Automatic Generation of Complex Properties for Hardware Designs*

Frank Rogin‡ Thomas Klotz‡ Görschwin Fey** Rolf Drechsler** Steffen Rülke‡

‡ Fraunhofer Institute for Integrated Circuits, Division Design Automation, 01069 Dresden, Germany
{frank.rogin, thomas.klotz, steffen.ruelke}@eas.iis.fraunhofer.de

** University of Bremen, Institute of Computer Science, 28359 Bremen, Germany
{fey, drechsle}@informatik.uni-bremen.de

Abstract

Property checking is a promising approach to prove the correctness of today's complex designs. However, in practice this requires the formulation of formal properties which is a time consuming and non-trivial task. Therefore the acceptance and efficiency of formal verification techniques can be raised by an automated support for formulating design properties. In this paper we propose a new methodology to automatically generate complex properties for a given design. The tool, Dianosis, implements this methodology by analyzing a simulation trace. The extracted properties describe the abstract design behavior and are presented in a format that is easy to read and can be added to the set of properties used for formal or assertion-based verification. We provide experimental results on industrial hardware designs that show the effectiveness of Dianosis and motivate the practical use.

1. Introduction

The conventional simulation-centered verification methodology is insufficient to meet today's requirements, such as an exhaustive verification of complex designs, or the demand for first-time-right designs. As verification has become the dominating factor in circuit and system design, huge efforts have been made in the past to increase the productivity and quality of the verification process. New formal and semi-formal verification techniques were proposed over the past few years and some of them are now part of daily use in industrial design flows (e.g. equivalence checking). Furthermore, verification techniques such as property checking or assertion-based verification are getting increased application in the current industry.

All these techniques, however, require a formal specification of the design. For that, properties have to be defined manually which is an error-prone and non-trivial task. They are normally derived from a high-level specification written in natural language that may contain inconsistent, erroneous, or incomplete requirements. The declarative nature of property languages often also hampers a widespread application of formal techniques. Furthermore, complex properties expressing the inter-module interaction are hard to write. The increasing design complexity and a distributed development process additionally complicate this task. So, any method to reduce the effort of writing properties and in particular to abstract the design behavior would be beneficial. Most especially, this could help to increase the acceptance of formal techniques.

In this paper we introduce a new approach that is inspired by prior work on automatic generation of properties for software programs [4, 7] and hardware designs [2, 3, 6]. Daikon [7] and IODINE [3] report likely invariants searching a limited catalog of preset properties. In contrast our approach is not restricted to such a predefined property set. Instead, we use already validated properties to derive more complex ones that help the verification engineer to obtain more abstract design information. The further class of tools infer formerly unknown properties. The authors in [6] propose a method to extract common design behavior by means of transaction activity at any user-defined interface of a hardware design. A disadvantage of this approach is the limitation to control signals to gain usable information. In contrast, our solution generates properties over all possible signals. Anomaly detection is performed by a further tool called DIDUCE [4] that only reports violations of relatively simple program invariants in Java programs. Pattern matching is used in [2] to generate arbitrary properties over a selected set of signals. Compared to our solution, a size-limited time window has to be defined by the user which prevents an efficient search for long running properties.

Unlike all stated work our approach aims at the generation of complex properties that gain a better insight into the abstract design behavior. The properties are inferred over a given simulation trace and all its containing signals. In the first phase, a set of predefined properties is hypothesized over the design behavior leaving only such property candidates that are valid on the trace. During the subsequent second phase the found properties are combined to new more complex candidates and are checked on the sim-

*Partial funding provided by the URANOS project, BMBF-01M3075.

ulation trace. Surviving property candidates are recombined until no more valid properties can be created. The extracted functional behavior is presented to the verification engineer for manual inspection. Our solution has a number of advantages: (i) improved design understanding – properties provide an abstraction from the implementation, (ii) tool support for the formulation of properties – derived properties are a starting point for formal verification, (iii) identification of holes in the test suite – a derived property that cannot be proven unveils behavior not exercised by the test suite, and (iv) enhanced efficiency of the overall verification process – the application of our tool *Dianosis* in the verification flow speeds up the process and improves its quality.

2. Generation methodology

Figure 1 shows our property generation algorithm that is divided into two phases. During the first phase predefined basic properties are inferred over all signals of the given design and their correctness is validated on the simulation trace. To obtain high-quality properties it is necessary that the design is extensively simulated using a testbench with a high functional coverage. Otherwise, a large number of properties are incorrectly inferred and does not hold on the design. All properties that are not falsified during analysis (cp. Phase 1 in Figure 1) are used in a subsequent second phase that checks the temporal dependencies between them. If such a dependency is valid on the given trace, this new context is formulated as a complex property and is recorded into a database. Repeating this procedure, temporal dependencies between the already found and the newly generated properties are examined. Thus, even more complex property candidates are created and checked on the simulation trace (cp. Phase 2 in Figure 1). The second phase ends when no new properties are found. That means P_{new} remains empty.

Since property generation based on simulation traces cannot be "all-embracing" by definition, incomplete or wrong behavior could be extracted. So, the verification engineer is always required to check the correctness of the derived properties with respect to the design intent and the specification. Hence, user interaction is required here.

3. Dianosis property generation

Based on the algorithm described in Section 2 we developed a tool called *Dianosis* (Dynamic Invariant Analysis on Simulation Traces). *Dianosis* handles simulation traces provided in the industry standard VCD format [5]. Thus, properties can be generated independently of the used simulator tool or the modeled design level. Different formats suitable for advanced verification tasks (e.g. SystemVerilog Assertions) can be chosen to output the derived properties.

To generate valuable properties, a large basis of basic properties that can be combined is essential. The Open Verification Library (OVL) [1] provides a set of 51 parametrizable assertion-based checkers that ensure typical basic behavior in hardware designs. Due to this generality we selected a couple of OVL assertion checkers and implemented proper search algorithms in terms of checkers in *Dianosis*. Beyond this, a number of custom-made checker types are realized (see Table 1).

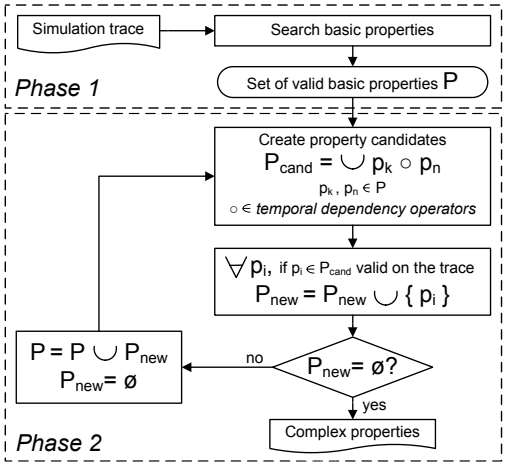

Figure 1. Dianosis general algorithm

Table 1. Selection of basic property checkers

Checker	Description
OVL_Increment	Analyze increment counters.
OVL_OneHot	Identify one-hot coded busses.
OVL_Handshake	Identify signal pairs that follow the handshake pattern.
Req-N-Grant	Identify a signal changing its state whenever a second signal is active.

Extract basic properties. After reading the simulation data and performing a parallel check for some properties such as constant signals, *Dianosis* initiates the process of property generation. Initially, for the current module all scalar and multi-bit signals are determined. Based on this signal list property candidates for all kinds of signals and their combinations are created over all selected checker types. To increase the chance for inferring reasonable properties and to accelerate the search, the creation of candidates is subject to several restrictions:

- Discard reflexive binary candidates.
- Treat only one combination of a symmetric property.
- Disallow candidates over signals in the exclusion list.

During an update phase the signal values at the current time stamp are passed to each appropriate checker instance. There, a checker is only invoked when the observed signal values changed which speeds up its execution. After that, the simulation time is shifted to the next value change. Each checker type is individually implemented and comprises depending on the complexity of the analyzed property up to many hundreds of lines of C++ code.

```
(0)  // initialization: mutex_start = −1
(1)  update(v_a[t], v_b[t]) begin
(2)    if (v_a[t] = active and v_b[t] = active)
(3)      return false      // invalid at time t
(4)    if (mutex_start = −1)
(5)      if (v_a[t] = active or v_b[t] = active)
(6)        mutex_start = t
(7)    else if (v_a[t] = inactive and v_b[t] = inactive)
(8)      mutex_start = −1
(9)    return true       // valid at time t
(10) end
```

Figure 2. OVL_Mutex checker

Example 1 *Figure 2 exemplifies the test of the mutual exclusive activation of two signals a, b in terms of their values $v_a[t]$ and $v_b[t]$ at each time stamp t. If the mutex behavior is falsified, the* update*-function returns* false *(line 3). Thereupon, the property candidate is removed from the candidate set. If only one of the two signals is activated, the checker has recognized a new mutual exclusion activity (line 5) which does not end until both signals are inactive again (line 8).*

The generation process is finished when no more candidates are left or the end of the simulation trace is reached. A candidate that has "survived" over the complete trace describes a valid basic property and is stored into the property database (see Figure 3 (a)).

Transaction encoding. After the search for basic properties, the signal behavior of each binary property is composed into transactions. A transaction describes the property activity in terms of the activity of each participating signal. Hence, transaction encoding allows interpretation of property behavior as an elementary signal. The handshake property is a typical representative where each *request-acknowledge* pair is composed into a transaction with a defined start and end time (see Figure 3 (b)). In case of properties that refer to a single signal only (e.g. multi-bit signals), the original signal behavior is treated as transaction.

Extract complex properties. Now, the phase of combination starts. The current *Dianosis* implementation examines three temporal dependencies between properties:

- *Mutual exclusion* of two properties at the same time.
- *Equal activity* of two properties at the same time.
- *Successive temporal order* of a number of properties.

Additionally to the restrictions applied for the creation of basic candidates property combination is limited, as well:

- Remove combinations that are not meaningful.
- Create only new candidates.
- Discard a property that is covered by another property.

Composed property candidates are analyzed over the simulation trace in the same manner that basic properties are processed (see Figure 3 (c)). Surviving candidates are added to the set of complex properties to be combined again in the next iteration. The algorithm terminates when no new property is generated.

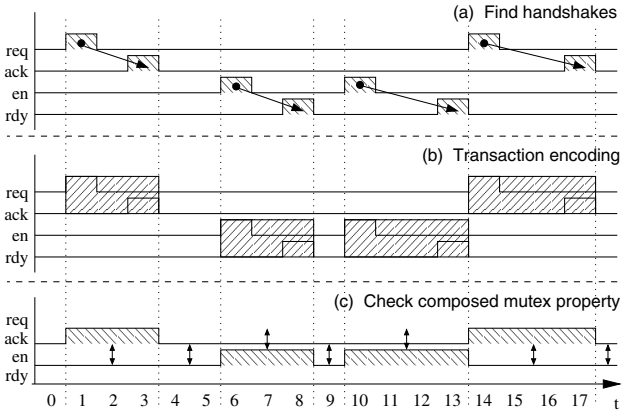

Figure 3. Property generation example

Example 2 *Figure 3 depicts the property generation in case of four signals* `req`*,* `ack`*,* `en`*, and* `rdy`*. First, two handshake properties are found between* `req` *and* `ack`*, and* `en` *and* `rdy`*. Second, the handshakes are composed into transactions describing each property activity. Last, both basic properties are checked to be mutually exclusive. Assuming that the mutex candidate survives over the simulation trace, Dianosis reports:* `not(req |−>##[2:3] ack and en |−>##[2:3] rdy)`.

4. Experimental results

To evaluate the effectiveness and quality of our solution we applied *Dianosis* to several modules of different industrial hardware designs: a SIMD multiprocessor design, a SATA FIFO interface, and a DRAM controller interface. Table 2 gives a brief characterization of the examined design components. The table shows the number of examined modules with their lines-of-code (LOC), the signals per module used to generate properties, and the number of simulation cycles in the input trace for *Dianosis*. The trace lengths vary up to nearly 14 million cycles (trace size approx. 1GB). Since we are interested in the external design behavior, only the module I/O signals are inspected.

Table 2. Testbench characteristics

Testbench	#Modules	LOC	#Signals	Cycles
SIMD MP	9	till 498	8 to 158	990,438
SATA FIFO	1	1369	109	13,763,441
DRAM ctrl	1	183	120	229,820

4.1. Generated properties

The results of the basic property generation phase are shown in Table 3. *Dianosis* generates a relatively small but relevant set of properties. All properties were cross checked by designers and classified into three categories: C - correct, I - incomplete, and W - wrong. Incomplete properties only

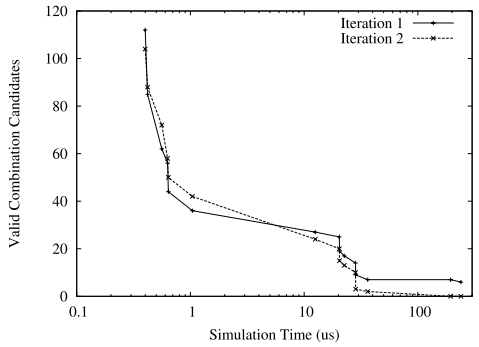

Figure 4. SATA FIFO combination candidates

partially characterize the expected design behavior. Usually, this indicates an insufficient simulation trace unveiling a bad testbench coverage. This case also leads to the sole unproved property. Hence, incomplete or wrong properties supply valuable information to improve the test suite.

Table 3. Found basic properties

Testbench	properties	C	I	W	gen. time
SIMD MP	43	38	4	1	74 sec[1]
SATA FIFO	20	11	9	0	34 min[1]
DRAM ctrl	9	9	0	0	150 sec[1]

[1] **Test system:** AMD Opteron™248 @2.2GHz, 8GB RAM

Table 4 summarizes the results we achieved from the property combination phase. This phase requires only a small fraction of the time needed to infer basic properties. There, all complex properties are generated by 1–3 iterations using maximally three temporal dependency operators. This demonstrates the efficiency of our pruning strategies (see Section 3). Comparing the quality of inferred versus hand-written properties is difficult. Some properties show a kind of cause-and-effect chain describing the interaction between multiple components that would not be explicitly written by a designer in this way. On the other hand, a formal specification for some blocks of the SIMD design contains some of the generated complex properties. This observation and the designer feedback suggest that *Dianosis* partially generates complementary properties compared to a manual writing. Furthermore, properties abstracting the design behavior are inferred (see Section 4.2).

Table 4. Found complex properties

Testbench	properties	C	I	W	gen. time
SIMD MP	25	22	3	0	3.3 sec[1]
SATA FIFO	6	4	2	0	12 sec[1]
DRAM ctrl	5	5	0	0	0.1 sec[1]

[1] **Test system:** AMD Opteron™248 @2.2GHz, 8GB RAM

Figure 4 shows the number of valid candidates while combining properties for the SATA FIFO. The combination phase consists of two iterations where the first one confirms six candidates. The second iteration finishes at 194.24 μs where all current candidates are falsified.

4.2. Complex property example

To briefly demonstrate the analysis capabilities of *Dianosis* we illustrate one complex property that is found in the SIMD design. This property is composed of three basic properties and two temporal dependency operators and is extracted from an arbiter trace where the arbiter controls the access to the processor local cache: `not(grant_lc |-> ##[2:18] ack_lc and grant_sc |-> ##[2:18] ack_sc) |-> ##0 phase=3`. The two handshakes are triggered in a mutually exclusive fashion. Here, the first handshake synchronizes a load operation into the local cache while the second handshake controls a store operation from the cache. This behavior denotes the arbitration scheme of the arbiter and confirms its correct implementation. Additionally, the constant value of the state vector at the same time approves the correct synchronization of both handshakes which is a required design decision. Hence, this complex property helps the verification engineer to understand and validate the abstract system behavior.

5. Conclusion

In this paper we introduced *Dianosis*, a tool that automatically generates complex properties. Starting from a basis of predefined properties more complex ones are iteratively composed and checked over a given simulation trace. The flexible architecture of *Dianosis* allows to easily extend its analysis capabilities by additional checkers generating further complex and valuable properties. Besides providing a new approach for an improved design understanding, automatically generated properties lower the barrier to successfully applying formal verification techniques.

References

[1] Accellera Organization. Accellera Standard OVL V2. www.accellera.org, 2007.

[2] R. Drechsler and G. Fey. Design Understanding by Automatic Property Generation. In *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pages 274–281, 2004.

[3] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. IODINE: A Tool to Automatically Infer Dynamic Invariants for Hardware Designs. In *Design Automation Conference*, pages 775–778, 2005.

[4] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301, 2002.

[5] IEEE Computer Society. IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. 2005.

[6] B. Isaksen and V. Bertacco. Verification through the Principle of Least Astonishment. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 860–867, 2006.

[7] J. Perkins and M. Ernst. Efficient Algorithms for Dynamic Detection of Likely Invariants. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 23–32, 2004.