CVC4

Clark Barrett[1], Christopher L. Conway[1], Morgan Deters[1], Liana Hadarean[1], Dejan Jovanović[1], Tim King[1], Andrew Reynolds[2], and Cesare Tinelli[2]

[1]New York University [2]University of Iowa

Abstract. CVC4 is the latest version of the Cooperating Validity Checker. A joint project of NYU and U Iowa, CVC4 aims to support the useful feature set of CVC3 and SMT-LIBv2 while optimizing the design of the core system architecture and decision procedures to take advantage of recent engineering and algorithmic advances. CVC4 represents a completely new code base; it is a from-scratch rewrite of CVC3, and many subsystems have been completely redesigned. Additional decision procedures for CVC4 are currently under development, but for what it currently achieves, it is a lighter-weight and higher-performing tool than CVC3. We describe the system architecture, subsystems of note, and discuss some applications and continuing work.

1 Introduction

The Cooperating Validity Checker series has a long history. The Stanford Validity Checker (SVC) [3] came first, incorporating theories and its own SAT solver. Its successor, the Cooperating Validity Checker (CVC) [16], had a more optimized internal design, produced proofs, used the Chaff [13] SAT solver, and featured a number of usability enhancements. Its name comes from the *cooperative* nature of decision procedures in Nelson-Oppen theory combination [14], which share amongst each other equalities between shared terms. CVC Lite [1], first made available in 2003, was a rewrite of CVC that attempted to make CVC more flexible (hence the "lite") while extending the feature set: CVC Lite supported quantifiers where its predecessors did not. CVC3 [4] was a major overhaul of portions of CVC Lite: it added better decision procedure implementations, added support for using MiniSat [11] in the core, and had generally better performance.

CVC4 is the new version, the fifth generation of this validity checker line that is now celebrating fifteen years of heritage. It represents a complete re-evaluation of the core architecture to be both performant and to serve as a cutting-edge research vehicle for the next several years. Rather than taking CVC3 and redesigning problem parts, we've taken a clean-room approach, starting from scratch. Before using any designs from CVC3, we have thoroughly scrutinized, vetted,

This work partially supported by the NSF (CCF–0644299, CCF–0914956, CNS–1049495, and 0914877), AFOSR (FA9550–09–1–0596 and FA9550–09–1–0517), SRC 2008–TJ–1850, and MIT Lincoln Laboratory.

and updated them. Many parts of CVC4 bear only a superficial resemblance, if any, to their correspondent in CVC3. However, CVC4 is fundamentally similar to CVC3 and many other modern SMT solvers: it is a DPLL(T) solver [12], with a SAT solver at its core and a delegation path to different decision procedure implementations, each in charge of solving formulas in some background theory. The re-evaluation and ground-up rewrite was necessitated, we felt, by the performance characteristics of CVC3. CVC3 has many useful features, but some core aspects of the design led to high memory use, and the use of heavyweight computation (where more nimble engineering approaches could suffice) makes CVC3 a much slower prover than other tools. As these designs are central to CVC3, a new version was preferable to a selective re-engineering, which would have ballooned in short order. Some specific deficiencies of CVC3 are mentioned in this article.

2 Design of CVC4

CVC4 is organized around a central core of *engines:*

- The *SMT Engine* serves as the main outside interface point to the solver. Known in previous versions of CVC as the *ValidityChecker*, the *SMT Engine* has public functions to push and pop solving contexts, manipulate a set of currently active assumptions, and check the validity of a formula, as well as functions to request proofs and generate models. This engine is responsible for setting up and maintaining all user-related state.
- The *Prop Engine* manages the propositional solver at the core of CVC4. This, in principle, allows different SAT solvers to be plugged into CVC4. (At present, only MiniSat is supported, due to the fact that a SAT solver must be modified to dispatch properly to SMT routines.)
- The *Theory Engine* serves as an “owner” of all decision procedure implementations. As is common in the research field, these implementations are referred to as *theories* and all are derived from the base class *Theory*.

CVC3 used what was in effect a domain-specific language for proof rules, which formed the trusted code base of the system. No fact could be registered by the system without first constructing a *Theorem* object, and no *Theorem* object could be constructed except through the trusted proof rules.

CVC4’s design takes a different approach. Much time and memory was spent in CVC3’s *Theorem*-computation. When not producing proofs, CVC4 uses a more lightweight approach, with *Theory* objects similar to those suggested by modern DPLL(T) literature [12] and used in other solvers. CVC4’s *Theory* class is responsible for checking consistency of the current set of assertions, and propagating new facts based on the current set of assertions. *Theorem* objects are not produced up front for theory-propagated facts, but rather can be computed lazily (or not at all, when the DPLL core doesn’t require them).

CVC4 incorporates numerous *managers* in charge of managing subsystems:

- The *Node Manager* is one of the busiest parts of CVC4, in charge of the creation and deletion of all expressions ("nodes") in the prover. *Node* objects are immutable and subject to certain simplifying constraints.[1] Further, *Node* objects are unique; the creation of an already-extant *Node* results in a reference to the original. Node data is reference-counted (the *Node* class itself is just a reference-counted smart pointer to node data) and subject to reclamation by the *Node Manager* when no longer referenced; for performance reasons, this is done lazily (see below for performance justification).
- The *Shared Term Manager* is in charge of all shared terms in the system. Shared terms are detected by the *Theory Engine* and registered with this manager, and this manager broadcasts new equalities between shared terms.
- The *Context Memory Manager* is in charge of maintaining a coherent, backtrackable data context for the prover. At its core, it is simply a region memory manager, from which new memory regions can be requested ("pushed") and destroyed ("popped") in LIFO order. These regions contain saved state for a number of heap-allocated objects, and when a pop is requested, these heap objects are "restored" from their backups in the region. This leads to a nice, general mechanism to do backtracking without lots of *ad hoc* implementations in each theory; this is highly useful for rapid prototyping. However, as a general mechanism, it must be used sparingly; it is often beneficial to perform backtracking manually within a theory using a lighter-weight method, to timestamp to indicate when a previously-computed result is stale, or to develop approaches requiring little or no backtracking at all (*e.g.*, tableaux in Simplex).

2.1 Expressions ("nodes")

Expressions are represented by class *Node* and are considerably more efficient than CVC3's expression representation. In the latest version of CVC3, expressions maintain 14 word-sized data members (plus pointers to child expressions). In CVC4, nodes take 64 bits plus child pointers, a considerable space savings. (In part, this savings results from clever bit-packing. Part is in storing node-related data outside of *Node* objects when appropriate.)

The expression subsystem of CVC4 has been carefully designed, and we have analyzed runtime profiling data to ensure its performance is reasonable. On stress tests, it beats CVC3's expression subsystem considerably. We performed a handful of targeted experiments to demonstrate this (all results are speedups observed over a large number of iterations of the same test within the same process):

[1] For example, PLUS nodes, representing arithmetic addition, must have two or more children. This is specified by the theory of arithmetic and enforced by the *Node Manager*; this arity can then be assumed by code that manipulates arithmetic-kinded nodes. If an input language permits unary PLUS, that language's parser must convert that input expression into a valid CVC4 *Node*.

Set-up/tear-down. First, we wanted to measure raw set-up and tear-down time for the CVC4 expression subsystem with respect to CVC3. For CVC3, this involves the creation and destruction of a *ValidityChecker* object. For CVC4, this involves the creation and destruction of an *Expression Manager.* CVC4 performs this task almost 10× faster than CVC3.

Same-exprs. CVC4 keeps a unique copy of expression information for each distinct expression. When the client requests an expression node, a lookup in an internal node table is performed to determine whether it already exists; if it does, a pointer to the existing expression data is returned (if not, a pointer to a new, freshly-constructed expression data object is returned). CVC3's behavior is similar. For this stress test, we created a simple expression, then pointed away from it, causing its reference count to drop to 0. CVC4 is 3.5× faster than CVC3 at this simple task. This is largely because CVC3 does garbage-collection eagerly; it thus does collection work when the reference count on the expression data drops to 0, and must construct the expression anew each time it is requested. CVC4's lazy garbage-collection strategy never collects the expression data (as it is dead only a short time) and therefore must never re-construct it anew.

Same-exprs-with-saving. Because the reference count on node data falls to 0 in the previous test, we performed a similar test where the reference count never drops to 0. This removes the advantage of the lazy collection strategy and measures the relative performance CVC4's lookup in its internal node table. Because the same expression is requested each time, the lookup is always successful (after the first time). CVC4's advantage in this test drops to 1.5× speedup over CVC3, less of an advantage but still considerably faster.

Separate-exprs. Finally, the performance of raw expression construction is measured by producing a stream of new expressions. These will each result in a failed lookup in the internal node table and the construction of a new expression structure. Here, CVC4 is again roughly 3.5× faster than CVC3.

As mentioned above, all of the above stress tests were run for a high number of iterations (at least ten million) to get stable performance data on which to base the comparisons. In the *separate-exprs* test, expressions were built over fresh variables, ensuring their distinctness.

We conclude that CVC4's expression subsystem has better performance in setting up and tearing down, in creating already-existing expressions, and in creating not-yet-existing expressions. We further demonstrated one case justifying the use of a lazy garbage collector implemented in CVC4 over the eager one in CVC3.

We performed similar experiments on typical linear arithmetic workloads (drawn from QF_LRA benchmarks in the SMT-LIB library). The time for the expression subsystem operations (not involving any solver machinery) was roughly 1.4× faster in CVC4 than in CVC3, and CVC4 allocated only a quarter of the memory that CVC3 did.

2.2 Theories

CVC4 incorporates newly-designed and implemented decision procedures for its theory of uninterpreted functions, its theory of arithmetic, of arrays, of bitvectors, and of inductive datatypes, based on modern approaches described in the literature. Performance generally is far better than CVC3's (see the note in the conclusion).

In a radical departure from CVC3, CVC4 implements a version of the Simplex method in its implementation of arithmetic [10], whereas CVC3 (and earlier provers in the CVC line) had used an approach based on Fourier-Motzkin variable elimination.

2.3 Proofs

CVC4's proof system is designed to support LFSC proofs [15], and is also designed to have *absolutely zero footprint* in memory and time when proofs are turned off at compile-time.

2.4 Library API

As CVC4 is meant to be used via a library API, there's a clear division between the public, outward-facing interface, and the private, inward-facing one. This is a distinction that wasn't as clear in the previous version; installations of CVC3 required the installation of *all* CVC3 header files, because public headers depended on private ones to function properly. Not so in CVC4, where only a subset of headers declaring public interfaces are installed on a user's machine.

Further, we have decided "to take our own medicine." Our own tools, *including CVC4's parser and main command-line tool,* link against the CVC4 library in the same way that any end-user application would. This helps us ensure that the library API is complete—since if it is not, the command-line CVC4 tool is missing functionality, too, an omission we catch quickly. This is a considerable difference in design from CVC3, where it has often been the case that the API for one or another target language was missing key functionality.

2.5 Theory modularity

Theory objects are designed in CVC4 to be highly *modular:* they do not employ global state, nor do they make any other assumptions that would inhibit their functioning as a client to another decision procedure. In this way, one *Theory* can instantiate and send subqueries to a completely subservient client *Theory* without interfering with the main solver flow.

2.6 Support for concurrency

CVC4's infrastructure has been designed to make the transition to multiprocessor and multicore hardware easy, and we currently have an experimental lemma-sharing portfolio version of CVC4. We intend CVC4 to be a good vehicle for other research ideas in this area as well. In part, the modularity of theories (above) is geared toward this—the absence of global state and the immutability of expression objects clearly makes it easier to parallelize operations. Similarly, the *Theory* API specifically includes the notion of *interruptibility*, so that an expensive operation (*e.g.*, theory propagation) can be interrupted if work in another thread makes it irrelevant. Current work being performed at NYU and U Iowa is investigating different ways to parallelize SMT; the CVC4 architecture provides a good experimental platform for this research, as it does not need to be completely re-engineered to test different concurrent solving strategies.

3 Conclusion

SMT solvers are currently an area of considerable research interest. Barcelogic [5], CVC3 [4], MathSat [6] OpenSMT [7], Yices [9], and Z3 [8] are examples of modern, currently-maintained, popular SMT solvers. OpenSMT and CVC3 are open-source, and CVC3, Yices, and Z3 are the only ones to support all of the defined SMT-LIB logics, including quantifiers.

CVC4 aims to follow in CVC3's footsteps as an open-source theorem prover supporting this wide array of background theories. CVC3 supports all of the background theories defined by the SMT-LIB initiative, and provides proofs and counterexamples upon request; CVC4 aims for full compliance with the new SMT-LIB version 2 command language and backward compatibility with the CVC presentation language.

In this way, CVC4 will be a drop-in replacement for CVC3, with a cleaner and more consistent library API, a more modular, flexible core, a far smaller memory footprint, and better performance characteristics.

The increased performance of CVC4's (over CVC3's) expression subsystem was demonstrated in section 2; CVC4's solving apparatus also performs better than CVC3's. In SMT-COMP 2010 [2], both solvers competed in the QF_LRA division. CVC4 solved more than twice the benchmarks CVC3 did, and for the benchmarks they both solved, CVC4 was almost always faster.

Our goal in CVC4 has been to provide a better-performing implementation of CVC3's feature set, while focusing on flexibility so that it can function as a research vehicle for years to come. Our first goal has been realized for the features that CVC4 currently supports, and we believe this success will continue as we complete support for CVC3's rich set of features. We have been successful in our second as well: a number of internal, complicated, non-intuitive assumptions on which CVC3 rests have been removed in the CVC4 redesign. We have been able to simplify greatly the component interactions and the data structures used in CVC4, making it far easier to document the internals, incorporate new developers, and add support for new features.

References

1. Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the Cooperating Validity Checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of*

the 16^{th} *International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.

2. Clark Barrett, Morgan Deters, Albert Oliveras, and Aaron Stump. SMT-COMP 2010: the 2010 edition of the satisfiability modulo theories competition. `http://www.smtcomp.org/2010/`.
3. Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. pages 187–201. Springer-Verlag, 1996.
4. Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.
5. Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodrguez-Carbonell, and Albert Rubio. The Barcelogic SMT solver. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 294–298. Springer Berlin / Heidelberg, 2008.
6. Roberto Bruttomesso, Alessandro Cimatti, Anders Franzn, Alberto Griggio, and Roberto Sebastiani. The MathSAT 4 SMT solver. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, volume 5123 of *Lecture Notes in Computer Science*, pages 299–303. Springer Berlin / Heidelberg, 2008.
7. Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The OpenSMT solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 150–153. Springer Berlin / Heidelberg, 2010.
8. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
9. Bruno Dutertre and Leonardo de Moura. The YICES SMT solver. `http://yices.csl.sri.com/tool-paper.pdf`.
10. Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.
11. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.
12. Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. pages 175–188. Springer, 2004.
13. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *ANNUAL ACM IEEE DESIGN AUTOMATION CONFERENCE*, pages 530–535. ACM, 2001.
14. Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–57, 1979.
15. Duckki Oe, Andrew Reynolds, and Aaron Stump. Fast and flexible proof checking for SMT. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, pages 6–13, New York, NY, USA, 2009. ACM.
16. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14^{th} *International Conference on Computer Aided Verification (CAV '02)*, volume

2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, July 2002. Copenhagen, Denmark.