Presented at the National Security Agency's third High Confidence Software and Systems Conference, Baltimore MD, April 2003

Embedded Deduction With ICS[*]

Leonardo de Moura, Harald Rueß, John Rushby, and Natarajan Shankar

Computer Science Laboratory
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025, USA
`demoura|ruess|rushby|shankar@csl.sri.com`

Abstract. Formal analyses can provide valuable assurance for high confidence software and systems. The analyses can range from strong typechecking through test case generation and static analysis to model checking and full verification. In all cases, the tools that support the analyses use formal deduction in some way or other. ICS is a fully automatic, high-performance decision procedure for a broad combination of theories that can be embedded in all tools of this kind to provide them with a core deductive capability of exceptional power and performance. We describe the design choices underlying ICS and the capabilities it provides.

1 Introduction

Formal deduction—that is, automated theorem proving—lies at the heart of all tools for formal analysis of software and system descriptions. In formal verification systems such as PVS [10], the deductive capability is explicit and visible to the user, whereas in tools such as test case generators it is hidden and often ad-hoc. We believe that all tools for formal analysis would benefit—both in performance and ease of construction—if they could draw on a powerful embedded service to perform common deductive tasks.

Examples of the tasks that can be required are those that ask whether one formula is a consequence of others (e.g., is $4 \times x = 2$ a consequence of $x \leq y$, $x \leq 1 - y$, and $2 \times x \geq 1$ when the variables range over the reals?), and those that ask whether an assignment to variables can be found that satisfies a set of constraints (e.g., find an a such that $car(a) = cons(b, c)$). The first task is a decision problem that might arise in verification, the second is a constraint satisfaction problem that could arise in test case generation. Notice that both examples involve interpreted theories: rational linear arithmetic in the first, and lists in the second.

An embedded deductive service should be fully automatic, and this suggests that its focus should be restricted to those theories whose decision and satisfiability problems are decidable. However, there are some contexts that can tolerate incompleteness (e.g., in extended static checking, the failure to prove a true theorem results only in a spurious warning message), and others where speed may be favored over completeness (e.g., in

[*] This research was supported by SRI internal investment funds, by NASA under contract NAS1-00079, by the DARPA NEST program under AFRL contract F33615-01-C-1908, and by NSA under contract MDA904-02-C-1196

construction of abstractions), so that undecidable theories (e.g., nonlinear integer arithmetic) and those whose decision problems are often considered infeasible in practice (e.g., real closed fields) should not be ruled out completely.

Most problems that arise in practice involve *combinations* of theories: the question whether

$$f(cons(4 \times car(x) - 2 \times f(cdr(x)), y)) = f(cons(6 \times cdr(x), y))$$

follows from $2 \times car(x) - 3 \times cdr(x) = f(cdr(x))$, for example, requires simultaneously the theories of uninterpreted functions, linear arithmetic, and lists. The ground (i.e., quantifier-free) fragment of many combinations is decidable when the full (i.e., quantified) combination is not, and practical experience indicates that automation of the ground case is adequate for most applications.

Practical experience also suggests several other desiderata for an effective deductive service. Some applications (e.g., construction of abstractions) invoke their deductive service a huge number of times in the course of a single calculation, so that performance of the service must be very good (e.g., tens or hundreds of thousands of invocations per second). Other applications (e.g., proof search) explore many variations on a formula (i.e., alternately asserting and denying various combinations of its premises), so the deductive service should not examine individual formulas in isolation, but should provide a rich API that supports incremental assertion, retraction, and querying of formulas. Other applications (e.g., test case generation) generate propositionally complex formulas (i.e., formulas with thousands or millions of propositional connectives applied to terms over the decided theories), so that this type of proof search must be performed efficiently inside the deductive service.

We have developed a system called ICS (the name stands for *Integrated Canonizer/Solver*) that can be embedded in applications to provide deductive services satisfying the desiderata above. In the following sections, we outline the design choices embodied in ICS, its capabilities and method of operation, and describe some of its applications.

2 Core ICS

The core of ICS is a decision procedure for a combination of ground theories including equality with function symbols, integer and rational linear arithmetic, fixed-length bitvectors, arrays, tuples, and coproducts (the combination of the last two provides abstract datatypes such as lists and binary trees). Apart from bitvectors, this capability is similar to that of the decision procedures in PVS (e.g., the `assert` command), but ICS can handle much larger formulas.

It is crucial to its utility that ICS is able to decide a *combination* of theories. It is desirable to achieve this by combining decision procedures for its individual theories in a modular fashion. However, there is a tradeoff between modularity and performance. The combination method of Nelson and Oppen [9], for example, imposes few restrictions on its component theories and their decision procedures, but yields relatively low performance. This is because the separate decision procedures do not share much state and communicate only by propagating newly discovered equalities back and forth. The combination method of Shostak [14], on the other hand, requires that its component theories are *canonizable* and *solvable*, and achieves high performance by tightly integrating these components through an efficient data structure for congruence closure. Most theories of practical interest are canonizable and solvable, so ICS uses a corrected version of Shostak's method. Theories that do not satisfy the requirements for Shostak's method can be integrated using Nelson and Oppen's method above the Shostak combination.

As mentioned, an efficient data structure and procedure for congruence closure lies at the heart of ICS. This provides a decision procedure for the theory of equality with uninterpreted function symbols, and is used to integrate decision procedures for other canonizable and solvable theories. Early treatments of this integration were incorrect and could yield incomplete or nonterminating procedures. The first correct treatment for the integration of congruence closure with one other theory was developed by Shankar and Rueß [12]; this construction has been formally verified in PVS by Ford and Shankar [6]. The extension to multiple theories is not straightforward because, although the combination of the canonizers for the constituent theories yields a canonizer for the combined theory (which is an independently useful artifact), the combination of the solvers may not (contrary to previous belief) be a solver for the combination. The first correct extension to multiple theories also was developed by Shankar and Rueß [13].

A decision procedure (i.e., canonizer and solver) for rational linear arithmetic is quite straightforward and efficient, but integer linear arithmetic is more challenging because it can require case-splitting (i.e., search) to determine whether some property is satisfied by an integer in a certain range (hence, the problem is NP-complete). There are straightforward methods for this problem that are easily shown to be complete (e.g., the method of Fourier-Motzkin), but they are inefficient on cases that commonly arise in practice (e.g., constraints of the form $x - y \leq c$, where x, y are variables and c is an integer constant). ICS uses a new method that is efficient on the common cases, complete, and smoothly extensible to richer fragments such as nonlinear arithmetic.

Verification and model checking for hardware generally involve reasoning over bitvectors. It is, of course, possible to treat each bit as a Boolean variable and then use an efficient decision procedure for the Booleans, but this immediately invites an exponential case explosion. A better method is to split the bitvectors into chunks (not individual bits) and to do so only when necessary. ICS uses a method of this kind for fixed-length bitvectors [2, 7] and integrates it with integer arithmetic for their numerical (e.g., unsigned and twos-complement) interpretations.

In addition to the theories described above, ICS also decides the theories of arrays, tuples, and coproducts; the combination of the latter two can represent abstract datatypes such as lists and binary trees.

Core ICS operates as a decision procedure: it reports whether the formula under consideration is valid—which is equivalent to its negation being unsatisfiable. In the case that a formula is satisfiable, the ICS data structures contain sufficient information to extract a satisfying assignment—although this is not yet implemented.

3 ICS with SAT

Core ICS operates on formulas that are conjunctions of terms in the combination of its theories. However, many applications generate proof obligations or constraints that have richer propositional structure. For example, a test case of length 2 for a shift register may reduce to satisfiability of the following formula.

$$(x_1 = x_0[1:n-1] +\!\!+ 1_1) \wedge (x_2 = x_1[1:n-1] +\!\!+ 1_1) \wedge$$
$$(x_0 \neq 0_n \vee x_1 \neq 0_n \vee x_2 \neq 0_n) \wedge (x_0 = x_2 \vee x_1 = x_2).$$

where $x[1:r]$ denotes extraction of bits 1 through r of the bitvector x of length n, $+\!\!+$ denotes bitvector concatenation, and 1_r (resp. 0_r) denotes the bitvector of length r whose bits are all 1 (resp. 0).

The disjunctions in formulas such as this necessitate search and the challenge is to integrate this capability with core ICS. The PVS `ground` command provides modest functionality of this type with the assistance of an external BDD package. The problem with this approach is that the BDD represents all possible satisfying assignments (and is therefore expensive to construct), whereas we would be satisfied with just one (or the knowledge that there are none). Propositional satisfiability solvers (SAT solvers) provide this more targeted type of search and recent advances have made them extraordinarily fast for many problems that arise in practice—often they are able to discharge formulas with hundreds of thousands of variables and millions of terms in seconds or a few minutes [8].

To connect core ICS to a SAT solver, we use *variable abstraction*: each interpreted term (e.g., $x_1 = x_0[1:n-1] +\!\!+ 1_1$) is replaced by a distinct propositional variable (e.g., p) and the SAT solver is asked to solve the resulting propositional system. The truth values assigned to the propositional variables by the SAT solver are then extended to their original interpretations and the core ICS decision procedure checks them for consistency. If the interpretations are consistent, then we are done; if not, the root of the inconsistency can be generated and passed to the SAT solver as an additional constraint (we call this the generation of "lemmas on demand" [3]). For example, if p represents the term $x = y$, q represents $f(x) = f(y)$, and the SAT solver returns $p, \neg q$, then core ICS will detect the inconsistency in the interpretation $x = y \wedge f(x) \neq f(y)$ and can generate the lemma $\neg p \vee q$ as a new constraint for the SAT solver. Proceeding back and forth in this way, the SAT solver generates new candidate assignments and the decision procedure generates new additional constraints until either we find an assignment whose interpretation is satisfiable, or the set of constraints becomes unsatisfiable. The effectiveness of this approach depends on how rapidly the search space is cut down at each stage by the new constraints generated by the decision procedure. The most potent constraints would be the true "root causes" of the inconsistencies detected at each stage but it can take a long time to calculate such precise constraints and this negates the savings due to the smaller search space. Good overall performance is obtained using fast heuristics that generate an approximate "explanation" for the root cause of each inconsistency [3]. We are still tuning our heuristics in search of the best overall performance.

Full ICS integrates the combined decision procedure of core ICS with a SAT solver in the manner described. We do not use an off-the-shelf SAT solver because the back-and-forth interaction with the decision procedure imposes novel requirements (e.g., we want to process new constraints incrementally from the current state, not restart from the beginning, and we also use “don’t care” assignments), but we do employ many of the techniques that make such solvers fast [15]. Our experiments indicate that the integrated SAT solver in ICS yields several orders of magnitude improvement over a looser combination using an off-the-shelf SAT solver. Used purely as a SAT solver, the performance of full ICS is comparable to Chaff [8].

Like core ICS, full ICS operates as a decision procedure, but we plan to extend it to a satisfiability procedure in the near future.

4 Using ICS

Core ICS is implemented in Objective Caml, and its SAT solver in C++; the full system functions as a C library and can be called from virtually any language. We have experience using it from C, C++, Lisp, Scheme, and Objective Caml. The system was developed under Linux but has been ported to MAC OS X and to Windows XP (under cygwin), and we anticipate little difficulty in porting it to other systems.

In addition to its C interface, ICS is provided with a simple text-based interactor that can be used for experimenting with its capabilities. ICS maintains a state that can be manipulated and queried by a series of commands. Most importantly, the `assert` command extends the current state with a new fact. The following command, for example, adds an equality over terms built from the the variable `x`, the uninterpreted function symbol `f`, the operators of linear arithmetic, and S-expressions built from the pairing function `cons(.,.)` and its first and second projections `car(.)` and `cdr(.)`.

```
ics> reset.
:ok
ics> assert 2 * car(x) - 3 * cdr(x) = f(cdr(x)).
:ok
```

We can now assert a second equality, and the response `valid` indicates that this is deduced to be a consequence of the previously asserted facts.

```
ics> assert f(cons(4 * car(x) - 2 * f(cdr(x)), y))
      = f(cons(6 * cdr(x), y)).
:valid
```

The command `sat` invokes the SAT solver (here `|` denotes disjunction and & is conjunction).

```
ics> sat (x = 1 | x = 2 | x = 3) & x > 1.
:sat(s5) [-1 + x > 0;  x = 3]
```

The response from ICS indicates that all assignments to `x` satisfying both `-1 + x > 0` and `x = 3`, describe models for the input formula (the annotation `s5` simply names this logical state). There is obviously only one possible assignment here, so the description is not minimal. Construction of concrete satisfying assignments is planned for the near future.

5 Applications of ICS

ICS can be used to provide embedded deductive support for existing applications, but its speed and power also make new applications possible. We describe representatives applications of each kind.

5.1 Discharging Proof Obligations

ICS can be used to augment or replace existing deductive capabilities in systems that generate and discharge proof obligations.

For example, ICS can be used in place of the standard decision procedures in PVS. Because the standard decision procedures have different capabilities than ICS, a PVS proof script developed using the former will generally require adjustment to work with the latter. For testing and benchmarking purposes, we have run PVS in a mode where proof scripts are guided by the standard decision procedures, but ICS is run in parallel and its behavior compared with the standard procedures. Differences were examined to ensure they were intended. We used proofs of the 750 theorems in the PVS prelude (built-in library) as our test bench. Despite its more costly interface (PVS and its standard decision procedures are implemented in Lisp, from which ICS is invoked as a foreign-function through its C interface) and the fact that PVS uses only its core capabilities, ICS is substantially faster on examples that really exercise the decision procedures (for small examples, any differences are swamped by the overhead of other processing in the PVS prover). Future versions of PVS will make fuller use of ICS capabilities. We anticipate that this will be beneficial both to users of PVS and to those who intend to use ICS directly but wish to use PVS to explore and prototype the deductive "glue" needed to reduce their application to the capabilities provided by ICS. Such glue is likely to involve Skolemization (and possibly quantifier instantiation), and definition expansion (and possibly rewriting).

We are currently optimizing the capabilities of ICS to support the deductive requirements of the Destiny verification system under development at NSA.

5.2 Bounded Model Checking and Test Case Generation

Bounded model checking (BMC) has become a popular debugging and assurance method for hardware designs [1]. Bounded model checking asks whether there is a counterexample of length k or less to a given property P (typically an invariant, but the method works for full linear temporal logic) of a design represented as an initiality predicate I and transition relation T. For hardware designs at the register transfer level, P, I, and T are represented directly in propositional calculus and the BMC problem then reduces to a (typically, huge) SAT problem. The performance of modern SAT solvers allows BMC to find deeper bugs on bigger designs than a standard BDD-based symbolic model checker. More importantly, BMC requires less tinkering (e.g., variable ordering, downscaling) by the user than standard model checking. Typically, the process is to try $k = 1$, then $k = 2, 3, \ldots$ until either a counterexample is found, or the resources of the computer—or the patience of the user—are exhausted.

Full ICS immediately allows BMC to be extended from hardware designs consisting of purely Boolean circuits to software and system designs (and hardware designs at higher levels of description) whose state is defined over integers, arrays, bitvectors, and datatypes, and their corresponding operations—in short, over any combination of the theories decided by ICS. We call this "Infinite BMC" since the state space is potentially infinite [5].

Given a system specified by initiality predicate I and transition relation T, there is a counterexample of length k to invariant P if there is a sequence of states $s_0, \ldots, s_k$ such that

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k).$$

The Infinite BMC problem is simply to find a satisfying assignment for $s_0, \ldots, s_k$ in this formula—which is exactly the capability of ICS. [1]

Using correct designs supplied for evaluation purposes by an industrial collaborator (they are hardware designs, but we do not know their origins or purpose), we performed Infinite BMC for increasing k until the time taken by ICS approached 30 minutes (on a 2GHz Pentium IV with 1GB of memory). At this point, one of the BMC formulas had 227,108 terms and its representation as a text file occupied 5Mb, another had 105,844 terms and a 3Mb text file, while a third had 72,291 terms and a 2Mb text file. In all cases, ICS required less than 80 Mb of memory. Observe that these are worst-case examples: the designs are correct (for the invariants concerned) and hence the BMC formulas have no satisfying assignments and the full search space must be explored. Other invariants do manifest bugs in the second of the designs mentioned above, and ICS found a counterexample to one of them of length 4, and a counterexample to another of length 6, both in under a minute.

Structural test coverage criteria, including the MC/DC criterion required for flight control software, can be specified as formulas in temporal logic [11]. Counterexamples to the negation of these formulas then constitute suitable test cases. Experiments with symbolic model checkers have shown that they can be used within this framework as very effective test case generators. Bounded model checkers should be even more effective (since they are specialized to the efficient construction of counterexamples). However, these strictly Boolean and propositional methods apply only to Boolean abstractions of software designs specified over arithmetic variables and data structures and can therefore generate infeasible test cases. Infinite BMC using ICS can be applied directly to software designs, thereby eliminating infeasible test cases and achieving accurate coverage.

[1] As noted earlier, ICS currently operates as a decision procedure: it can indicate whether a formula is valid or, equivalently, whether its negation is unsatisfiable. In the case that the negation to a formula is satisfiable, ICS does not yet produce a satisfying assignment (i.e., a concrete counterexample to the original formula). However, the Infinite BMC procedure does extract "symbolic counterexamples" from information in the ICS data structures.

5.3 k-Induction

If BMC finds a counterexample of length k, then we have found a bug, and are done. But if we fail to find a counterexample for any k up to some limit on our resources or patience, we cannot conclude that we have verified the design—for there could always be a counterexample of length longer than any that we tried.[2] To verify the design (for safety property P), we must perform some kind of inductive argument that applies to traces of all lengths. The usual way to do this by theorem proving is to establish that the property concerned is *inductive*: that is, it is true of all initial states (i.e., $I(s) \supset P(s)$) and if it is true of some state, then it is true of all its successors (i.e., $P(s) \wedge T(s,t) \supset P(t)$). The weakness of this method is that the second condition may be violated by a state s that is unreachable from an initial state. We must then replace P by a stronger property that excludes the troublesome state s and repeat the process. It is not uncommon to have to iterate this process many tens of times. Strengthening often requires human insight, though a good heuristic is often to conjoin to P a formula that asserts that s is unreachable.

A stronger form of induction requires that only when we have a sequence of k states satisfying P must all the successors also satisfy P. This is called k-induction, and it combines well with BMC: we first perform BMC of depth k and if that fails to refute the formula, we try k-induction (the formulas generated are very similar to those for BMC), and if that fails, we repeat the process for $k + 1$ ($k + 1$-induction is stronger—proves more formulas—than k-induction). Subject to certain side conditions (for example, the initial k-sequence should be acyclic), k-induction is a *complete* method for finite-state systems. These results generalize from the finite- to infinite-state case when ICS is substituted for a SAT solver, and the method becomes complete for important classes of infinite-state systems, such as timed automata [4].

Our Infinite BMC procedure built on ICS has been extended to perform k-induction (with additional optimizations–e.g., requiring that only the first state in a sequence may be an initial state) and to strengthen invariants (using the heuristic described earlier). Standard examples such as the abstracted Futurebus and Illinois cache coherence protocols are verified in seconds by this method, and standard timed automata examples such as the Fischer protocol and train gate controller are verified in fractions of a second. These results suggest that ICS can be competitive with specialized systems operating in their own domains.

6 Conclusion

ICS packages a powerful and efficient set of deductive capabilities in the form of a C library that can easily be accessed by other applications. This makes deduction available as an *embedded* capability, whereas previously it was available only through theorem provers intended for standalone operation.

[2] For some examples, it is possible to compute a *completeness threshold*, such that failure to find a counterexample shorter than the threshold is sufficient for verification. However, for most examples in practice, it is either too expensive to compute the threshold, or its value is beyond the reach of BMC.

Powerful embedded deduction will allow many conventional tools to provide new capabilities, or more potent forms of existing capabilities, at little cost. For example, a compiler can perform truly accurate common subexpression detection by asserting the path predicates to ICS, then using its canonizer to compare subexpressions.

Simple formal analysis tools (e.g., completeness and consistency checkers for tabular specifications, test case generators, and bounded model checkers) can obtain most of their deductive support from ICS, with little deductive “glue” needed in the application.

We plan to enlarge the services provided by ICS so that even less deductive glue will be required in future. In particular, we intend to add quantifier elimination, rewriting (which will also perform definition expansion), and forward chaining (which is very effective for transitive relations). The quantified form of the combination of theories used in ICS is not decidable (e.g., quantified integer linear arithmetic—Presburger Arithmetic—becomes undecidable when uninterpreted function symbols are added), but the circumstances that trigger undecidability are sharply defined (and rare in practice) so that it is possible to decide a very large and useful fragment of the full theory. We expect that our methods will be heuristically effective on the undecidable fragment also, and on other undecidable extensions (e.g., nonlinear integer arithmetic).

Other planned enhancements include generation of concrete solutions to satisfiability problems (and hence concrete counterexamples to BMC problems), and generation of proof objects (independently checkable explanations for the decisions made by ICS). We expect that the latter will also improve the interaction between core ICS and its SAT solver, and thereby further increase the performance of full ICS.

ICS focuses on providing full automation for the cases where that is effective; we do not intend to extend ICS to a general theorem prover. However, just as our original decision procedures made it possible for PVS (and its NSA-sponsored predecessor EHDM) to have a different architecture and style of interaction than previous interactive theorem provers [10], so the increased capability of ICS will allow future systems to support new and more productive styles of human interaction. We intend to explore these opportunities in our research with future versions of PVS, and to assist NSA to do the same with its own systems.

ICS is freely available for noncommercial research purposes under license to SRI. Please visit its home page at `ics.csl.sri.com`.

Acknowledgments

We are grateful for the support and guidance provided by Bill Legato and Frank Rimlinger in tailoring ICS to applications of interest to NSA.

References

Papers on formal methods and automated verification by SRI authors can generally be located by visiting home pages or doing a search from `http://www.csl.sri.com/programs/formalmethods`.

[1] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, Volume 1579 of Springer-Verlag *Lecture Notes in Computer Science*, pages 193–207, Amsterdam, The Netherlands, March 1999.

[2] David Cyrluk, Harald Rueß, and Oliver Möller. An efficient decision procedure for the theory of fixed-sized bit-vectors. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, Volume 1254 of Springer-Verlag *Lecture Notes in Computer Science*, pages 60–71, Haifa, Israel, June 1997.

[3] Leonardo de Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. Presented at SAT 2002, accepted for journal publication, May 2002. Available at `http://www.csl.sri.com/users/demoura/sat02_journal.pdf`.

[4] Leonardo de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification. Submitted for publication.

[5] Leonardo de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *International Conference on Automated Deduction (CADE'02)*, Volume 2392 of Springer-Verlag *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002.

[6] Jonathan Ford and Natarajan Shankar. Verifying Shostak. In A. Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction*, Volume 2392 of Springer-Verlag *Lecture Notes in Computer Science*, pages 347–362, Copenhagen, Denmark, July 2002.

[7] Oliver Möller and Harald Rueß. Solving bit-vector equations. In Ganesh Gopalakrishnan and Phillip Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, Volume 1522 of Springer-Verlag *Lecture Notes in Computer Science*, pages 36–48, Palo Alto, CA, November 1998.

[8] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[9] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[10] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[11] Sanjai Rayadurgam and Mats Heimdahl. Test-sequence generation from formal requirement models. In *High-Assurance Systems Engineering Symposium*, pages 23–31, IEEE Computer Society, Boca Raton, FL, October 2001.

[12] Harald Rueß and Natarajan Shankar. Deconstructing Shostak. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 19–28, IEEE Computer Society, Boston, MA, July 2001.

[13] Natarajan Shankar and Harald Rueß. Combining Shostak theories. In Sophie Tison, editor, *International Conference on Rewriting Techniques and Applications (RTA '02)*, Volume 2378 of Springer-Verlag *Lecture Notes in Computer Science*, pages 1–18, Copenhagen, Denmark, July 2002.

[14] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[15] Lintao Zhang and Sharad Malik. The quest for efficient Boolean satisfiability solvers. In A. Voronkov, editor, *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction*, Volume 2392 of Springer-Verlag *Lecture Notes in Computer Science*, pages 295—313, Copenhagen, Denmark, July 2002.