Tool description to be presented at IJCAR, Cork, Ireland, July 2004.

The ICS Decision Procedures for Embedded Deduction*

Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, and N. Shankar

Computer Science Laboratory
SRI International, 333 Ravenswood Ave.
Menlo Park, CA 94025, USA
{demoura,owre,ruess,rushby,shankar}@csl.sri.com

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. Many 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.

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. For example, in extended static checking, the failure to prove a valid verification condition results only in a spurious warning message. In other contexts such as construction of abstractions, speed may be favored over completeness, 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 fully 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. Other applications such as 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 application programming interface that supports incremental assertion, retraction, and querying of formulas. Other applications such as test case generation

* This work was supported by SRI International, by NSF grants CCR-ITR-0326540 and CCR-ITR-0325808, by NASA/Langley under contract NAS1-00079, and by NSA under contract MDA904-02-C-1196.

generate propositionally complex 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. ICS includes functionality for

- deciding equality constraints in the combination of theories including arithmetic, lists, and other commonly used datatypes,
- for solving propositional combinations of constraints,
- for incrementally processing atomic formulas in an *online* manner, and for
- managing and manipulating a multitude of large assertional contexts in a functional way.

This makes ICS suitable for use in applications with highly dynamic environments such as proof search or symbolic simulation. With an interactive theorem prover such as PVS [10], ICS can be used as a backend verification engine that manages assertional contexts corresponding to open subgoals in a *multi-threaded* way, thereby supporting efficient context switching between open subgoals during proof search. ICS is also highly efficient and is able to deal with huge formulas generated by fully automated applications such as bounded model checking.

ICS is available free of charge for noncommercial use at

ics.csl.sri.com

ICS can be used as a standalone application that reads formulas interactively, and may also be included as a library in any application that requires embedded deduction. Binaries for Red Hat Linux, Mac OSX, and Cygwin are precompiled. This distribution also includes libraries for use with C, Ocaml, and Lisp. The source code of ICS is available under a license agreement.

1 Core ICS

The core algorithm of ICS is a corrected version of Shostak's combination procedure for equality and disequality with both uninterpreted and interpreted function symbols [11, 13]. The concepts of canonization and solving have been extended to include inequalities over arithmetic terms [12]. The theory supported by ICS includes rational and integer linear arithmetic (currently integer arithmetic is incomplete), tuples and projections from tuples, coproducts, Boolean constants, S-expressions, functional arrays, combinatory logic with case-splitting, bitvectors [8], and an (incomplete) extension to handle nonlinear multiplication.

Consider, for example, demonstrating the unsatisfiability of the conjunction of the literals `f(f(x)-f(y)) <> f(z)`, `y <= x`, `y >= x + z`, and `z > 0`. Using the ICS interactor, these literals are asserted from left to right.

```
ics> assert f(f(x)-f(y)) <> f(z).
:ok s1
```

This assertion causes ICS to add `f(f(x)-f(y)) <> f(z)` to the initially empty logical context, and the resulting context is named `s1`. These names can be used to arbitrarily jump between a multitude of contexts. The `show` command displays the current state of the decision procedure.

```
ics>  show.
 d:  {u!5 <> u!4}
 u:  {u!1 = f(y), u!2 = f(x), u!4 = f(v!3), u!5 = f(z)}
 la: {v!3 = u!2 + -1 * u!1}
```

This context consists of variable disequalities in `d`, equalities over uninterpreted terms in `u`, and linear arithmetic facts in `la`. The equalities in `u` are flat equalities of the form `x = f(x1,...,xn)` with `f` an uninterpreted function symbol. Fresh variables such as `u!1` are introduced to flatten input terms. Equalities in `la` are all in solved form as explained in [13].

```
ics>  assert y <= x; y >= x + z.
:ok s2
```

These inequalities are asserted using an online Simplex algorithm [12] which generates fresh slack variables such as `k!6` and `k!7` which are restricted to non-negative values.

```
ics> show.
 d:  {u!5  <>  u!4}
 u:  {u!1 = f(y), u!2 = f(x), u!4 = f(v!3), u!5 = f(z)}
 la: {y = x-k!6,  z = -k!7-k!6, v!3 = u!2-u!1}
```

Finally, the last assertion detects the inconsistency, and returns `:unsat`.

```
ics>  assert z > 0.
:unsat {-y + x >=0,  z >0, -z + y - x >=0}
```

In such a case, ICS returns a *justification* in terms of a set of the asserted atoms that participate in demonstrating the inconsistency. This is not only useful for suggesting counterexamples, but is essential for efficient integration with a SAT solver (see below). Since there is a trade-off between the preciseness of justifications and the cost for computing them, the justification set provided by ICS might not be minimal.

The above sequence of commands can also be placed in a file `foo.ics` and ICS can be invoked in batch mode as a shell command Alternatively, using the `-server` command line flag, ICS interacts through the specified port instead of reading and writing on the standard input and output channels.

The integration of non-solvable theories such as functional arrays is obtained by an extension of the basic Shostak combination procedure. The basic idea behind this completion-like procedure is similar to the one described by Nelson [9].

```
ics> assert a[j:=x] = b[k:=y]; i <> j; i = k.
:ok s1
```

Assertion of the three literals above, for example, causes the ICS engine to explicitly generate new equalities based on forward chains. The resulting state `s1` is displayed using the `show` command.

```
ics> show.
... arr: {a!3=b[k:=y], y=a!3[k], a!3=a[j:=x], x=a!3[j], y=a[k]}
```

The representation of the array context is in terms of equalities with variables on the right-hand side and flat array terms on the left-hand side. Fresh variables such as `a!3` are introduced to flatten terms. The equality `a!3 = b[k := y]`, for example, causes the addition of the derived equality `y = a!3[k]`. With these completions, a canonizer can be defined similar to the case for solvable Shostak theories [13], and `a[k]` has canonical form `y` in the current context.

```
ics>  can a[k].
:term y
:justification {i = k,  b[k := y] = a[j := x], i <> j}
```

In addition, the ICS canonizer returns a subset of the input literals sufficient for proving validity of the equality `a[k] = y`. In general, a full case-split on array indices is required for completeness.

2 SAT-Based Constraint Satisfaction

ICS decides propositional satisfiability problems with literals drawn from the combination of theories of the core theory described above.

```
ics> sat [x > 2 | x < 4 | p] & 2 * x > 6.
:sat s2
:model [p |-> :true][-6 + 2 * x >0]
```

This example shows satisfiability of a propositional formula with linear arithmetic constraints. In addition to the satisfying assignment to the propositional variable `p`, a set of assignments for the variable `x` is described by means of an arithmetic constraint.

The verification engine underlying the `sat` command combines the ICS ground decision procedures with a non-clausal propositional SAT solver using the paradigm of *lazy theorem proving* [6]. Let ϕ be the formula whose satisfiability is being checked. Let L be an injective map from fresh propositional variables to the atomic subformulas of ϕ such that $L^{-1}[\phi]$ is a propositional formula. We can use the SAT solver to check that $L^{-1}[\phi]$ is satisfiable, but the resulting truth assignment, say $l_1 \wedge \ldots \wedge l_n$, might be spurious, that is $L[l_1 \wedge \ldots \wedge l_n]$ might not be ground-satisfiable. If that is the case, we can repeat the search with the added clause $(\neg l_1 \vee \ldots \vee \neg l_n)$ and invoke the SAT solver on $(\neg l_1 \vee \ldots \vee \neg l_n) \wedge L^{-1}[\phi]$. This ensures that the next satisfying assignment returned is different from the previous assignment that was found to be ground-unsatisfiable.

The `sat` command implements several crucial optimizations. First, the SAT solver notifies the ground decision procedures of every variable assignment during its search, and the ground decision procedure might therefore trigger non-chronological backtracking and determine adequate backtracking points. Second, the justifications of inconsistencies provided by the ground decision procedure are used to further prune the search space as described in [6]. Note that for the combination to be effective, both the ground decision procedures and the SAT solver must support the incremental introduction of information.

3 Applications

One of our main applications of ICS is within SAL [4] where it is used for bounded model checking of infinite-state systems ($BMC(\infty)$) [6] and induction proofs [7]. Transition systems are encoded in the SAL language, and $BMC(\infty)$ problems are generated in terms of satisfiability problems of propositional constraint formulas. Currently, we support verification backends for UCLID [3], CVC [2], SVC [1], and ICS for discharging these satisfiability problems. In comparison with these other systems, ICS performs favorably on a wide range of benchmarks [5].

4 Outlook

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, and forward chaining. Other planned enhancements include generation of concrete solutions to satisfiability problems, and generation of proof objects. We expect that the latter will also improve the interaction between core ICS and its SAT solver, and thereby further increase the performance of ICS.

References

1. C.W. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. *LNCS*, 1166:187–201, 1996.
2. C.W. Barrett, D.L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. *LNCS*, 2404:236–249, 2002.
3. R.E. Bryant, S. K. Lahiri, and S. A. Seshia. Deciding CLU logic formulas via boolean and pseudo-boolean encodings. *LNCS*, 2003.
4. L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV'2004*, LNCS, Boston, MA, July 2004. Springer Verlag.
5. L. de Moura and H. Rueß. An experimental evaluation of ground decision procedures. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV'2004*, LNCS, Boston, MA, July 2004. Springer Verlag.
6. L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. *LNCS*, 2392:438–455, 2002.
7. L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. *LNCS*, 2725:14–26, 2003.
8. O. Möller and H. Rueß. Solving bit-vector equations. *LNCS*, 1522:36–48, 1998.
9. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, Palo Alto, Ca., 1981.
10. S. Owre, J. Rushby, N. Shankar, and F. 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. H. Rueß and N. Shankar. Deconstructing Shostak. In *16th LICS*, pages 19–28. IEEE Computer Society, 2001.
12. H. Rueß and N. Shankar. Solving Linear Arithmetic Constraints. Technical Report SRI-CSL-04-01, CSL, SRI International, Menlo Park, CA, 94025, March 2004.
13. N. Shankar and H. Rueß. Combining Shostak theories. In S. Tison, editor, *RTA'02*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002.