ICS: Integrated Canonizer and Solver

(Tool Presentation)

Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar

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

Decision procedures are at the core of many industrial-strength verification systems such as ACL2 [2], PVS [6], STeP [3], or Z/Eves [8]. Effective use of decision procedures in these verification systems require the management of large assertional contexts. Many decision procedure, however, are typically used in a *fire-and-forget* environment, and lack an appropriate API for managing contexts and efficiently switching between contexts.

ICS (Integrated Canonizer and Solver) is a decision procedure developed at SRI International. It is based on well-developed theory, it efficiently decides formulas in a useful combination of theories, and it provides an API that makes it suitable for use in applications with highly dynamic environments such as proof search or symbolic simulation.

The theory decided by ICS is a quantifier-free, first-order theory with uninterpreted function symbols and a rich combination of datatypes including arithmetic, arrays, sets, and bit-vectors. This theory is particularly interesting for many applications in the realm of software and hardware verification. Combinations of a multitude of datatypes occur naturally in system specifications and the use of uninterpreted function symbols has proven to be essential for many real-world verifications.

The core of ICS is a recently developed congruence closure procedure [7] for the theory of equality and disequality with both uninterpreted and interpreted function symbols. The concepts of canonization and solving have been extended to include inequalities over linear arithmetic terms and propositional logic [1]. The theory currently includes:

- The usual propositional constants `true`, `false` and connectives `not`, `&`, `|`, `=>`, `<=>`.
- Equality (`=`) and disequality (`/=`).

- Rational constants and the arithmetic operators `+`, `*`, `-`; multiplication is restricted to multiplication by constants. Arithmetic predicates include an integer test and the usual inequalities `<`, `<=`, `>`, `>=`.
- Lookup a`[`x`]` and update a`[`x`:=`t`]` operations for a functional array a.
- The constant sets (`empty`, `full`), set membership (x `in` s), and set operators, including complement (`compl(`s`)`), union (s_1 `union` s_2), and intersection (s_1 `inter` s_2).
- Fixed-sized bitvectors including constants `1[`i`]` and `0[`i`]` of size i, concatenation (b_1 `++` b_2), extraction (b`[`i`:`j`]`), bit-wise operations like bitwise conjunction (`conj(`b_1`,`b_2`)`), and built-in arithmetic relations such as `add(`b_1`,`b_2`,`b`)`. This latter constraint encodes the fact that the sum of the unsigned interpretations of b_1 and b_2 equals the unsigned interpretation of b. Fixed-sized bitvectors are decided using the techniques described in [4].

ICS is capable of deciding formulas such as

- `x+2 = y => f(a[x:=3][y-2]) = f(y-x+1)`
- `f(y-1)-1 = y+1 & f(x)+1 = x-1 & x+1 = y => false`
- `f(f(x)-f(y)) /= f(z) & y <= x & y >= x+z & z >= 0 => false`

These formulas contain uninterpreted function symbols such as `f` and interpreted symbols drawn from the theories of arithmetic and the functional arrays. The list of interpreted theories above is open-ended in the sense that new theories can be added to ICS as long as they are canonizable and algebraically solvable. The modular design of ICS—both the underlying algorithms and their implementation—supports such extensions.

Verification conditions are usually proved within the context of a large number of assertions derived from the antecedents of implications, conditional tests, and predicate subtype constraints. These contexts must be changed in an incremental manner when assertions are either added or removed. Through the use of functional data structures, ICS allows contexts to be incrementally enriched in a side-effect-free manner.

ICS is implemented in Ocaml [5], which offers satisfactory run-time performance, efficient garbage collection, and interfaces well with other languages like C. The implementation of ICS is based on optimization techniques such as hash-consing and efficient data structures like Patricia trees for representing sets and maps efficiently. ICS uses arbitrary precision rational numbers from the GNU multi-precision library (GMP).

There is a well-defined API for manipulating ICS terms, asserting formulas to the current database, switching between databases, and functions for canonizing terms. This API is packaged as a C library, an Ocaml module, and a CommonLisp interface. The C library API, for example, has been used to connect ICS with PVS [6], and both an interaction and a batch processing capability have been built using this API.

Consider, for example, processing `f(y-1)-1 = y+1`, `f(x)+1 = x-1`, and `x+1 = y` from left-to-right using the interactive mode of ICS.

```
$ ics
ICS interpreter. Copyright (c) 2001 SRI International.
Ctrl-d to exit.

>  assert f(y - 1) - 1 = y + 1.
```

This equation is asserted in its solved form as `y = -2 + f(-1 + y)`. This equation is indeed considered to be in solved form, since `y` on the right-hand side occurs only in the scope of the uninterpreted `f`. Terms in the database are partitioned into equivalence classes, and the canonical representative of any term t with respect to this partition is represented by `can` t; for example:

```
> can -1 + y.
-3 + f(-1 + y)
```

It can be shown that `can` t_1 is identical to `can` t_2 iff the equality $t_1 = t_2$ is derivable in the current context. Now, the second equation is processed

```
> assert f(x) + 1 = x - 1.
```

by canonizing it to `1 + f(x) = -1 + x` and solving this equation as `x = 2 + f(x)`. Finally, `can x + 1` yields `3 + f(x)` and `can y` is `-2 + f(-1 + y)`. Thus, the third equation is solved as `f(x) = -5 + f(-1 + y)`. Since `f(x) = f(-1 + y)`, using `x = -1 + y` and congruence, there is a contradiction `-5 = 0`. Indeed, ICS detects this inconsistency, when given the assertion below.

```
> assert x + 1 = y.
Inconsistent!
```

The efficiency and scalability of ICS in processing formulas, the richness of its API, and its ability for fast context-switching should make it possible to use it as a black box for discharging verification conditions not only in a theorem proving context but also in a multitude of applications like static analysis, abstract interpretation, extended type checking, symbolic simulation, model checking, or compiler optimization.

ICS will be available free of charge under the PVS license. It will also be the default decision procedure of the upcoming release of PVS 3.0.

References

[1] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, , and N. Shankar. An integrated decision procedure for equality and inequality. Unpublished, 2001.

[2] Matt Kaufmann and J Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.

[3] Z. Manna and the STeP group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV 96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 415–418, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[4] O. Möller and H. Rueß. Solving bit-vector equations. In G. Gopalakrishnan and Ph. Windley, editors, *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, pages 36–48, Palo Alto, CA, November 1998. Springer-Verlag.

[5] The Objective Caml language. `http://caml.inria.fr/`.

[6] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[7] Harald Rueß and N. Shankar. Deconstructing Shostak. Submitted to LICS, 2001, available from `http://www.csl.sri.com/~shankar/shostak2000.ps.gz`, January 2001.

[8] Mark Saaltink. The Z/EVES system. In *ZUM '97: The Z Formal Specification Notation; 10th International Conference of Z Users*, volume 1212 of *Lecture Notes in Computer Science*, pages 72–85, Reading, UK, April 1997. Springer-Verlag.