From SVC to CVC4

15 Years of Decision Procedures
SAT/SMT Summer School

Clark Barrett

New York University

13 Jun 2011

Outline

1. From SVC to CVC4
   - SVC
   - CVC
   - CVC Lite
   - CVC3
   - CVC4
2. Verification of Low-Level Code
   - Satisfiability Modulo Theories
   - Processing Packets
   - Memory Models
   - Example

SVC

Motivation for a Validity Checker

- Processor Verification via Symbolic Simulation
- Prove that Abstract Specification Machine matches Implementation
- Burch-Dill Commuting Diagram
- Check equality of two big formulas

Burch-Dill Commuting Diagram

$$
\begin{tikzcd}
Abs(q_i) \arrow[r, "F_s"] & \stackrel{?}{=} \\
q_i \arrow[u, "Abs"] \arrow[r, "F_i"] & F_i(q_i) \arrow[u, "Abs"]
\end{tikzcd}
$$

SVC

SVC

- Stanford Validity Checker [Barrett, Dill, & Levitt '96]
- Authors: Clark Barrett, Jeremy Levitt, Aaron Stump, Robert Jones, David Dill
- First source release: 1998

Innovations

- Theory reasoning based loosely on Shostak's method [Shostak '84, Levitt '98]
- Powerful rewriter/simplifier
- Helpful built-in support for backtracking data structures
- Novel decision procedures (e.g. bit-vectors, arrays, records)
- Modular theory solver design

SVC

Applications

- Processor Verification [Levitt & Olukotun '97]
- Specification Checking [Park et al. '98]
- Theorem prover assistance [Heilmann '99]

Headaches

- Shostak's method - too complicated and restrictive
- Equational solvers required to respect restrictive total order
- Boolean reasoning too primitive
- Software architecture - too entangled

CVC

CVC

- Cooperating Validity Checker [Stump, Barrett, & Dill '02]
- Authors: Aaron Stump, Clark Barrett, David Dill, Sergey Berezin, Vijay Ganesh
- First release: 2002

Innovations

- Use of SAT solver (Chaff) for Boolean reasoning [Barrett, Dill, & Stump '02]
- Theory combination framework based on Nelson-Oppen with features of Shostak [Barrett '03]
- Proof production [Stump, Barrett, & Dill '02]

CVC

Applications

- Predicate Abstraction [Das & Dill '02]
- Software Verification (BLAST tool) [Henzinger et al. '03]
- Compiler Validation [Barrett, Goldberg, & Zuck '03]

Headaches

- Software architecture - too entangled

CVC Lite

CVC Lite

- CVC Lite [Barrett & Berezin '04]
- Authors: Clark Barrett, Sergey Berezin, David Dill, Vijay Ganesh
- Additional Contributors: Cristian Cadar, Jake Donham, Yeting Ge, Deepak Goyal, Ying Hu, Sean McLaughlin, Mehul Trivedi, Michael Veksler, Daniel Wichs, Mark Zavislak, Jim Zhuang
- First release: 2004

Innovations

- Theorem-based computation
- Handling of partial functions via TCC's [Berezin et al. '04]
- Mixed integer-real arithmetic (plus some non-linear reasoning)
- Quantifiers
- Predicate sub-typing

CVC Lite

Applications

- Translation validation for compilers [Goldberg, Zuck, & Barrett '04]
- Trusted theorem prover assistance [McLaughlin, Barrett, & Ge '05]
- Hardware equivalence checking at Calypto Systems

Headaches

- Performance
- Software architecture - too entangled

CVC3

CVC3

- CVC3 [Barrett & Tinelli '07]
- Authors: Clark Barrett, Cesare Tinelli, Chris Conway, Morgan Deters, Alexander Fuchs, Yeting Ge, George Hagen, Mina Jeong, Dejan Jovanović, Tim King
- First release: 2007

Innovations

- Enhanced MiniSat Boolean core with proof capability
- New decision procedures (bit-vectors, data types, quantifiers)
- Improved support for non-linear arithmetic
- Extensive support for SMT-LIB and format translation

CVC3

Applications

- Deductive program verification with Why [Filliâtre & Marché '07]
- Symbolic analysis of software at IBM [Chandra, Fink, & Sridharan '09]
- Static analysis of C programs [Conway & Barrett '10]
- Many more...

Headaches

- Performance
- Incompleteness due to non-stably-infinite theories
- Software architecture - too entangled

CVC4

CVC4

- CVC4 [Barrett et al. '11]
- Designers and Authors: Kshitij Bansal, Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Tim King, Dejan Jovanović, Andrew Reynolds, Cesare Tinelli
- First release: 2011

Innovations

- New efficient expression package
- Decentralized and more powerful theory combination techniques (polite theories, care functions) [Jovanović & Barrett '10]
- New state-of-the-art theory implementations (uninterpreted functions, real arithmetic, arrays, bit-vectors)
- Performance-neutral proof production
- Designed to be easily parallelizable

CVC4

Applications

- BMC of Hybrid Systems [King & Barrett '11]
- More to come...

Headaches

- Trying to keep the software architecture from becoming too entangled

CVC4

Applications

- BMC of Hybrid Systems [King & Barrett '11]
- More to come...

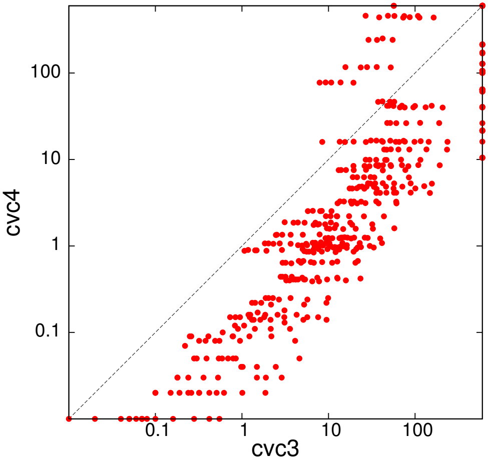
Headaches

- Trying to keep the software architecture from becoming too entangled

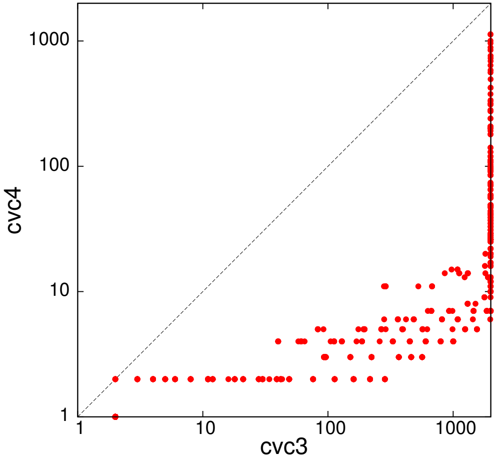
A Sneak Peek at CVC4

- CVC4 vs CVC3 (time and memory)
- CVC4 vs other solvers (time and memory)

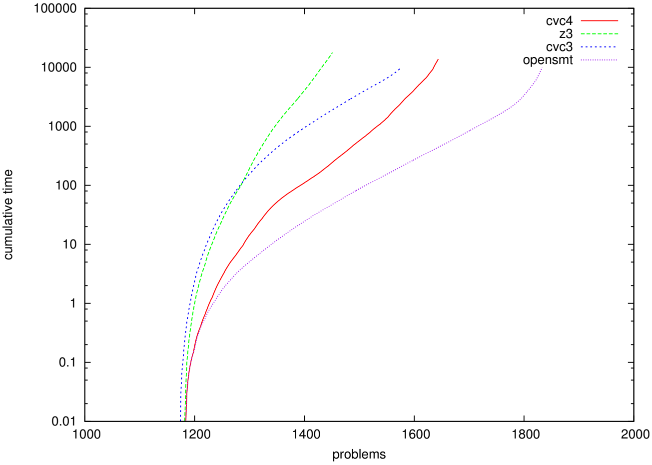
CVC4 vs CVC3 (time)



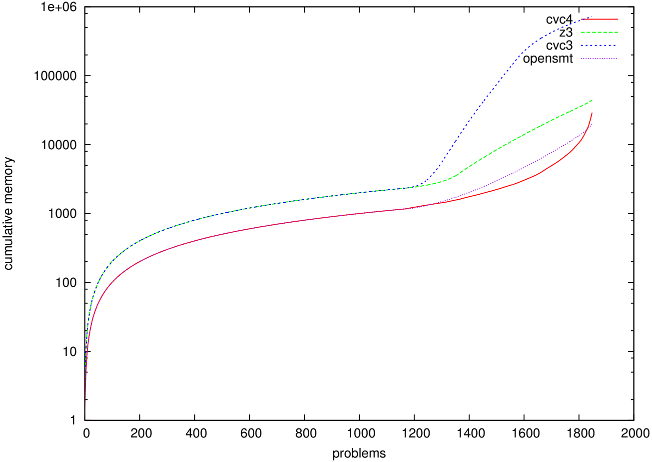
CVC4 vs CVC3 (memory)



Cumulative Time Cactus Plot



Cumulative Memory Cactus Plot



Outline

1. From SVC to CVC4
   - SVC
   - CVC
   - CVC Lite
   - CVC3
   - CVC4
2. Verification of Low-Level Code
   - Satisfiability Modulo Theories
   - Processing Packets
   - Memory Models
   - Example

Satisfiability Modulo Theories

For a theory T, the T*-satisfiability problem* consists of deciding whether there exists a model $\mathcal{A}$ and variable assignment α such that $(\mathcal{A}, \alpha) \models T \cup \varphi$ for a given formula φ.

Theories of Inductive Data Types

An *inductive data type* (IDT) defines one or more *constructors*, and possibly also *selectors* and *testers*.

Example: *list* **of** *int*

- Constructors: $cons : (int, list) \rightarrow list$, $null : list$
- Selectors: $car : list \rightarrow int$, $cdr : list \rightarrow list$
- Testers: is_cons, is_null

The *first order theory* of a inductive data type associates a function symbol with each constructor and selector and a predicate symbol with each tester.

Example: $\forall\, x : list.\, (x = null \vee \exists\, y : int, z : list.\, x = cons(y, z))$

Theories of Inductive Data Types

An *inductive data type* (IDT) defines one or more *constructors*, and possibly also *selectors* and *testers*.

Example: *list* **of** *int*

- Constructors: $cons : (int, list) \rightarrow list$, $null : list$
- Selectors: $car : list \rightarrow int$, $cdr : list \rightarrow list$
- Testers: *is_cons*, *is_null*

The *first order theory* of a inductive data type associates a function symbol with each constructor and selector and a predicate symbol with each tester.

Example: $\forall x : list. (x = null \vee \exists y : int, z : list. x = cons(y, z))$

For IDTs with a single constructor, a conjunction of literals is decidable in polynomial time [Oppen '80].

Theories of Inductive Data Types

An *inductive data type* (IDT) defines one or more *constructors*, and possibly also *selectors* and *testers*.

Example: *list* **of** *int*

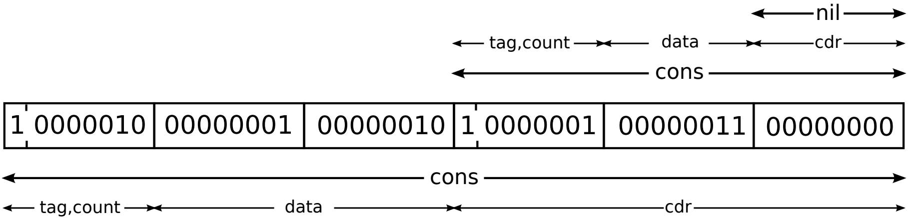
- Constructors: $cons : (int, list) \rightarrow list$, $null : list$
- Selectors: $car : list \rightarrow int$, $cdr : list \rightarrow list$
- Testers: is_cons, is_null

The *first order theory* of a inductive data type associates a function symbol with each constructor and selector and a predicate symbol with each tester.

Example: $\forall\, x : list.\, (x = null \vee \exists\, y : int, z : list.\, x = cons(y, z))$

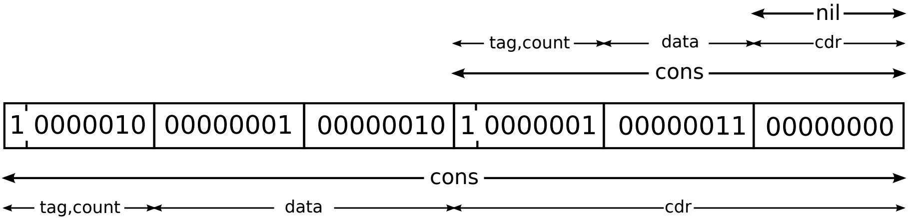
For more general IDTs, the problem is NP complete, but reasonbly efficient algorithms exist in practice [Barrett et al. '07].

Processing Packets



Network packets are highly structured

Processing Packets



Network packets are highly structured
but usually processed with low-level bit-twiddling code

```
while( (n = *p++) & 0x80 ) {
    p += n & 0x7f;
}
```

Processing Packets

One solution: packet-processing DSLs
(e.g., binpac, Melange, Morpheus, Prolac)

```
type List =
  cons {
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }
| nil {
    tag:8 = 0x00
  }
```

Processing Packets

One solution: packet-processing DSLs
(e.g., binpac, Melange, Morpheus, Prolac)

```
type List =
  cons {
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }
| nil {
    tag:8 = 0x00
  }
```

- High level
- Type safe

Processing Packets

One solution: packet-processing DSLs
(e.g., binpac, Melange, Morpheus, Prolac)

```
type List =
  cons {
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }
| nil {
    tag:8 = 0x00
  }
```

- High level
- Type safe
- Slower than C
- Need to rewrite existing code

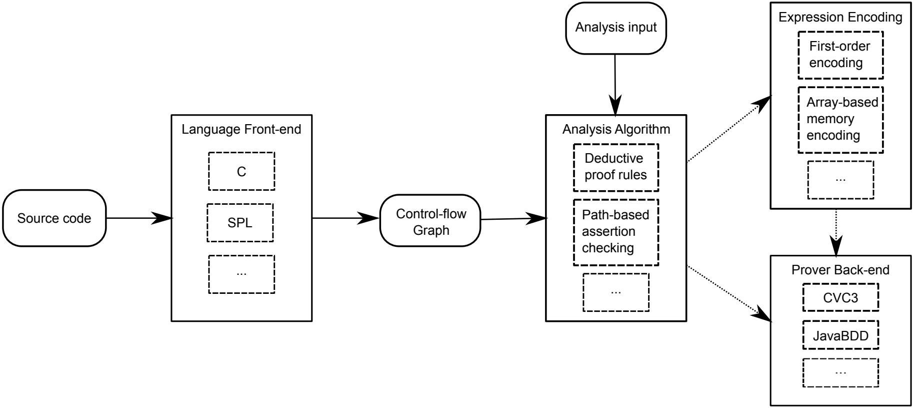
Packet Types as Specification

Instead of synthesizing a performant implementation,
let's use packet types as the basis of a specification

```
while( (n = *p++) & 0x80 ) {
    assert( isCons(prev(p)) );
    p += n & 0x7f;
    assert( p == cdr(prev(p)) );
}
```

We can use bit-precise reasoning to prove that the code satisfies the assertions using CASCADE.

Cascade Verification Framework



Cascade/C

- High-precision verification of program paths
- Intended for use in a *multi-stage analysis*
- Path is defined and assertions are injected using an XML control file

Cascade/C

swap.ctrl:

```
<controlFile>
  <sourceFile name="swap.c" id="1" />
  <run>
    <startPosition fileId="1" line="1" />
    <endPosition fileId="1" line="5">
      <assert><![CDATA[
         orig(*x)==*y && orig(*y)==*x
      ]]></assert>
    </endPosition>
  </run>
</controlFile>
```

swap.c:

```
void swap(int*x, int*y) {
  *x = *x + *y;
  *y = *x - *y;
  *x = *x - *y;
}
```

```
*x = *x + *y;
*y = *x - *y;
*x = *x - *y;
assert( orig(*x)==*y && orig(*y)==*x );
```

CVC3 Encoding

- Encode verification conditions as SMT instances
- Use CVC3 SMT solver to decide validity
- CVC3 includes theories for:
  - Arrays
  - Uninterpreted functions
  - Bit vectors
  - Inductive datatypes
- Connect the high-level assertions and the low-level code by generating:
  - An inductive datatype
  - Functions mapping datatype values to arrays of bytes
  - Encode program semantics using bit vectors

CVC3 Encoding

```
type List =
  cons {
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }
| nil {
    tag:8 = 0x00
  }
```

CVC3 Encoding

```
ptrType : BITVECTOR(N);
byteType : BITVECTOR(8);
memType : ARRAY ptrType OF byteType;

DATATYPE
List =
  cons( tag: BITVECTOR(1),
        len: BITVECTOR(7),
        data: memType,
        cdr: List )
| nil( tag: BITVECTOR(8) )
| undefined;
END;

toList : (memType, ptrType) -> List;

∀ m:memType, i:ptrType.
  isNil(toList(m,i)) ⟺ m[i] = 0;
∀ m: memType, i: ptrType.
  isCons(toList(m,i)) ⟺ m[i][7] = 1;
∀ m: memType, i: ptrType.
  isCons(toList(m, i)) ⟹
      cdr(toList(m,i)) = toList(m,i+len(toList(m,i))+1);
etc...
```

Verification Condition Generation

```
n = *p++;
assume( (n & 0x80) != 0 );
assert( isCons(prev(p)) );
```

becomes

$$\frac{\begin{array}{c} m_1 = m_0[\&\mathrm{n} \mapsto m_0[m_0[\&\mathrm{p}]]] \\ m_2 = m_1[\&\mathrm{p} \mapsto m_1[\&\mathrm{p}] + 1] \\ m_2[\&\mathrm{n}] \ \& \ 0\mathrm{x}80 \neq 0\mathrm{x}00 \end{array}}{\mathit{isCons}(\mathit{toList}(m_2, m_0[\&\mathrm{p}]))}$$

Memory Models

- "Flat" memory model
  - Memory is one big array:

$$m_1 = m_0[\&\mathrm{n} \mapsto m_0[m_0[\&\mathrm{p}]]]$$
$$m_2 = m_1[\&\mathrm{p} \mapsto m_1[\&\mathrm{p}] + 1]$$

  - No "frame rule" is implied.
    - E.g., the following isn't necessarily valid:

```
{ toList(q)==cdr(p) }
  i++
{ toList(q)==cdr(p) }
```

    - We can't rule out `&i` being reachable if `toList` is unrolled enough times.
  - Detailed non-aliasing assumptions have to be added by hand

Memory Models

- "Flat" memory model
  - Memory is one big array:

$$m_1 = m_0[\&\mathrm{n} \mapsto m_0[m_0[\&\mathrm{p}]]]$$
$$m_2 = m_1[\&\mathrm{p} \mapsto m_1[\&\mathrm{p}] + 1]$$

  - No "frame rule" is implied.
    - E.g., the following isn't necessarily valid:

```
{ toList(q)==cdr(p) }
  i++
{ toList(q)==cdr(p) }
```

    - We can't rule out `&i` being reachable if `toList` is unrolled enough times.
  - Detailed non-aliasing assumptions have to be added by hand
  - And they don't help much

Memory Models

- Burstall model [Burstall '72, Bornat '00]
  - A separate memory array for each static type:

$$
\begin{aligned}
m'_{char} &= m_{char}[\&\mathrm{n} \mapsto m_{char}[m_{char*}[\&\mathrm{p}]]] \\
m'_{char*} &= m_{char*}[\&\mathrm{p} \mapsto m_{char*}[\&\mathrm{p}] + 1]
\end{aligned}
$$

  - Can't handle safe dynamic casts
  - Can't handle promiscuous pointer manipulation

Memory Models

- Burstall model [Burstall '72, Bornat '00]
  - A separate memory array for each static type:

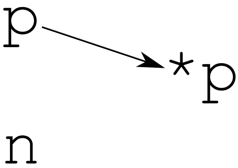
$$m'_{char} = m_{char}[\&\mathrm{n} \mapsto m_{char}[m_{char*}[\&\mathrm{p}]]]$$
$$m'_{char*} = m_{char*}[\&\mathrm{p} \mapsto m_{char*}[\&\mathrm{p}] + 1]$$

  - Can't handle safe dynamic casts
  - Can't handle promiscuous pointer manipulation
  - Which is exactly what packet processing is

Partitioning the Heap

An "in between" model, based on separation analysis
[Hubert & Marché '07, Rakamaric & Hu '09]

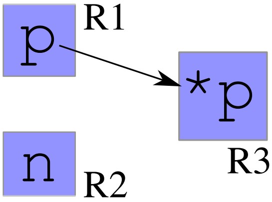
- Memory is partitioned into disjoint *regions*.
- Every pointer expression is associated with a region



Partitioning the Heap

An "in between" model, based on separation analysis
[Hubert & Marché '07, Rakamaric & Hu '09]

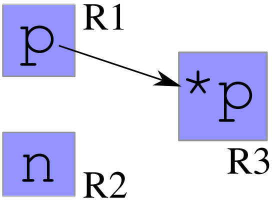
- Memory is partitioned into disjoint *regions*.
- Every pointer expression is associated with a region



Partitioning the Heap

An "in between" model, based on separation analysis
[Hubert & Marché '07, Rakamaric & Hu '09]

- Memory is partitioned into disjoint *regions*.
- Every pointer expression is associated with a region



- Each region can be represented by a separate "memory"

Partitioning the Heap

Flat:

$$\frac{\begin{array}{c} m_1 = m_0[\texttt{\&n} \mapsto m_0[m_0[\texttt{\&p}]]] \\ m_2 = m_1[\texttt{\&p} \mapsto m_1[\texttt{\&p}] + 1] \\ m_2[\texttt{\&n}] \ \texttt{\&} \ \texttt{0x80} \neq \texttt{0x00} \end{array}}{\mathit{isCons}(\mathit{toList}(m_2, m_0(\texttt{\&p})))}$$

Partitioning the Heap

Partitioned:

$$\frac{\begin{array}{c} m'_{\mathrm{n}} = m_{\mathrm{n}}[\&\mathrm{n} \mapsto m_{*\mathrm{p}}[m_{\mathrm{p}}[\&\mathrm{p}]]] \\ m'_{\mathrm{p}} = m_{\mathrm{p}}[\&\mathrm{p} \mapsto m_{\mathrm{p}}[\&\mathrm{p}] + 1] \\ m'_{\mathrm{n}}[\&\mathrm{n}]\ \&\ \mathtt{0x80} \neq \mathtt{0x00} \end{array}}{\mathit{isCons}(\mathit{toList}(m_{*\mathrm{p}}, m_{\mathrm{p}}[\&\mathrm{p}]))}$$

Partitioning the Heap

Partitioned:

$$\frac{\begin{array}{c} m'_{\mathrm{n}} = m_{\mathrm{n}}[\&\mathrm{n} \mapsto m_{*\mathrm{p}}[m_{\mathrm{p}}[\&\mathrm{p}]]] \\ m'_{\mathrm{p}} = m_{\mathrm{p}}[\&\mathrm{p} \mapsto m_{\mathrm{p}}[\&\mathrm{p}] + 1] \\ m'_{\mathrm{n}}[\&\mathrm{n}] \ \& \ 0\mathrm{x}80 \neq 0\mathrm{x}00 \end{array}}{\mathit{isCons}(\mathit{toList}(m_{*\mathrm{p}}, m_{\mathrm{p}}[\&\mathrm{p}]))}$$

- Separation creates a "frame" around datatype values
- Makes hard problems easy and easy problems trivial
- The verification condition is sound if the partition is sound

"Real World" Example: Encoded Domain Name

```
type Dn =
  label {
    tag:2 = 0b00,
    len:6 != 0b000000,
    name:u_char[len],
    rest:Dn
  }
| indirect {
    tag:2 = 0b11,
    offset:14
  }
| nullt {
    tag:8 = 0x00
  }
```

```
#define NS_CMPRSFLAGS (0xc0)
int ns_name_skip(const u_char **ptrptr, const u_char *eom) {
  { allocated(*ptrptr, eom) }
  const u_char *cp; u_int n;
  cp = *ptrptr;
  { @invariant: cp <= eom =>
                cp + sizeOfDn(cp) = init(cp) + sizeOfDn(init(cp)) }
  while (cp < eom && (n = *cp++) != 0) {
    switch (n & NS_CMPRSFLGS) {
      case 0: /* normal case, n == len */
        { isLabel(prev(cp)) }
        cp += n;
        { rest(prev(cp)) = toDn(cp) }
        continue;
      case NS_CMPRSFLGS: /* indirection */
        { isIndirect(prev(cp)) }
        cp++; break;
      default: /* illegal type */
        __set_errno (EMSGSIZE); return (-1);
    }
    break;
  }
  if (cp > eom) { __set_errno (EMSGSIZE); return (-1); }
  { cp = eom _ cp = init(cp) + sizeOfDn(init(cp)) }
  *ptrptr = cp;
  return (0);
}
```

Experimental results

- Verification times for `ns_name_skip`.
- 30 LOC, 4 assertions + a loop invariant

Name	Lines	Time (seconds) Flat	Part.
Init	5–12	0.34	0.03
Case 0 (1)	12-16	13.94	0.05
Case 0 (2)	12-28	33.42	0.06
Case 0 (3)	12-19	*	0.12
Case 0xc0 (1)	12–14, 20–21	6.14	0.04
Case 0xc0 (2)	12–14, 20–23, 30, 34	*	0.07
Term (1)	12, 30, 34	0.63	0.06
Term (2)	12, 30, 34	*	0.05

Final Thoughts

15 years of checking formulas

- SMT has come a long way in last 15 years
- Dramatic advances in theory and practice
- Explosion of application areas

Lessons

- Balancing high-performance and software flexibility is a challenge
- Modularity and solid theoretical foundations can help
- But in a rapidly advancing area, may have to reimplement every few years anyway

CVC4 is coming

- Goals: open source, high-performance, full-featured SMT solver
- Contributions and collaborations welcome after first release

References

[Bar03] Clark W. Barrett. *Checking Validity of Quantifier-Free Formulas in Combinations of First-Order Theories*. PhD thesis, Stanford University, January 2003. Stanford, California

[BB04] 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

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the* 23^{rd} *International Conference on Computer Aided Verification (CAV '11)*, Lecture Notes in Computer Science. Springer, July 2011. Snowbird, Utah, to appear

[BDL96] Clark W. Barrett, David L. Dill, and Jeremy R. Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the* 1^{st} *International Conference on Formal Methods In Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, November 1996. Palo Alto, California

References

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Proceedings of the* 14th *International Conference on Computer Aided Verification (CAV '02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 236–249. Springer-Verlag, July 2002. Copenhagen, Denmark

[BGZ03] Clark Barrett, Benjamin Goldberg, and Lenore Zuck. Run-time validation of speculative optimizations using CVC. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Proceedings of the* 3rd *International Workshop on Run-time Verification (RV '03)*, volume 89(2) of *Electronic Notes in Theoretical Computer Science*, pages 89–107. Elsevier, October 2003. Boulder, Colorado

[BST07] Clark Barrett, Igor Shikanian, and Cesare Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19th *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

References

[BBS+05] Sergey Berezin, Clark Barrett, Igor Shikanian, Marsha Chechik, Arie Gurfinkel, and David L. Dill. A practical approach to partial functions in CVC Lite. In Wolfgang Ahrendt, Peter Baumgartner, Hans de Nivelle, Silvio Ranise, and Cesare Tinelli, editors, *Selected Papers from the Workshops on Disproving and the Second International Workshop on Pragmatics of Decision Procedures (PDPAR '04)*, volume 125(3) of *Electronic Notes in Theoretical Computer Science*, pages 13–23. Elsevier, July 2005. Cork, Ireland

[Bor00] Richard Bornat. Proving pointer programs in hoare logic. In Roland Backhouse and Jos Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer Berlin / Heidelberg, 2000

[Bur72] R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 1972

[CFS09] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 363–374, New York, NY, USA, 2009. ACM

[CB10] Christopher L. Conway and Clark Barrett. Verifying low-level implementations of high-level datatypes. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV '10)*, volume 6174 of *Lecture Notes in Computer Science*, pages 306–320. Springer, July 2010. Edinburgh, Scotland

References

[DD02] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*. Springer-Verlag, November 2002

[FM07] Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer Berlin / Heidelberg, 2007

[GZB05] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In J. Knoop, G.C. Necula, and W. Zimmermann, editors, *Proceedings of the* 3^{rd} *International Workshop on Compiler Optimization meets Compiler Verificaiton (COCV '04)*, volume 132(1) of *Electronic Notes in Theoretical Computer Science*, pages 53–71. Elsevier, May 2005. Barcelona, Spain

[Hei99] Søren T. Heilmann. *Proof Support for Duration Calculus*. PhD thesis, Technical University of Denmark, 1999

[HJMS03] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Grgoire Sutre. Software verification with blast. In Thomas Ball and Sriram Rajamani, editors, *Model Checking Software*, volume 2648 of *Lecture Notes in Computer Science*, pages 624–624. Springer Berlin / Heidelberg, 2003

[HM07] T. Hubert and C. Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV)*, pages 81–93, March 2007

References

[JB10b] Dejan Jovanović and Clark Barrett. Sharing is caring. In *Proceedings of the* 8^{th} *International Workshop on Satisfiability Modulo Theories (SMT '10)*, July 2010. Edinburgh, Scotland

[JB10a] Dejan Jovanović and Clark Barrett. Polite theories revisited. In Christian G. Fermüller and Andrei Voronkov, editors, *Proceedings of the* 17^{th} *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR '10)*, volume 6397 of *Lecture Notes in Computer Science*, pages 402–416. Springer, October 2010. Yogyakarta, Indonesia

[KB11] Tim King and Clark Barrett. Exploring and categorizing error spaces using bmc and smt. In *Proceedings of the* 9^{th} *International Workshop on Satisfiability Modulo Theories (SMT '11)*, July 2011. Snowbird, Utah, to appear

[Lev99] J. Levitt. *Formal Verification Techniques for Digital Systems*. PhD thesis, Stanford University, 1999

[JK97] Jeremy Levitt and Kunle Olukotun. Verifying Correct Pipeline Implentation for Microprocessors. In *International Conference on Computer Aided Design*, San Jose, CA, November 1997. IEEE Computer Society Press

[MBG06] Sean McLaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In Alessandro Armando and Alessandro Cimatti, editors, *Proceedings of the* 3^{rd} *Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05)*, volume 144(2) of *Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, January 2006. Edinburgh, Scotland

References

[Opp80] D. C. Oppen. Reasoning about recursively defined data structures. *JACM*, 27(3):403–411, July 1980

[PSH+98] David Y.W. Park, Jens U. Skakkebæk, Mats P.E. Heimdahl, Barbara J. Czerny, and David L. Dill. Checking Properties of Safety Critical Specifications Using Efficient Decision Procedures. In *FMSP'98: Second Workshop on Formal Methods in Software Practice*, pages 34–43, March 1998

[RH09] Zvonimir Rakamarić and Alan Hu. A scalable memory model for low-level code. In Neil Jones and Markus Mller-Olm, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *Lecture Notes in Computer Science*, pages 290–304. Springer Berlin / Heidelberg, 2009

[Sho84] R. Shostak. Deciding combinations of theories. *Journal of the Association for Computing Machinery*, 31(1):1–12, 1984

[SBD02a] 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

[SBD02b] Aaron Stump, Clark W. Barrett, and David L. Dill. Producing proofs from an arithmetic decision procedure in elliptical LF. In Frank Pfenning, editor, *Proceedings of the* 3*rd* *International Workshop on Logical Frameworks and Meta-Languages (LFM '02)*, volume 70(2) of *Electronic Notes in Theoretical Computer Science*, pages 29–41. Elsevier, July 2002. Copenhagen, Denmark