FoCs - Automatic Generation of Simulation Checkers from Formal Specifications

Yael Abarbanel, Ilan Beer, Leonid Gluhovsky,
Sharon Keidar, and Yaron Wolfsthal

IBM Haifa Research Laboratory, Israel
{yaell,beer,leonid,sharon,wolfstal}@il.ibm.com

1 Introduction and Motivation

For the foreseeable future, industrial hardware design will continue to use both simulation and model checking in the design verification process. To date, these techniques are applied in isolation using different tools and methodologies, and different formulations of the problem. This results in cumulative high cost and little (if any) cross-leverage of the individual advantages of simulation and formal verification.

With the goal of effectively and advantageously exploiting the co-existence of simulation and model checking, we have developed a tool called **FoCs** ("**Formal Checkers**"). FoCs, implemented as an independent component of the RuleBase toolset, takes RCTL[1] properties as input and translates them into VHDL programs ("checkers") which are integrated into the simulation environment and monitor simulation on a cycle-by-cycle basis for violations of the property.

Checkers, also called Functional Checkers, are not a new concept: manually-written checkers are a traditional part of simulation environments (cf. [GB+99]). Checkers facilitate massive random testing, because they automate test results analysis. Moreover, checkers facilitate the analysis of intermediate results, and therefore save debugging effort by identifying problems directly - "as they happen", and by pointing more accurately to the source of the problems.

However, the manual writing and maintenance of checkers is a notoriously high-cost and labor-intensive effort, especially if the properties to be verified are complex temporal ones. For instance, in the case of a checker for a design with overlapping transactions (explained in Section 3), writing a checker manually is an excruciating error-prone effort.

Observing the inefficient process of manual checker writing in ongoing IBM projects has inspired the development of FoCs, as a means for automatically generating checkers from formal specifications. For each property of the specification, represented as an RCTL formula, FoCs generates a checker for simulation. This checker essentially implements a state machine which will enter an error

[1] RCTL includes a rich and useful set of CTL safety formulas and regular expressions, see [BBL98]

E.A. Emerson and A.P. Sistla (Eds.): CAV 2000, LNCS 1855, pp. 538–542, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Cadence Design Systems, Inc. - Ex. 1004, Page 00001
IPR2024-00356 (Cadence Design Systems, Inc. v. Semiconductor Design Technologies, LLC)

state in a simulation run if the formula fails to hold in this run. The next section will describe the checker generation process in more detail.

Experience with FoCs in multiple projects has been very favorable in terms of verification cost and quality. Verification effort is reduced by leveraging the same formal rules for model checking of small design blocks as well as for simulation analysis across all higher simulation levels. An equally important benefit of FoCs is the conciseness and expressiveness of RCTL formulas. Formulas consisting of just a few lines can efficiently represent complex and subtle cases, which would require many lines of code if described in a language such as VHDL. This makes maintenance, debugging, porting and reuse of specifications and checkers highly cost-effective.

2 Tool Architecture and Implementation

Figure 1 shows the overall environment in which FoCs operates. The user provides a design to be verified, as well as formal specifications and a set of test programs generated either manually or automatically. FoCs translates the formal specification into checkers, which are then linked to the design and simulated with it. During simulation, the checker produces indications of property violations. It is up to the user to decide what action to take: fix the design, the property, or the simulation environment.

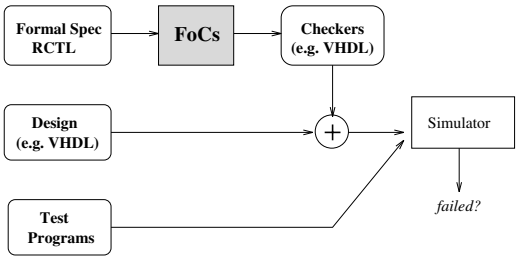

Fig. 1. FoCs Environment

FoCs translates RCTL into VHDL as follows: First, each property is translated into a non-deterministic finite automaton (NFA) and a simple $AG(p)$ formula, where p is a Boolean expression. The NFA has a set of distinguished error states, and the formula specifies that the NFA will never enter an error state (entering an error state means that the design does not adhere to the specification under the test conditions). The translation details are described in [BBL98].

Since contemporary simulators do not support non-determinism, the NFA has to be converted into a deterministic automaton (DFA). The DFA, in turn, is translated into a VHDL process - the FoCs checker. The $AG(p)$ formula is translated into a VHDL $Assert(p)$ statement that prints a message when the VHDL process reaches a state where the underlying property is violated, and possibly stops simulation.

Cadence Design Systems, Inc. - Ex. 1004, Page 00002

The number of states of the DFA may be exponential in the number of states of the NFA, but simulation is sensitive to the size of the representation (the number of VHDL lines) rather than to the number of states. The number of VHDL lines in the resulting VHDL checker is at most quadratic in the size of the property. Practically, it is almost always linear because of the types of properties that people tend to write.

The above translation process is implemented within the RuleBase model checker [BBEL96].

3 Example

The following example will demonstrate the conciseness and ease of use of RCTL for checker writing and the advantage of automatic checker generation. Assume that the following property is part of the specification: **If a transaction that starts with tag t has to send k bytes, then at the end of the transaction k bytes have been sent**. The user can formulate this property in RCTL as follows:

forall k: [2]

$\{*, start \ \& \ start_tag{=}t \ \& \ to_send{=}k, !end*, end \ \& \ end_tag{=}t\} \rightarrow \{sent{=}k\}$

Manual writing of a checker for this property may become complicated if transactions may overlap, which means that a new transaction may start while previous transactions are still active. The checker writer has to take into consideration all possible combinations of intervals and perform non-trivial bookkeeping. The RCTL formula is evidently much more concise and readable than the resulting VHDL file or a manually written VHDL or C program.

4 Using FoCs for Coverage Analysis

The quality of simulation-based verification depends not only on thorough checking of results, but also on the quality of the tests used (a.k.a. input patterns or test vectors) [KN96]. FoCs checkers can serve to enhance the quality of tests by providing a means for measuring *test coverage*. Test coverage measurement is the process of recording if certain user-defined events occurred during simulation of a set of tests. When used for coverage purposes, the FoCs checkers will evaluate the quality of the test suite by discovering events, or scenarios, that never happen during simulation. This feedback will guide the user which further tests are needed in order to cover scenarios that have not been exercised.

The implementation of coverage checkers is similar to that of functional checkers. The only difference is that instead of reporting an error, a coverage checker provides a positive indication when covering the relevant scenario. An

[2] The semantics of *forall* are intuitive. The implementation, however, is not trivial. It involves spawning a new automaton whenever a new value of k is encountered. Neither the formal semantics nor the implementation are included here due to lack of space.

Cadence Design Systems, Inc. - Ex. 1004, Page 00003

example of a scenario to cover is: **a snoop event happens twice between read and write**, which can be formulated in RCTL as follows:

$$\{*,\ read,\ \{!snoop*,\ snoop,\ !snoop*,\ snoop\}\ \&\ \{!write*\}\ \} \rightarrow \{cover\}$$

A special case of coverage analysis is detection of *simulation vacuity*. While vacuity in model checking is defined as a failure of a subformula to influence the model checking results [BBER97], simulation vacuity refers to the failure of a set of tests to trigger a functional checker, which means that the checker did not influence simulation results. To detect simulation vacuity, FoCs attaches a coverage checker to each functional checker it generates; the coverage checker indicates whether the functional checker was triggered in at least one simulation run.

5 Experience

The FoCs toolset has been deployed in several projects in IBM, notably in the GigaHertz processor development effort in Austin and in the IBM Haifa ASIC development laboratory. FoCs has also been successfully used by the formal verification team of Galileo Technology, Inc. The experience with FoCs in these projects has been very favorable in terms of verification cost and quality. Using FoCs, verification effort was reduced - reportedly by up to 50% - by using the same formal rules for model checking at the unit level and for simulation analysis in the subsystem and system levels. This reduction was achieved despite the fact that the addition of checkers increases simulation time considerably (up to a factor of two). Thousands of FoCs checkers were written so far by virtue of the great ease of writing RCTL specifications and translating them to FoCs checkers.

6 Related Work

In a previous work, Kaufmann et al [KMP98] described an approach to verification in which all specifications and assumptions have the form $AG(p)$, where p is a Boolean formula. Both specifications and assumptions of this form can be used in simulation, with specification being translated into simple checkers, and assumptions being translated into testbench code which avoids leading the simulation into undesired states.

Canfield et al [CES97] describe a platform called Sherlock, aiming to serve both model checking (through translation to CTL) and simulation. Sherlock includes a high level language for specifying reactive behavior, while we use RCTL - a regular expression based language. While Sherlock postprocesses simulation traces, our method works during simulation. No experience has been described in [CES97]. Our own experience has demonstrated that the simple, concise syntax and semantics of RCTL, coupled with online simulation checking, is highly useful to the verification teams with whom we work.

Cadence Design Systems, Inc. - Ex. 1004, Page 00004

Finally, in [SB+97], Schlipf et al describe a methodology and tool that inspired our work. They unify formal verification and simulation efforts by automatically translating state machines which represent the environment specification either to simulation behavioral models or to input for a model checker. Boolean assertions attached to the state machines serve as $AG(p)$ formulas in model checking or $Assert(p)$ in simulation. In contrast, our solution represents the specification as temporal logic formulas rather than state machines, for the sake of conciseness and readability.

7 Future Plans

Although focused on checker generation for functional testing and coverage analysis, we view FoCs as a step towards a full methodology of "Formal Specification, Design and Verification". We intend to provide a set of integrated, complementary tools that will facilitate the use of formal specification for multiple purposes. Once written, the formal specification will:
- Serve as an executable specification; architects can experiment with it while defining the specification
- Be used as a golden model to resolve ambiguities and misunderstandings during the implementation stage
- Be translated into temporal formulas for model checking
- Be translated into simulation checkers
- Be used for derivation of coverage criteria
- Provide hints for automatic test generation

We believe that such a methodology, once supported by the appropriate tools, will significantly contribute to the quality and efficiency of the design process.

References

[BBEL96] I. Beer, S. Ben-David, C. Eisner, A. Landver, "RuleBase: an Industry-Oriented Formal Verification Tool", Proc. DAC'96, pp. 655-660.

[BBER97] I. Beer, S. Ben-David, C. Eisner, Y. Rodeh, "Efficient Detection of Vacuity in ACTL Formulas", CAV'97, LNCS 1254, pp. 279-290.

[BBL98] I. Beer, S. Ben-David, A. Landver, "On-The-Fly Model Checking of RCTL Formulas", CAV'98, LNCS 1427, pp. 184-194.

[CES97] W. Canfield, E.A. Emerson, A. Saha, "Checking Formal Specifications under Simulations" Proc. ICCD'97.

[GB+99] D. Geist, G. Biran, T. Arons, Y. Nustov, M. Slavkin, M. Farkash, K. Holtz, A. Long, D. King, S. Barret, "A Methodology for Verification of a System on Chip", Proc. DAC'99.

[KN96] M. Kantrowitz, L. M. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness ", Proc. DAC'96.

[KMP98] M. Kaufmann, A. Martin, C. Pixley, "Design Constraints in Symbolic Model Checking", CAV '98, LNCS 1427, pp. 477-487.

[SB+97] T. Schlipf, T. Buechner, R. Fritz, M. Helms and J. Koehl, "Formal verification Made Easy", IBM Journal of R&D, Vol. 41, No. 4/5 , 1997.

Cadence Design Systems, Inc. - Ex. 1004, Page 00005