PathFinder: A Tool for Design Exploration

Shoham Ben-David, Anna Gringauze, Baruch Sterin, and Yaron Wolfsthal

IBM Research Laboratory in Haifa
{shoham,vanna,baruch,wolfstal}@il.ibm.com

1 Introduction

In this paper we present a tool called PathFinder[1], which exploits the power of model checking for developing and debugging newly-written hardware designs. Our tool targets the community of design engineers, who—in contrast to verification engineers—are not versed in formal verification, and therefore have traditionally been distant from the growing industry momentum in the area of model checking[2].

PathFinder provides a means for the designer to explore, debug and gain insight into the behaviors of the design at a very early stage of the implementation—even before their design is complete. In the usage paradigm enabled by PathFinder, which we call *Design Exploration*, the design engineer specifies a behavior of interest, and the tool then finds and demonstrates—graphically—a set of execution *traces* compliant with the specified behavior, if any exist. When presented with each such execution sequence, the designer is essentially furnished with an insight into the design behavior, and specifically with an example of a concrete scenario in which the behavior of interest occurs. This scenario can then be closely inspected, refined, or abandoned in favor of another scenario.

Technically, PathFinder works by translating scenarios specified by the designer into safety properties, and then challenging an underlying model checker with proving the negation of those properties. If the property presented to the model checker turns out to be false, the counter example is a *trace* demonstrating the scenario requested by the designed. Thus, with PathFinder, designers can harness the power of static analysis - without being subjected to the learning curve involved with formal specification and verification.

2 The Visual Specification Interface

Path Specification. To specify a design behavior of interest (a *scenario*), the user of PathFinder creates a graphical representation of the scenario as an

[1] There is no connection between our tool and the Java PathFinder tool from NASA.

[2] In the hardware industry, there is an age-old practical separation between the roles of design engineer and verification engineer. Designers implement chip specifications, while verification engineers check the compliance of the implementation to the specifications.

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 510–514, 2002.

ordered sequence of *phases*. Each phase is represented by Boolean expressions which define the beginning and the termination conditions of the phase.

As a simple example, consider a state machine "machine(0:3)", with 16 possible values. Suppose the designer is interested in seeing a scenario where the state machine passes through states 4 and 6 and then reaches state 1—not necessarily in consecutive clock cycles. The specification is expressed by a graphical path description as shown in Figure 1 below.

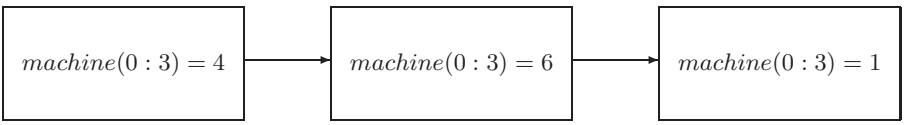

Fig. 1. Simple path specification

This path specification drives the underlying model checker to look for a trace with a state where $machine(0:3)$ has the value of 4, then in a later state the value is 6, and then on the final state of the trace, $machine(0:3)$ has the value of 1. Although restrictive, we found that this path specification formalism is expressive enough to describe behaviors of interest. More important, it incurs a minimal learning curve.

Controlling Input Behavior. A basic design principle of PathFinder has been that the user should be able to produce first traces with minimal effort. Input signals therefore have default values, to save the effort of assigning a behavior to each of them. We chose this default behavior to be non-deterministic behavior. Thus, with minimal effort, the user is able to generate initial traces. The user then moves to restrict input behavior–essentially, debugging the environment model. The more the user is willing to invest in this process, the more accurate the input behavior will be.

PathFinder offers the user a variety of ways to restrict input signal behavior. These include the ability to describe a deterministic behavior through a graphical timing diagram editor, and the use of predefined state machines. Unless a very complicated input behavior is needed, in which case it should be modeled using either Verilog or VHDL with non-deterministic extension, the user can easily define the desired behavior through graphical means.

A screen capture of the PathFinder GUI is shown in Figure 2. The path specification panel is just below the center of the GUI, where a path scenario—in the form of a sequence of phases as described above—can be visually entered by the user. Below that, we see the path constraints panel, where the user can impose constraints—specified in the Sugar [2] language—to further restrict the paths

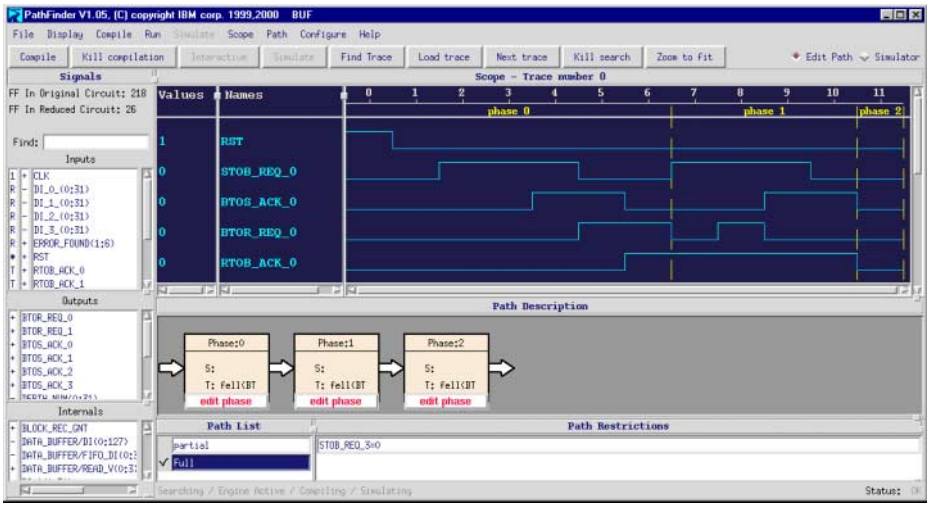

Fig. 2. Screen capture of the PathFinder GUI

he/she wants shown. The left part of the GUI is where the signals of the design are managed; in particular, input variables are controlled from here. Above the path specification panel, we see the timing diagram panel, where traces are displayed and can be further manipulated (to be presented in the next section).

3 More Key Features of PathFinder

As described in Section 1, the central theme in PathFinder is to visually demonstrate a set of execution traces of the design, which match an abstract path specified by the user. PathFinder also offers a host of additional features aimed at computing useful information on the design's behavior and making it rapidly accessible to the user. These features include:

Generation of Disjoint Multiple Traces. PathFinder includes an algorithm which, for a given path description, produces multiple traces which comply with the path specification. The generation of such multiple traces—while maintaining many variations between them—provides the user with additional information, which proved to be useful in practice. The number of traces can be specified by the user.

The Disjoint Multiple Traces algorithm [3] is heuristic, and therefore is not guaranteed to find disjoint traces. However, our practical experience shows that it almost always does.

Detection of Maximal Partial Trace. In design exploration (contrast to "bug hunting" with Model Checking), the user always expects to be presented with a trace as a result of the search. Accordingly, PathFinder produces a maximal *partial trace* (maximal in terms of *events* encountered), when no full trace exists for the given path. The algorithmic details of this feature are described in [3]. The user furthermore can interrupt the search at any time, and be presented with the maximal partial trace found in the search thus far.

Interactive Design Exploration Mode. The main purpose of this feature is to let the user gain additional information about the design as quickly as possible. In *interactive mode*, the underlying model checker does not terminate after finding the desired traces. Rather, it saves all information in memory (reachable state set, traces etc.), and interactively serves new requests coming from the user, thereby providing the user with new information as desired. The primary types of user requests supported by our experimental exploration system are presented below.

1. **Additional Cycles.** With this type of request, the user can specify a number, N, of additional cycles to extend the current trace(s). The algorithm then performs N forward steps from the final state of each trace previously computed for the path specification.
2. **Additional Traces.** This type of request allows the user to ask for N more traces, different from all the others previously produced.
3. **Longer Trace.** This type of request allows the user to ask the model checker to search for a longer trace than those already produced.

Simulation Engine. A very useful feature of PathFinder is its integrated simulator, which provides insights on design behavior and in particular can help in the debugging of traces. Once a trace is produced and displayed for the user as a timing diagram, the user can modify the values of input variables by directly manipulating the timing diagram; clicking on the input signal at the cycle to be changed will toggle the value. Then, the user can explore the different scenarios made possible by the introduction of these changes ("what-if" analysis).

4 Related Work and Experience

The problem of making formal specification and verification techniques easier to access has been addressed before. Fisler in [4] and Amla et al in [1] discuss the usage of *timing diagrams* for specification, as those are a commonly used and visually appealing specification method for designers. Hardin et al [5], in the model checker COSPAN, have implemented a feature called "check-pointing", which provides for path exploration. The contribution of PathFinder is in that unlike more common property verification tools, it provides for interactive exploration and debugging by designers, in a highly intuitive way.

Initial experiments with PathFinder reveal a good level of designer acceptance. PathFinder is used on a newly written designs with a few hundred state variables for 3–4 days, and finds 10–15 bugs on the average. We are therefore optimistic about the prospects of the Design Exploration paradigm embodied in the tool, which we feel can open new frontiers in making the power of model checking accessible to engineers at large.

References

[1] N. Amla, E. A. Emerson, R. P. Kurshan, and K. S. Namjoshi. Model checking synchronous timing diagrams. In *Formal Methods in Computer-Aided Design*, pages 283–298, 2000. 513

[2] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The temporal logic sugar. In *Computer Aided Verification, Proc. 13th International Conference*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer, 2001. 511

[3] S. Ben-David, A. Gringauze, S. Keidar, B. Sterin, and Y. Wolfsthal. Design exploration through model checking. Technical Report H0097, IBM Haifa Research Laboratory, December 2001. 512, 513

[4] K. Fisler. Timing diagrams: Formalization and algorithmic verification. *Journal of Logic, Language and Information*, 8(3):323–361, 1999. 513

[5] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Computer Aided Verification, Proc. 8th International Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 1996. 513