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
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

\[
\begin{align*}
Abs(q_i) & \xrightarrow{F_s} ? \\
q_i & \xrightarrow{F_i} F_i(q_i)
\end{align*}
\]
### 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
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

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]
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** [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 Calypso Systems

### Headaches
- Performance
- Software architecture - too entangled
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
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 [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
Applications

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

Headaches

- Trying to keep the software architecture from becoming too entangled
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)

![Graph showing the comparison between CVC4 and CVC3 in terms of time. The x-axis represents CVC3 time, and the y-axis represents CVC4 time. The points on the graph indicate the relative performance of CVC4 compared to CVC3. The diagonal line represents equal performance between the two systems.]
CVC4 vs CVC3 (memory)
Cumulative Time 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
For a theory $T$, the $T$-satisfiability problem consists of deciding whether there exists a model $\mathcal{A}$ and variable assignment $\alpha$ such that $(\mathcal{A}, \alpha) \models T \cup \varphi$ for a given formula $\varphi$. 
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:** \( \text{cons} : (\text{int}, \text{list}) \to \text{list}, \text{null} : \text{list} \)
- **Selectors:** \( \text{car} : \text{list} \to \text{int}, \text{cdr} : \text{list} \to \text{list} \)
- **Testers:** \( \text{is} \_\text{cons}, \text{is} \_\text{null} \)

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:** \( \forall x : \text{list}. (x = \text{null} \lor \exists y : \text{int}, z : \text{list}. x = \text{cons}(y, z)) \)
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) → list, null : list`
- **Selectors:** `car : list → int, cdr : list → 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:** \( \forall x : list. (x = null \lor \exists y : int, z : list. x = cons(y, z)) \)

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:** \( \text{cons} : (\text{int}, \text{list}) \rightarrow \text{list}, \text{null} : \text{list} \)
- **Selectors:** \( \text{car} : \text{list} \rightarrow \text{int}, \text{cdr} : \text{list} \rightarrow \text{list} \)
- **Testers:** \( \text{is} \_\text{cons}, \text{is} \_\text{null} \)

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

**Example:** \( \forall \, x : \text{list}. \ (x = \text{null} \lor \exists \, y : \text{int}, z : \text{list}. \ x = \text{cons}(y, z)) \)

For more general IDTs, the problem is NP complete, but reasonably efficient algorithms exist in practice [Barrett et al. '07].
Network packets are highly structured
Network packets are highly structured but usually processed with low-level bit-twiddling code.

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

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

```plaintext
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)

```haskell
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)

\[
\text{type \hspace{0.5em} List} = \\
\text{cons \hspace{1em} \{} \hspace{0.5em} \\
\hspace{1.5em} \text{tag:} 1 = 0b1, \hspace{1em} \\
\hspace{1.5em} \text{count:} 7, \hspace{1em} \\
\hspace{1.5em} \text{data: u_char[count]}, \hspace{1em} \\
\hspace{1.5em} \text{cdr: List} \hspace{1em} \} \\
\text{| nil \hspace{1em} \{} \hspace{0.5em} \\
\hspace{1.5em} \text{tag:} 8 = 0x00 \hspace{1em} \}
\]

- 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

```c
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

Source code

Language Front-end
- C
- SPL
- ...

Control-flow Graph

Analysis input
- Analysis Algorithm
  - Deductive proof rules
  - Path-based assertion checking
  - ...

Expression Encoding
- First-order encoding
- Array-based memory encoding
  - ...

Prover Back-end
- CVC3
- JavaBDD
  - ...

Clark Barrett (New York University)
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
swap.c:

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

swap.ctrl:

```xml
<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>
```

```c
*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

define type List =
  cons { 
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }

| nil { 
  tag:8 = 0x00 
}
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;

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

\[
\begin{align*}
    n &= \ast p++; \\
    \text{assume} &\left((n \& 0x80) \neq 0\right); \\
    \text{assert} &\left(\text{isCons}\left(\text{prev}\left(p\right)\right)\right); \\
\end{align*}
\]

becomes

\[
\begin{align*}
    m_1 &= m_0[&n \mapsto m_0[m_0[&p]]] \\
    m_2 &= m_1[&p \mapsto m_1[&p] + 1] \\
    m_2[&n] \& 0x80 \neq 0x00 \\
\hline
    \text{isCons}\left(\text{toList}(m_2, m_0[&p])\right)
\end{align*}
\]
Memory Models

- "Flat" memory model
  - Memory is one big array:
    
    \[
    m_1 = m_0[n \mapsto m_0[m_0[p]]] \\
    m_2 = m_1[p \mapsto m_1[p] + 1]
    \]

- No "frame rule" is implied.
  - E.g., the following isn’t necessarily valid:
    
    \[
    \{ \text{toList}(q) == \text{cdr}(p) \} \\
    \quad i++ \\
    \{ \text{toList}(q) == \text{cdr}(p) \}
    \]

    - 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:
    
    \[ m_1 = m_0[&n \mapsto m_0[m_0[&p]]] \]
    \[ m_2 = m_1[&p \mapsto m_1[&p] + 1] \]
  
  - No “frame rule” is implied.
    - E.g., the following isn’t necessarily valid:
      
      \[
      \begin{align*}
      \{ & \ \text{toList}(q) == \text{cdr}(p) \} \\
      & i++ \\
      \{ & \ \text{toList}(q) == \text{cdr}(p) \}
      \end{align*}
      \]
    
    - 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:
    
    \[
    m'_{\text{char}} = m_{\text{char}}[\&n \mapsto m_{\text{char}}[m_{\text{char}}[\&p]]]
    
    m'_{\text{char}^*} = m_{\text{char}^*}[\&p \mapsto m_{\text{char}^*}[\&p] + 1]
    \]
  
  - Can’t handle safe dynamic casts
  - Can’t handle promiscuous pointer manipulation
Burstall model [Burstall ’72, Bornat ’00]

A separate memory array for each static type:

\[m'_{\text{char}} = m_{\text{char}}[\&n \mapsto m_{\text{char}}[m_{\text{char}^*}[\&p]]]\]

\[m'_{\text{char}^*} = m_{\text{char}^*}[\&p \mapsto m_{\text{char}^*}[\&p] + 1]\]

- 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

\[ p \rightarrow \ast p \]

\[ n \]
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.

\[ \text{Memory is partitioned into disjoint regions.} \]

\[ \text{Every pointer expression is associated with a region.} \]
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:

\[ m_1 = m_0[\&n \mapsto m_0[m_0[\&p]]] \]
\[ m_2 = m_1[\&p \mapsto m_1[\&p] + 1] \]
\[ m_2[\&n] \& 0x80 \neq 0x00 \]
\[ \text{isCons(toList}(m_2, m_0[\&p])) \]
Partitioning the Heap

Partitioned:

\[
\begin{align*}
    m'_n &= m_n[\&n \leftrightarrow m_p[m_p[\&p]]] \\
    m'_p &= m_p[\&p \leftrightarrow m_p[\&p] + 1] \\
    m'_n[\&n] &\neq 0x00
\end{align*}
\]

\[
isCons\left(\text{toList}(m'_p, m_p[\&p])\right)
\]
Partitioning the Heap

Partitioned:

\[
\begin{align*}
  m'_n &= m_n[\&n \mapsto m*p[m_p[\&p]]] \\
  m'_p &= m_p[\&p \mapsto m_p[\&p] + 1] \\
  m'_n[\&n] &\neq 0x80
\end{align*}
\]

\[isCons(toList(m*_p, m_p[\&p]))\]

- 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

define 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 \texttt{ns\_name\_skip}.
- 30 LOC, 4 assertions + a loop invariant

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Flat</td>
</tr>
<tr>
<td><strong>INIT</strong></td>
<td>5–12</td>
<td>0.34</td>
</tr>
<tr>
<td><strong>CASE 0 (1)</strong></td>
<td>12-16</td>
<td>13.94</td>
</tr>
<tr>
<td><strong>CASE 0 (2)</strong></td>
<td>12-28</td>
<td>33.42</td>
</tr>
<tr>
<td><strong>CASE 0 (3)</strong></td>
<td>12-19</td>
<td>*</td>
</tr>
<tr>
<td><strong>CASE 0xc0 (1)</strong></td>
<td>12–14, 20–21</td>
<td>6.14</td>
</tr>
<tr>
<td><strong>CASE 0xc0 (2)</strong></td>
<td>12–14, 20–23, 30, 34</td>
<td>*</td>
</tr>
<tr>
<td><strong>TERM (1)</strong></td>
<td>12, 30, 34</td>
<td>0.63</td>
</tr>
<tr>
<td><strong>TERM (2)</strong></td>
<td>12, 30, 34</td>
<td>*</td>
</tr>
</tbody>
</table>
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


References


References


References


References


References


