurately representing the fundamental behavior of the design.

V. Transactional Verification With Inferno

We developed a novel verification methodology based on Inferno's semantic inference capability. Our methodology relies on a closed-loop feedback flow that greatly reduces the large amount of redundant information presented to engineers in traditional simulation-based methodologies. The Inferno-based verification flow only presents to the user a new activity that has not been previously observed and could potentially indicate the presence of a bug.

Fig. 8 shows a schematic of our proposed verification flow. The methodology operates in the context of a logic simulation of a design, ideally a constraint-based random simulation. Inferno continuously monitors the signals in the target interface, extracting transactions as they occur in the simulation. When a new transaction is observed, it is visualized and presented to the user for approval, either manually or with the assistance of a specification aggregator such as [3]. At this point, an engineer can evaluate the transaction and determine whether it corresponds to a specified behavior for the system. Correct transactions are approved and included in Inferno's internal *transaction database*. Unapproved transactions trigger the end of the simulation to be further investigated by the engineering team for debugging, potentially leading to a design revision. Once a transaction becomes part of the transaction database, each subsequent occurrence is silently recognized by Inferno. Thus, the amount of user input required by this methodology is reduced greatly, focusing the engineer's attention only on potentially buggy situations.

In our experimental evaluation, we found that, in most cases, the frequency of new transactions is very high in the initial part of the simulation, tapering off over time. Section VIII-B and Fig. 14 show some of these trends for a few of our most complex designs.

High-quality intuitive transactions, i.e., compact transactions repeated many times, produce the best results when used in transactional verification with Inferno. However, even when transactions do not reflect the design intent in an intuitive form, transactional verification is still effective. Transactions always accurately reflect the system's observed behavior; thus, Inferno is still capable of matching simulation activity against the database of approved transactions and pinpointing the occurrence of new anomalous activity. Hence, while the process is most intuitive with transactions reflecting the design intent, it is still valuable and effective when this is not the case.

Transactional verification with Inferno allows the engineer to operate in the domain of high-level descriptions, more closely matching the intentions of a system designer. In this context, the glut of bit-level information generated by traditional methodologies, such as waveform viewers, is distilled to a small group of high-level transactions. Inferno uses these transactions to work with the user, learning about a design's behavior while simulation progresses, gradually taking on more responsibility in differentiating correct from flawed behavior.

VI. Limitations

While Inferno is effective in abstracting high-level behavior from simulated designs, there are some limitations in its approach. First, the selection of which signals to monitor is a manual process and has significant impact on the size and the quality of the transactions. Second, the testbench or workload used as a stimulus for the design when generating the trace is a factor affecting the diversity and the completeness of the extracted transactions. Additionally, designs with overlapping transactions due to delays on some signals on the interface or no clear boundary state can result in low-quality transactions.

When configuring Inferno, signal selection is essential to extract high-quality transactions. Selecting too many signals leads to large unintuitive transactions, which typically do not reflect high-level behavior. On the other hand, selecting too few signals will often fail to capture all the behavior necessary to form a meaningful transaction. Consider the Universal Serial Bus (USB) physical layer design from Section VIII. When the entire interface (18 signals) is selected, 94 transactions are found with a relatively low number of repetitions (7017). These transactions are difficult to understand, and it is not apparent that they correspond to a designer's idea of high-level behavior. Upon closer inspection, the top-level signal interface contains two subsets of signals—the transceiver and the USB 2.0 Transceiver Macrocell Interface. When Inferno was separately directed at these interfaces, seven and six transactions were

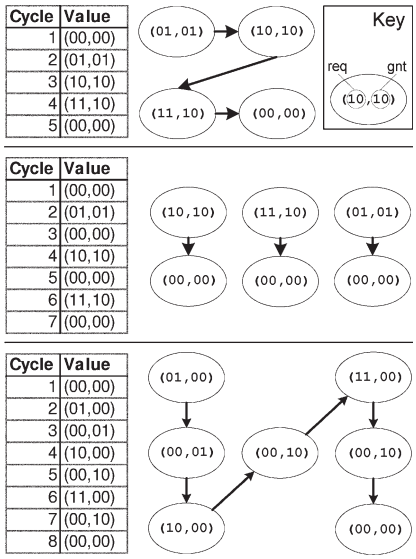

Fig. 9. Priority selector example demonstrating overlapping transactions. The first version is a single cycle design and results in a single monolithic transaction. The second version reflects an idle state between requests, resulting in a clean set of transactions. Finally, the last version has an output delay of one cycle, once again resulting in a single monolithic transaction.

discovered with a total of 88 596 repetitions. Additionally, the transactions were compact and easy to understand, highlighting the importance of signal selection.

The number of transactions discovered by Inferno is strongly affected by the length and the strength of the stimulus provided to the design while generating a trace. Long running workloads generally result in a broader set of transactions, up until the point where all transactions have been discovered. For example, the Z80 design in Section VIII ran for 389 771 cycles, resulting in 11 transactions. With a shorter testbench of 28 870 cycles, only two transactions are found.

Lack of a clear boundary state can also result in poor quality transactions. This is illustrated through an example of a simple two-bit priority selector, whose simulation trace is shown in Fig. 9. The selector grants access on the same cycle as it receives a request, without returning to an idle state. Inferno produces one monolithic transaction representing the entire trace. A modified version of the priority selector, which returns to the idle state between requests, results in a clean set of transactions, one for each operation of the priority selector.

Another limitation of Inferno is exposed by interfaces with overlapping transactions due to delays on a subset of signals on the interface. Since transactions are marked by unique signal combinations, delayed value changes on a subset of the signals under consideration can sometimes result in erroneous transaction boundary identification. To illustrate this issue, we consider again the priority selector, but we now modify it to use a two-cycle computation; requests are granted after a delay of one cycle. As shown in Fig. 9, the output delay causes an overlap between input and output responses, and the result is again a single monolithic transaction. In this case, the problem can easily be solved by using the grant signals as the boundary identifiers.

Some of Inferno's discussed limitations can be circumvented by means of manual transaction boundary specification. This option allows the user to directly specify a signal that governs the transaction boundary. In practice, selecting control signals from the client side is usually effective since clients tend to handle one transaction at a time, while servers are multitasking.

VII. Applications and Case Studies

We first examine Inferno's capabilities with two case studies on complex processor designs. Then, we show Inferno's effectiveness in a related domain—equivalence checking.

A. Case Study: Dual-Core Alpha

We discussed in Section IV-B a number of verification methodologies where Inferno can be deployed effectively. In the context of design development, situations may arise when a well-defined specification of the interface protocol is missing. In this scenario, Inferno can be a 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 is 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 valuable 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. Over the course of a single test, we were able to use Inferno for direct bug-finding (i.e., identifying bugs simply by examining the diagram) and to check the equivalence of two distinct implementations of the same module.

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 level-one (L1) instruction and data caches, and they shared a level-two (L2) cache using the MESI protocol [9]. 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, whereas the cores could operate at lower speed, the interface between the L1 and L2 caches was asynchronous. The initial division of labor vastly underestimated the complexity 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 continued to fail 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

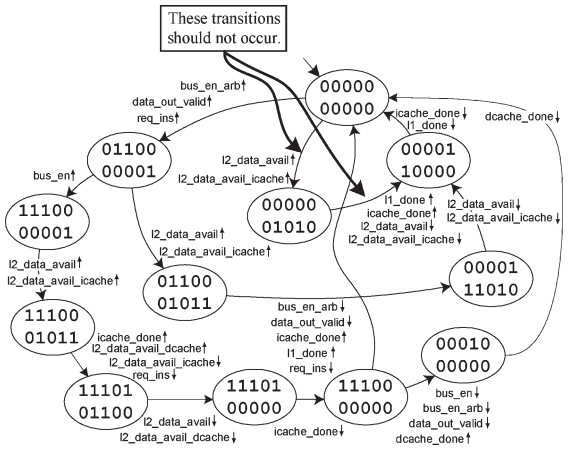

Fig. 10. Protocol diagram for the cache arbiter interface in the dual-core Alpha processor. Notice that the labeled edge shows the interface leaving the idle state and entering a configuration where the arbiter signals data ready from L2 cache. This is clearly a bug since no one has requested any data, and yet they are there. This case study presents a situation where direct inspection of a simple protocol diagram yields insightful understanding and debugging support.

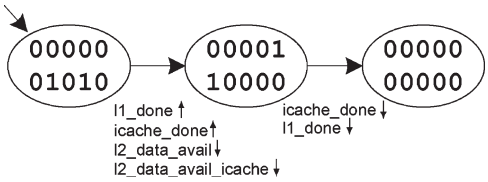

Fig. 11. Transaction diagram showing the bug in the cache arbiter interface. This transaction correlates with the buggy states and erroneous transitions noted in the protocol diagram (Fig. 10). It occurred only once in the trace, raising suspicion of abnormal behavior.

as the one shown in Fig. 10, 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. The bug was found in a transaction that occurred only once (Fig. 11), a clue indicating abnormal behavior. As indicated in Fig. 10, the bug also appears in the protocol diagram through transitions, revealing that data are ready from the L2 cache, with no preceding request. The most naive 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 transitions are missing.

This case study also exposed the benefit of Inferno in evaluating the quality of a testbench suite. We set up an experiment that 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, sharing only one transaction. Further analysis showed that this was because the two cores were executing the same program on the same data; hence, the one that accessed the L2 cache first would always be the first to request new data, whereas the other would always find the cached values ready. Consequently, the transactions

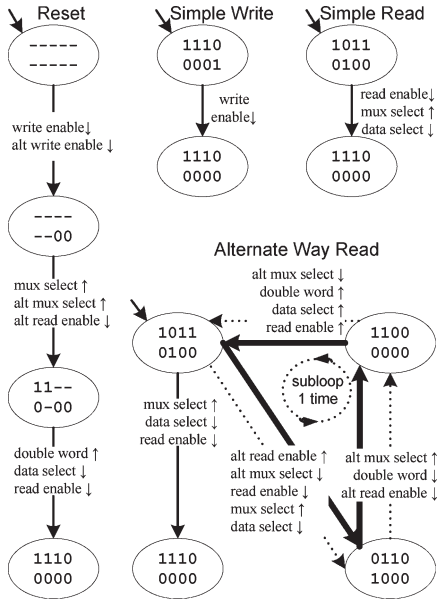

Fig. 12. Transaction diagrams from OpenSPARC T1 data cache interface, representing a simple read, a simple write, and an alternate way read. Arrows next to the signal names denote rise and falling edges.

were very different at the two interfaces: One would have all misses, and the other would have all hits. This result was the key in driving the development of a more complete test suite, leading to diversified coverage on both instances.

B. Case Study: OpenSPARC T1

In this case study, we demonstrate the capabilities of Inferno on an industrial microprocessor design, i.e., OpenSPARC T1 [24]. Sun Microsystems' OpenSPARC T1 is a complex processor design, which encompasses half a million lines of code. The system consists of eight Scalable Processor Architecture cores, each capable of executing four simultaneous threads. OpenSPARC T1's memory subsystem consists of L1 data and instruction caches at each core connected to a unified L2 cache and four main memory controllers.

We studied the behavior of the control signals of the L1 data cache interface with Inferno. The data source was a full system simulation running the regression suite provided with the OpenSPARC release. We quickly obtained a set of four distinct transactions represented in Fig. 12. Upon inspection of OpenSPARC's specification manual, we determined that these precisely represented all the data cache interface's valid modes of operation. Note how each transaction in Fig. 12 terminates with the same vertex, which is precisely the boundary vertex ID in this analysis.

In Fig. 13, we plot the corresponding protocol diagram, where we indicate the correspondence between each transaction and its subgraph in the protocol diagram. In addition, our visualization techniques for protocol diagrams make it immediately apparent that the simple read operation is by far the most common activity. In fact, the two thickest edges in the diagram of Fig. 13 correspond to the edge starting a simple read transaction and the edge internal to this transaction.

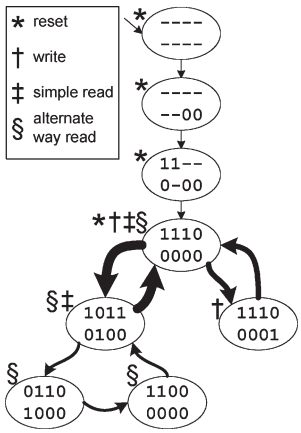

Fig. 13. Protocol diagram of the OpenSPARC T1 data cache interface. Dashes indicate the high-Z state, occurring only in the reset sequence. Statistical line weighting on this graph indicates the most commonly exercised path, i.e., a common read, shown by heavy lines.

Our case study of OpenSPARC T1 demonstrates that Inferno is capable of tackling large industrial-scale designs. We could extract the functional activity of the L1 data cache interface with Inferno's high degree of automation. The abstraction made possible by transactions made the interface easy to understand, and the concurrent simulation processing allowed us to monitor results without waiting for a lengthy tedious simulation to complete.

C. Equivalence Checking

Inferno can also be used in applications related to sequential equivalence checking. In this context, we take two distinct implementations of the same design and select a number of corresponding interfaces of interest. Often, this type of application would consider the same design before and after synthesis, or synthesis optimizations or modifications. Hence, it is commonly fairly straightforward to identify a set of interfaces at the boundary of the region that has been undergoing modification. Once the proper set of interfaces has been selected, Inferno can be applied to both versions of the design under study to extract the semantic behavior of the system at this boundary. Any difference among the corresponding transactions identified is a leading indicator of a mismatch between the two implementations. On the other hand, since the approach is not complete, when the transactions match, nothing can be said about the equivalence of the systems. This approach resembles some similarity to the technique that Ammons *et al.* [2] applied to the specifications of the API.

A related application to this is that of different modules or designs, which are intended to follow the same protocol. Once again, Inferno can be deployed to detect high-level mismatches in the high-level interface semantic. An example of this use was provided in the case study of the dual-core Alpha design discussed in Section VII-A.

VIII. Experimental Results

We evaluated Inferno on a set of RTL designs ranging in size from small decoders to an industrial chip multiprocessor design, i.e., the OpenSPARC T1. Designs were obtained from OpenCores [21] and from the Sun OpenSPARC project [24]; the dual-core Alpha was designed by a student team.

TABLE I
Designs for Experimental Evaluation. The Number of Nets at the Target Interface Excludes Clock and Reset

Testbench Design - *Interface*	Gates	FF	I/f Nets	Simulated Cycles
Dual-Core Alpha - *Cache Arb*	12,419	1,899	10	930
DMA - *I/O*	15,864	423	5	12,638,676
Single Clock FIFO - *I/O*	4,738	2,091	5	1,305,624
Dual Clock FIFO - *I/O*	8,737	4,210	5	53,706,289
OpenSPARC T1	517,637	123,757		
- *Cache Crossbar*			32	1,191,484
- *L1 Data Cache A*			8	16,218
- *L1 Data Cache B*			8	137,719
PCI	7,536	2,080		11,295,138
- *Arbitration*			12	
- *Wishbone Slave*			7	
- *Wishbone Master*			6	
RS Decoder - *I/O*	2,763	447	6	50,136
Simple SPI - *I/O*	1,257	229	5	48,175
USB	718	192		28,464,544
- *Core TX/RX*			7	
- *PHY, All I/O*			18	
- *PHY Transceiver*			8	
- *PHY UTMI*			10	
Z80 uP - *SRAM*	3,723	2,056	3	389,771

Table I characterizes each of the designs used to evaluate Inferno. The target interface to be analyzed by Inferno was selected manually; in most cases, this interface was easily accessible as a subset of the top-level module's I/O. We simulated each design with a logic simulator (Synopsys VCS) using the provided testbenches as the stimulus to reach cycle counts ranging from 930 cycles to almost 54 million cycles. The simulator was configured to generate a trace limited to the signals of the interface of interest.

Table I shows the key characteristics of our testbenches. The first column reports the design name and the interface under analysis (italicized). Gate and flip-flop counts are shown in the next two columns; these values correspond to the complete system. Next, the number of nets in the interface and the total simulated cycles are reported. Gate and flip-flop counts are obtained by synthesizing each design with Synopsys Design Compiler (GTECH is the target library). OpenSPARC gate and flip-flop counts exclude caches. Additionally, for the OpenSPARC T1, we used three different simulations: one for the cache crossbar, which exercised all eight cores (`chip8_mini` regression suite), and two single-core regression tests for the L1 data cache (`bypass_win` and `lsu_mbar`).

A. Extraction of Interface Semantics With Transactions

We ran the simulations described in the previous section, and as a first experiment, we extracted the protocol diagram and transactions for each of the interfaces under evaluation. Table II reports for each interface the total number of unique vertices observed. This number represents the number of vertices in the protocol diagram, as well as the aggregate of all unique vertices in the set of transaction diagrams. Furthermore, it also indicates the number of unique signal configurations observed at the interface over the entire simulation. The next two columns report the number of transactions that Inferno extracted and the cumulative number of occurrences of these transactions during the simulation.

TABLE II
SIZE OF PROTOCOL AND TRANSACTION DIAGRAMS. TRANSACTION REPETITIONS IS THE CUMULATIVE NUMBER OF REPETITIONS OVER ALL TRANSACTIONS

Testbench	Total Distinct Vertices	Total Distinct Transactions	Total Transaction Repetitions
Dual-Core Alpha - *Cache Arb*	13	6	43
DMA - *I/O*	13	8	142,227
Single Clock FIFO - *I/O*	8	6	2,011
Dual Clock FIFO - *I/O*	9	16	579,815
OpenSPARC T1			
- *Cache Crossbar*	867	2,946	29,831
- *L1 Data Cache A*	8	4	38
- *L1 Data Cache B*	10	11	3,839
PCI			
- *Arbitration*	226	1,097	4,246
- *Wishbone Slave*	22	62	15,297
- *Wishbone Master*	6	4	2,128
RS Decoder - *I/O*	6	6	3,008
Simple SPI - *I/O*	6	6	2,605
USB			
- *Core TX/RX*	13	7	20,733
- *PHY, All I/O*	32	94	7,017
- *PHY Transceiver*	10	7	83,610
- *PHY UTMI*	7	6	4,986
Z80 uP - *SRAM*	5	11	33,237

The number of distinct transactions observed is a linear indicator of the memory required to store these transactions, which clearly represents a significant reduction of the long simulation traces from which the transactions are derived. The cache crossbar testbench has a large number of distinct transactions because it implements the communication among the I/O pairs of all combinations of eight cores and four memory controllers. The Peripheral Component Interface (PCI) arbitration also had a large number of transactions due to an intensive testbench on a complex protocol. Apart from these two interfaces, all the other designs had a very small number of distinct transactions, occurring up to half a million times over the course of the simulation. In the context of our transactional verification methodology, repetitions are an indicator of a transaction's quality. Those transactions that exhibit a high number of repetitions result in a long period of automatic operation once they have been approved and added to the Inferno database. The dual-core Alpha and one of the OpenSPARC experiments had a low repetition in the number of transactions, but this was the result of a short simulation trace.

Table III provides an analysis of Inferno's performance. We ran Inferno in the context of our transactional verification methodology, accumulating the total execution time of Inferno during the course of the experiment. The waiting time for user approval of each new transaction was excluded, as well as the time spent in the simulation, which is reported separately. The table compares Inferno's runtime in seconds against that of the logic simulator. It is evident from the table that in all but the smallest experiments, Inferno constitutes an acceptable fraction (32% on average) of the simulation time. Hence, our ability to extract semantic information does not hamper the performance of the verification effort and can be completely folded into the time spent in simulating the design. This is particularly true for complex designs where simulation progress is slow, but Inferno's runtime remains unaffected. In conclusion, Inferno's performance is more than sufficient to enable and make practical the transactional verification methodology we propose.

TABLE III
INFERNO PERFORMANCE. THE TABLE COMPARES THE SIMULATION RUNTIME AGAINST INFERNO'S EXECUTION TIME ON THE CORRESPONDING TRACE

Testbench	Inferno runtime (s)	Simulation runtime (s)
Dual-Core Alpha - *Cache Arb*	<1	<1
DMA - *I/O*	107	890
Single Clock FIFO - *I/O*	26	26
Dual Clock FIFO - *I/O*	831	2,628
OpenSPARC T1		
- *Cache Crossbar*	134	3,344
- *L1 Data Cache A*	1	24
- *L1 Data Cache B*	3	159
PCI		
- *Arbitration*	259	901
- *Wishbone Slave*	202	901
- *Wishbone Master*	188	901
RS Decoder - *I/O*	<1	6
Simple SPI - *I/O*	2	8
USB		
- *Core TX/RX*	290	605
- *PHY, All I/O*	405	605
- *PHY Transceiver*	355	605
- *PHY UTMI*	346	605
Z80 uP - *SRAM*	5	17

TABLE IV
TIME AT WHICH THAT LAST NEW TRANSACTION IS DISCOVERED. AFTER THIS TIME, INFERNO CONTINUOUSLY MONITORS TRANSACTIONS FOR CORRECTNESS WITHOUT USER INPUT USING THE TRANSACTION DATABASE TO MATCH VALID TRANSACTIONS. ALSO SHOWN IS THE SIZE OF THE ASSERTIONS AUTOMATICALLY GENERATED BY INFERNO. THE NUMBER OF ASSERTIONS AS WELL AS THE AVERAGE NUMBER OF TERMS IN EACH ASSERTION ARE SHOWN

Testbench	Total Cycles	Time of last new transaction (cycles)	Assertions	
			Terms	Literals (Avg)
Dual-Core Alpha - *Cache Arb*	930	157	13	1.6
DMA - *I/O*	12,638,676	3,953,602	13	5.2
Single Clock FIFO - *I/O*	1,305,624	282	8	3.4
Dual Clock FIFO - *I/O*	53,706,289	627,746	9	3.6
OpenSPARC T1				
- *Cache Crossbar*	1,191,484	454,376	867	2.0
- *L1 Data Cache A*	16,218	8,987	8	1.4
- *L1 Data Cache B*	137,719	54,124	10	1.7
PCI	1,295,138			
- *Arbitration*		29,305	226	8.2
- *Wishbone Slave*		184,099	22	4.8
- *Wishbone Master*		328,615	6	6.5
RS Decoder - *I/O*	50,136	155	6	5.4
Simple SPI - *I/O*	48,175	568	6	2.9
USB	28,464,544			
- *Core TX/RX*		2,258	13	1.6
- *PHY, All I/O*		35,017	32	7.3
- *PHY Transceiver*		21,705	10	5.2
- *PHY UTMI*		1,517,985	7	6.0
Z80 uP - *SRAM*	389,771	14,237	5	1.7

B. Transactional Verification Methodology

We now evaluate the transactional verification methodology enabled by Inferno in terms of the demands it poses on engineering time and its ability to learn a design's valid behavior. To this end, we ran Inferno with our transactional verification flow, shown in Fig. 8 and as described in the previous section. For each design, we collected information on the temporal relation of unique transactions. The simulation time in cycles was recorded every time a new transaction (one that would be displayed to the user) was observed.

Table IV reports the time at which the last new transaction occurred during each design simulation. By comparing this

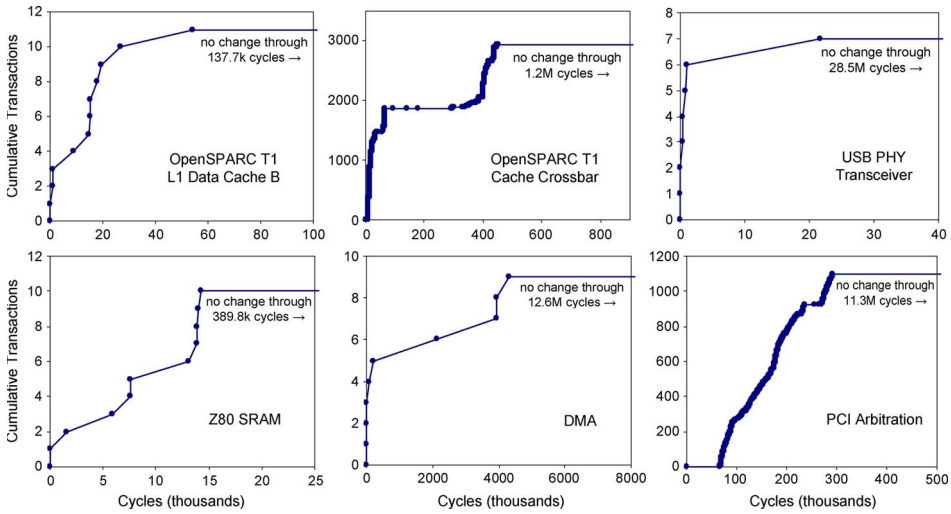

Fig. 14. Transaction discovery during simulation. The plots show the time of discovery of each unique transaction (indicated by a dot) during the simulation for six of our most complex designs. Note how in all cases the full set of transactions was discovered early in the simulation.

to the full length of the simulation trace (reported in the Total Cycles column), we ascertain the point at which Inferno has learned the entirety of the design semantic exposed by the simulation and can then proceed without assistance. On average, the entire set of transactions appeared within the first 14% of the simulation time. For example, Inferno was able to extract all unique transactions from the Dual Clock FIFO simulation within 63 000 simulation cycles, about 1% of the total simulation time. After this point, the simulation continued for an additional 53 million cycles, while Inferno silently checked transactions without user attention. Designs that required significantly more time to discover the last transaction were the OpenSPARC and DMA designs. In the case of OpenSPARC, the test programs used to obtain the traces were short and directed, their primary goal being to test specific hardware functionality. For the DMA design, the last transaction appeared about a third of the way through the simulation, and we discovered it to be part of a cluster of new transactions, indicating a new phase of testbench operation during which new functionality of the design was exercised.

In Fig. 14, we plotted the first discovery of each transaction for six of our largest designs during the experimental simulation. Every design exhibits phases of active transaction discovery, and all reach a point early in the simulation after which no new transactions are found. Some designs exhibit bursts of new transactions well after the initial set has been discovered. These bursts correspond to a new phase of operation by the testbench, stimulating different aspects of the design. In all cases, the cumulative transaction count quickly approaches a constant.

Our experiments show that, in the context of our proposed transactional verification methodology, Inferno is capable of quickly learning the transactions that describe a design's valid behavior. User interaction is only required early in the simulation, as many new transactions appear while the design is yet largely unexplored. As additional transactions are discovered and the design becomes more well known, Inferno takes on more of the responsibility of differentiating transactions, completely taking over once it has learned the set of approved transactions.

C. Automatic Assertions

We also evaluated the size and the feasibility of Inferno's automatically generated assertions. Inferno was configured to generate a single assertion in Verilog HDL that accepts all extracted transactions. This assertion is a disjunction of terms, where each term corresponds to a unique transaction. Each term representing a transaction comprises several literals, and each literal refers to the switching of a signal in the transaction diagram. In Table IV, we show the number of terms that are required to fully specify the assertion for each target interface, as well as the average number of literals in each term.

IX. Conclusion

We have presented Inferno, a software tool that operates on a logic simulation trace and automatically extracts transactions, i.e., high-level descriptions of a design's behavior. Transactions are presented to the user through simple and intuitive diagrams for which we have developed a number of specialized visualization enhancements. Complex repetitive design simulations are distilled to a compact set of transactions, which describe the semantic behavior of the system in a compact high-level format. In addition, Inferno can automatically generate assertions corresponding to extracted transactions, which can be used in a number of traditional verification methodologies, ranging from assertion-based verification to those using formal and semiformal tools.

We have proposed a new verification methodology enabled by Inferno, called transactional verification, which greatly reduces the verification effort required to determine the correct behavior of a system. Our methodology is based on a closed-loop approach, where transactions are extracted from a simulation and displayed to the user for approval. As the simulation progresses, Inferno learns from the approved transactions, requiring user input only for new behavior.

We have demonstrated Inferno's usefulness through several case studies on complex hardware designs. Additionally, we have provided a broad set of experimental evaluations on a

wide range of aspects and designs, demonstrating how Inferno is at once practical and effective at extracting high-level design behavior from low-level lengthy simulation data.

We are currently investigating additional techniques to determine transaction boundaries that directly address the limitations described in Section VI. Furthermore, recent work in the artificial intelligence community has developed new models of learnability based on statistical sampling, such as the probably approximately correct model [16], [20]. While we do not rely on statistical learning in this paper, this direction could constitute a valuable future research direction.

REFERENCES

[1] M. Acharya, "Automatic generation and inference of interface properties from program source code," in *Proc. Companion 21st ACM SIGPLAN Symp. Object-Oriented Program. Syst., Languages, Appl.*, 2006, pp. 750–751.

[2] G. Ammons, R. Bodik, and J. Larus, "Mining specifications," in *Proc. POPL*, 2002, pp. 4–16.

[3] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, "Debugging temporal specifications with concept analysis," in *Proc. PLDI*, 2003, pp. 182–195, pp. 182–195.

[4] T. Arts and L. Fredlund, "Trace analysis of Erlang programs," *ACM SIGPLAN Notices*, vol. 37, no. 12, pp. 18–24, Dec. 2002.

[5] S. Bensalem, Y. Lakhnech, and H. Sadi, "Powerful techniques for the automatic generation of invariants," in *Proc. CAV*, 1996, pp. 323–335.

[6] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. Comput.*, vol. C-21, no. 6, pp. 592–597, Jun. 1972.

[7] D. Brahme, S. Cox, J. Gallo, W. Grundmann, C. Ip, W. Paulsen, J. Pierce, J. Rose, D. Shea, and K. Whiting, "The transaction-based verification methodology," Cadence Design Syst., Inc., San Jose, CA, Tech. Rep. No. CDNL-TR-2000-0825, Aug. 2000.

[8] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proc. MICRO*, 2003, pp. 129–140.

[9] D. Culler, A. Gupta, and J. Singh, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan Kaufmann, 1997.

[10] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Proc. SOSP*, 2001, pp. 57–72.

[11] M. Ernst, "Verification for legacy programs," in *1st IFIP TC 2/WG 2.3 Conf., VSTTE 2005*, vol. 4171, *Lecture Notes in Computer Science*, B. Meyer and J. Woodcock, Eds., Zurich, Switzerland, Oct. 10–13, 2005, 546 p.

[12] G. Fey and R. Drechsler, "Improving simulation-based verification by means of formal methods," in *Proc. ASPDAC*, 2004, pp. 640–643.

[13] B. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via critical-path prediction," in *Proc. ISCA*, 2001, pp. 74–85.

[14] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proc. IEEE Symp. Security Privacy*, 1996, pp. 120–128.

[15] E. M. Gold, "Language identification in the limit," *Inf. Control*, vol. 10, no. 5, pp. 447–474, 1967.

[16] O. Guttman, S. Vishwanathan, and R. C. Williamson, *Learnability of Probabilistic Automata via Oracles*, vol. 3734, *Lecture Notes in Computer Science*. Berlin, Germany: Springer-Verlag, 2005.

[17] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proc. DAC*, 2005, pp. 775–778.

[18] B. Isaksen and V. Bertacco, "Verification through the principle of least astonishment," in *Proc. ICCAD*, 2006, pp. 860–867.

[19] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Proc. DAC*, 2006, pp. 532–535.

[20] K. P. Murphy, "Passively learning finite automata," Santa Fe Inst., Santa Fe, NM, Tech. Rep. No. 96-04-017, 1996.

[21] Opencores. [Online]. Available: http://www.opencores.org/

[22] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Automatic generation of complex properties for hardware designs," in *Proc. DATE*, 2008, pp. 545–548.

[23] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," Univ. California, Berkeley, Berkeley, CA, Tech. Rep. No. UCB/ERL M92/41, 1992. Tech. Rep.

[24] *Sun Microsystems OpenSPARC*. [Online]. Available: http://opensparc.net/

[25] C. Weaver, K. Barr, E. Marsman, D. Ernst, and T. Austin, "Performance analysis using pipeline visualization," in *Proc. ISPASS*, 2001, pp. 18–21.

[26] J. Yang and D. Evans, "Automatically inferring temporal properties for program evolution," in *Proc. ISSRE*, 2004, pp. 340–351.

Andrew DeOrio (S'07) received the M.S.E. degree in electrical engineering in 2008 from the University of Michigan, Ann Arbor, where he is currently working toward the Ph.D. degree, working in the Advanced Computer Architecture Laboratory of the Computer Science and Engineering Department.

His research interests include reliable system design, hardware verification, and postsilicon validation.

Adam B. Bauserman (M'07) received the B.S.E. and M.S.E. degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 2006 and 2007, respectively.

He was a Research Assistant with the Advanced Computer Architecture Laboratory, University of Michigan, where he worked on verification tools and fault-tolerant systems. He is currently a Hardware Engineer with NVIDIA, Santa Clara, CA. He is currently involved in the design and verification of high-speed memory interfaces for graphics processors.

Valeria Bertacco (S'95–M'03) received the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1998 and 2003, respectively, and the Laurea degree (*summa cum laude*) in computer engineering from the University of Padova, Padova, Italy.

She was a Staff Research Engineer for four years with the Advanced Technology Group, Synopsys. She is currently an Assistant Professor of electrical engineering and computer science (EECS) with the University of Michigan, Ann Arbor. Her research interests include the areas of formal and semiformal design verification, with emphasis on full design validation and digital system reliability.

Prof. Valeria is an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. She has served on the program committees for the Design Automation Conference and the International Conference on Computer-Aided Design. She was a recipient of the National Science Foundation CAREER Award and the Air Force Office of Scientific Research's Young Investigator Award.

Beth C. Isaksen (M'06) received the B.A. degree in mathematics from Carleton College, Northfield, MN, in 2000, the M.S.E.E. degree from the University of Notre Dame, Notre Dame, IN, in 2003, and the Ph.D. degree from the University of Michigan, Ann Arbor.

She is currently a Software Engineer with Jasper Design Automation, Mountain View, CA, where she is part of a team developing advanced formal verification tools.