Testing the $FormalCheck^{TM}$ Query Library

Shira Dershowitz Kathi Fisler Sandeep Shukla
Gerard Holzmann Robert Kurshan Doron Peled
Computing Principles Research
Bell Laboratories
Murray Hill, NJ 07974
email: {doron,gerard,k}@research.bell-labs.com

Abstract

FormalCheck^{TM} is a Computer-Aided Design tool developed jointly by Lucent Technologies' Bell Labs Research and its Advanced Technologies Design Automation Lab, for the automatic verification of hardware designs. Design Automation supports FormalCheck both internally and commercially. The utility of this tool lies in its ability to offer a more reliable means for checking the correctness of a control circuit than is afforded with simulation, together with a greater speed and ease of application than is possible using simulation. It is applied to a circuit by posing queries *about the circuit (relating to its proper operation) which the tool then answers either in the affirmative, or in the negative– presenting a trace to a point where the circuit fails to exhibit the behavior defined in the query. Internally to the tool, each query is translated into an automaton model, against which the circuit is checked. It therefore is absolutely critical that the automata models generated by the queries conform to the intentions of the designers who formulate the queries. We have tested this conformance and established a regression test bench for the query mechanism, by establishing two unrelated automata generators for queries, and a means to test that the generated automata are equivalent.*

1 Introduction

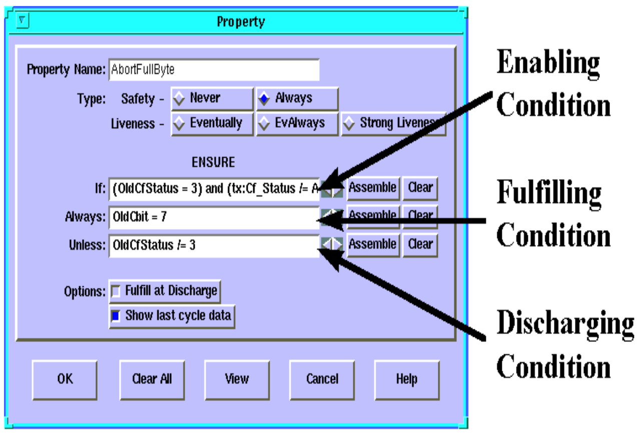
FormalCheck^{TM} [DG96] is a Computer-Aided Design tool developed jointly by Lucent Technologies' Bell Labs Research and its Advanced Technologies Design Automation Lab, for the automatic verification of hardware designs. Design Automation supports FormalCheck both internally and commercially. The utility of this tool lies in its ability to offer a more reliable means for testing a control circuit than is afforded with simulation, together with a greater speed (for given coverage) and ease of application than is possible using simulation.

Verification in the context of FormalCheck consists of an algorithmic check that the formal language of a circuit model is contained in the formal language of an automaton which captures the query or properties to be verified on the circuit model. When this language containment check passes, it means that the circuit design satisfies the properties stated in the given query, under all conditions consistent with the query. This is much more general than simulation which can check a circuit design only for selected input sequences or scenarios.

FormalCheck employs COSPAN [KH90, HHK96] as its verification engine. COSPAN, a general-purpose automata-based tool for *coordination specification analysis*, contains the algorithms to perform the language containment check [Kur94].

FormalCheck is applied to a circuit by posing *queries* about the circuit (relating to its proper operation) which the tool then answers either in the affirmative, or in the negative– presenting a trace to a point where the circuit fails to exhibit the behavior defined in the query. Internally to the tool, each query is translated into an automaton model, against which the circuit is checked by COSPAN. A query consists of two parts: *properties* for which the circuit model is checked, and *constraints* on the environment of the circuit, which are assumptions about the behavior of companion components. In order to verify the given circuit, it is assumed that the companion components fulfill their behavioral requirements. Without such an assumption, the given circuit may not behave properly (and it is not required to behave properly).

The user of FormalCheck defines each query (its properties and constraints) through a graphical interface (Figure 1) which supports a very simple logical idiom. In this idiom, each property and constraint is constructed from three qualified conditions: a *fulfilling* condition which defines a required or assumed event, an *enabling* condition which de-



Property Name: AbortFullByte
Type: Safety - Never [✓ Always]
Liveness - Eventually EvAlways Strong Liveness
If: (OldCfStatus = 3) and (tx:Cf_Status /= A
Always: OldCbit = 7
Unless: OldCfStatus /= 3
Options: Fulfill at Discharge [✓ Show last cycle data]

Figure 1: The FormalCheck *Property* panel.

fines a precondition for starting the check of the fulfilling condition, and finally, a *discharging* condition, after which the fulfilling condition no longer is required or assumed. An example is

After(*BusRequest*)
Always(*BusRequestFlag=1*)
Until(*BusGrant*)

which defines a property that requires a *BusRequestFlag* to remain high after a *BusRequest* event, until a *BusGrant* event. This behavior is required *ad infinitum*, whenever the *BusRequest* event occurs.

In this example, the three qualifiers After, Always and Until mediate the effect of the enabling, fulfilling and discharging conditions, respectively. This property would become a constraint by changing the Always qualifier to AssumeAlways instead.

In general, each condition is a Boolean parameter and each format such as the

After(*enabling condition*)
Always(*fulfilling condition*)
Until(*discharging condition*)

format of the example gets translated into a parameterized automaton whose parameters are the respective conditions.

Once the user has defined a query, FormalCheck translates that query into an automaton against which COSPAN checks the circuit model. The translation follows a separate translation rule for each possible format. All the possible formats are stored in a “format library” named QRY.h and FormalCheck performs the translation by invoking the appropriate format, instantiating each of its Boolean parameters with the respective user-provided condition for which it is substituted, and writing out the resulting automaton in the input language of COSPAN.

All queries (properties and constraints) are constructed in this fashion. It therefore is absolutely critical that the automata models generated by the queries conform to the intentions of the designers who formulate the queries. This note describes how we have tested this conformance and established a regression test for the query automaton-generation mechanism by establishing a second unrelated automaton generator for queries, and a means to test that the respective automata generated by FormalCheck and the second generator are equivalent.

The second automaton generator starts with the English language query idiom supported by the FormalCheck graphical interface. The plan was to (hand-)translate each property and constraint format into a parameterized formula of linear-time temporal logic (LTL) [VW86, Eme90], a logic well-suited for capturing temporal behavior. These formulas then could be translated into automata using an algorithm [GPVW95] developed for that purpose and implemented into the SPIN verification tool [Hol91, HP96]. The implementation in SPIN was augmented in order to generate automata in the input language of COSPAN. Having done this, an algorithm in COSPAN could be used to check the equivalence of each pair of automata: one generated by FormalCheck and the other generated by SPIN. Although the automata are parameterized by the respective Boolean conditions, it suffices to treat each Boolean condition as an independent Boolean input read by each of the two automata. In this way, the equivalence of any pair of automata could be checked once and for all, for any instantiation of the parameters.

Interestingly, it soon was discovered that virtually none of the automata formats in QRY.h could be translated into LTL. It is generally known that the expressive power of LTL is strictly weaker than that of automata [Eme90], and as it happens, this fact is exemplified by most of the automata of QRY.h (as was proved formally [Kup96, Wil96]). The particular reason the QRY.h automata cannot be translated into LTL lies in the “phase” nature of the QRY.h format: LTL is incapable of keeping track of whether the particular automaton is in the phase where the fulfilling condition is required to hold, or not. However, this phase is implemented by a very simple 2-state machine: starting in state 0, it moves to state 1 when the *enabling condition* becomes true, and moves back to state 0 when the *discharging condition* becomes true. Referring to the state of this *PHASE* machine, each QRY.h format then could be translated into LTL, and this was done. The rest of the test was carried out as above.

As a technical point, it should be noted that all the automata under discussion are ω-automata [VW86, Eme90, Kur94]. These are finite-state automata which define nonterminating behaviors, accepting (infinite) sequences of letters rather than (finite) strings as do conventional automata. There are many similarities (and some crucial differences) between the two classes of automata.

2 The Query Format Library QRY.h

Each property and each constraint format in QRY.h has a *fulfilling condition* and an optional *enabling condition* and an optional *discharging condition*. The query format supports two qualifiers for enabling conditions:

After
IfRepeatedly

and two main qualifiers for discharging conditions:

Unless
Until

the difference being that Until requires that the discharging condition eventually become true, whereas with Unless, discharge is not mandatory. Each discharging qualifier also has the respective variant

UnlessAfter
UntilAfter

which requires fulfillment at discharge: the fulfilling condition must be true at the instant the discharging condition becomes true (but may become false thereafter).

For properties, the fulfilling condition has two "safety" qualifiers:

Always
Never

and two "liveness" qualifiers:

Eventually
EventuallyAlways.

For constraints, the corresponding qualifiers are

AssumeAlways
AssumeNever
AssumeEventually
AssumeEventuallyAlways.

The *safety* qualifiers define behaviors which may be falsified by a finite trace, whereas *liveness* qualifiers define behaviors which can be falsified only by an infinite trace. The liveness qualifiers define behaviors in which the fulfilling condition eventually becomes true, or eventually stays true, respectively. In the later case, it is not required that it stays true the first time it becomes true.

The enabling condition qualifier IfRepeatedly is supported only with the fulfilling condition qualifiers Eventually and AssumeEventually. With the first, the resulting property is called "strong liveness" while with the second, the resulting constraint is called "strong fairness". On account of the ability to express strong fairness together with any finite state machine (which may be implemented by a sufficient number of copies of the PHASE machine), the automata of QRY.h are capable of expressing any ω-regular property [Kur94]. The ω-regular properties form the largest class of properties expressible with finite state automata.

Each property/constraint pair (formats which differ only by the Assume) define automata which accept complementary languages. Thus, another check on the validity of the QRY.h library is to check this requirement with COSPAN, as well. This additional check also was performed.

Thus, the validity of QRY.h was tested by the following checks:

- equivalence of each QRY.h automaton with the corresponding LTL+Phase formula

- respective property automaton/constraint automaton pairs define complementary languages

A script was written to invoke COSPAN to perform each check for each parameterized automaton, in succession. This script is used to perform regression tests on QRY.h if any changes are made to its elements. In fact, more efficient implementations for some of the QRY.h automata were subsequently discovered, and these were tested using the test script.

The QRY.h library and the COSPAN and SPIN tools are freely available to Lucent Technologies employees. For QRY.h and COSPAN, requests should be addressed to k@research.bell-labs.com. For SPIN, requests should be addressed to gerard@research.bell-labs.com. The FormalCheck tool is available under arrangement from Advanced Technologies' Design Automation Lab. Requests should be addressed to gdepalma@lucent.com.

3 Conclusion

A method was devised and implemented to test the validity of the FormalCheck query library QRY.h, by comparing two independent implementations using a script which later served as a regression tester. In the course of the original test, a few typographical errors were found in QRY.h elements, and were corrected. On account of this test, the anticipated reliability of QRY.h has been enhanced significantly.

References

[DG96] G. DePalma and A. B. Glaser. Formal verification augments simulation. *Electronic Engineering Times*, pages 56, 66, Jan. 1996.

[Eme90] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Vol B*, 1990.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. *In Proceedings Protocol Specification Testing and Verification, Warsaw, Poland*, pages 3–18, 1995.

[HHK96] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In *Proc. CAV'96*, volume 1102, pages 423–427. LNCS, 1996.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[HP96] G. J. Holzmann and D. Peled. The State of SPIN. In *Proc. CAV'96*, volume 1102, pages 385–389. LNCS, 1996.

[KH90] R. P. Kurshan and Z. Har'El. Software for analytical development of communication protocols. *AT&T Tech. J.*, 69(1):45–59, 1990.

[Kup96] O. Kupferman. personal correspondence, 1996.

[Kur94] R. Kurshan. *Computer Aided Verification of Coordinating Processes : The Automata-Theoretic Approach*. Princeton University Press, 1994.

[VW86] M. Y. Vardi and P. Wolper. An automata theoretic approach to automatic program verification. In *IEEE Logic In Computer Science*, 1986.

[Wil96] T. Wilke. personal correspondence, 1996.