Global Constraints in Software Testing Applications

Arnaud Gotlieb
Simula Research Laboratory
Norway
The Certus Centre

Software Validation and Verification

Hosted by SIMULA

Established and awarded SFI in Oct. 2011

duration: 8 years

www.certus-sfi.no
Agenda

I. Software Testing

II. Optimal Test Suite Reduction

III. Automatic Test Case Generation

IV. Conclusions and Perspectives
Software Testing

Informal World

User Requirements

Logical and Mathematical World

Constraint Model

Real World

Test cases

Spec / Model

Software Under Test

Verdict: Pass or Fail

Oracle

Results
Software Testing

Software test preparation is a **cognitively complex task**:

- Requires to understand both model and code to create interesting test cases;
- Program’s input space is usually very large (sometimes unbounded);
- Complex software (e.g., implementing ODEs or PDEs) yields to complex bugs;
- Test oracles are hard to define (non-testable programs);

**Not easily amenable to automation:**

- Automatic test data generation is undecideable in the general case!
- Exploring the input space yields to combinatorial explosion;
- Fully automated oracles are usually not available;
How software testing differs from other program verification techniques?

- **Static analysis** finds simple faults (division-by-zero, overflows, ...) at compile-time, while **software testing** finds functional faults at run-time (P returns 3 while 2 was expected).

- **Program proving** aims at formally proving mathematical invariants, while **software testing** evaluates the program in its execution environment.

- **Model-checking** explores paths of a model of the software under test for checking temporal properties or finding counter-examples, while **software testing** is based on program executions.
Some Hot Research Topics in Software Testing

- Automatic test case generation

  Find test cases to exercise specific behaviors, to execute specific code locations, to cover some test objectives (e.g., all-statements, all-k-paths)

- Test suite reduction, test suite prioritization, test execution scheduling

- Robustness and performance testing

- Testing complex code (e.g., floating-point and iterative computations)

Our thesis: Global constraints can efficiently tackle these problems!

(High-level primitives with specialised filtering algorithms)
Optimal Test Suite Reduction
Optimal TSR: the core problem

Optimal TSR: find a minimal subset of TC such that each F is covered at least once (Practical importance but NP-hard problem!) – An instance of Minimum Set Cover
The \textit{nvalue} global constraint

\[ nvalue(n, v) \]

Where:

- \textit{n} is an FD\_variable
- \textit{v} = (v_1, \ldots, v_k) is a vector of FD\_variables

\[ nvalue(n, v) \text{ holds iff } n = \text{card}\left( \{v_i\}_{i \text{ in } 1..k} \right) \]

Introduced in [Pachet and Roy’99], first filtering algorithm in [Beldiceanu’01]
Solution existence for \textit{nvalue} is NP-hard [Bessiere et al. ‘04]
Optimal TSR: CP model with nvalue (1)

\[ F_1 \text{ in } \{1, 2, 6\}, \ F_2 \text{ in } \{3, 4\}, \ F_3 \text{ in } \{2, 5\} \]
\[ \text{nvalue}(\text{MaxNvalue}, (F_1, F_2, F_3)), \]
\[ \text{label}(\text{minimize}(\text{MaxNvalue})) \]

/* branch-and-bound search among feasible solutions */
The global_cardinality constraint

\[ gcc(t, d, v) \]

Where

\[ t = (t_1, ..., t_N) \] is a vector of N variables, each \( t_j \) in \( Min_j .. Max_j \)

\[ d = (d_1, ..., d_k) \] is a vector of k values

\[ v = (v_1, ..., v_k) \] is a vector of k variables, each \( v_i \) in \( Min_i .. Max_i \)

\[ gcc(t, d, v) \] holds iff \[ \forall i \ in 1 .. k, \]

\[ v_i = card\{ t_j = d_i \}_{j \ in 1 .. N} \]

Filtering algorithms for \( gcc \) are based on max flow computations in a network flow [Regin AAAI’96]
Example

\[ \text{gcc}(F_1, F_2, F_3), (1,2,3,4,5,6), (V_1,V_2,V_3,V_4,V_5,V_6)) \]
means that:

In a solution of TSR:

TC_1 covers exactly V_1 requirements in (F_1, F_2, F_3)
TC_2 " " V_2 " "
TC_3 " " V_3 " "
...

Where F_1, F_2, F_3, V_1, V_2, V_3, ... denote finite-domain variables

F_1 in \{1, 2, 6\}, F_2 in \{3, 4\}, F_3 in \{2, 5\}
V_1 in \{0, 1\}, V_2 in \{0, 2\}, V_3 in \{0, 1\}, V_4 in \{0, 1\}, V_5 in \{0, 1\}, V_6 in \{0, 1\}

Here, for example,  \( V_1 = 1, V_2 = 2, V_3 = 1, V_4 = 0, V_5 = 0, V_6 = 0 \) is a feasible solution

But, not an optimal one!
Optimal TSR: CP model with two gcc

\[ F_1 \text{ in } \{1, 2, 6\}, \ F_2 \text{ in } \{3, 4\}, \ F_3 \text{ in } \{2, 5\} \]

\[ \text{gcc}( (F_1, F_2, F_3), (1,2,3,4,5,6), (V_1, V_2, V_3, V_4, V_5, V_6)) \]

\[ \text{gcc}((V_1, V_2, V_3, V_4, V_5, V_6), (0-_), (\text{Max0Req-} )) \]

label(maximize(\text{Max0Req}))

/* search heuristics by enumerating the Vi first */
Introducing model pre-treatment

\[ F_1 \text{ in } \{1, 2, 6\} \rightarrow F_1 = 2 \quad \text{as } \text{cov}(TC_1) = \text{cov}(TC_6) \subset \text{cov}(TC_2) \]
withdraw TC_1 and TC_6

F_3 \text{ is covered } \rightarrow \text{withdraw TC}_5

F_2 \text{ in } \{3, 4\} \rightarrow \text{e.g., } F_2 = 3, \text{ withdraw TC}_4

Three such pre-treatment rules have been identified and can be included to simplify the problem

But, they are currently statically applied!
3. Optimal TSR: CP model Mixt (3)

\[
F_1 \text{ in } \{1, 2, 6\}, \ F_2 \text{ in } \{3, 4\}, \ F_3 \text{ in } \{2, 5\}
\]
gcc( (F_1, F_2, F_3), (1,2,3,4,5,6), (V_1, V_2, V_3, V_4, V_5, V_6) ),
nvalue(MaxNvalue, (V_1, V_2, V_3, V_4, V_5, V_6))

label(minimize(MaxNvalue))

/* + pre-treatment + labelling heuristics based on max */
Model comparison on random instances
(Reduced Test Suite percentage in 30sec of search)
Model comparison on random instances (CPU time to find a global optimum)

<table>
<thead>
<tr>
<th>Requirements</th>
<th>TD1</th>
<th>TD2</th>
<th>TD3</th>
<th>TD4</th>
<th>TD5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test cases</td>
<td>70</td>
<td>100</td>
<td>100</td>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td>Density</td>
<td>8</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>8</td>
</tr>
</tbody>
</table>
Comparison with other approaches
(Reduced Test Suite percentage in 60 sec)

<table>
<thead>
<tr>
<th>Requirements</th>
<th>TD1</th>
<th>TD2</th>
<th>TD3</th>
<th>TD4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test cases</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>Density</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
</tr>
<tr>
<td>Density</td>
<td>7</td>
<td>7</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>
TITAN [Marijan et al. SPLC’13, SPLC’14]

Variability model to describe a software product line

Unoptimized test suite

Optimized (reduced/prioritized) test suite

Diagnostic views, feature coverage

Industrial case studies: ABB, Cisco
Automatic Test Case Generation
An example problem

```c
f( int i, ... )
{
  a.    j = 100;
      while( i > 1)
  b.        { j++; i--;}
...
  d. if( j > 500)
  e.   ...
```

Undecideable problem!
Path-oriented exploration

```c
f( int i, ... )
{
    j = 100;
    while( i > 1 )
    {
        j++ ; i-- ;
    }
    ...
    d. if( j > 500 )
    e. ...
```

1. Path selection
   e.g., (a-b)^14-...-d-e

2. Path condition generation (via symbolic exec.)
   \( j_1=100, i_1>1, j_2=j_1+1, i_2=i_1-1, i_2>1, ... , j_{15}>500 \)

3. Path condition solving
   unsatisfiable \( \rightarrow \) FAIL

Backtrack!
f( int i, ... )
{
  a.    j = 100;
       while( i > 1)
  b.        { j++ ; i-- ;}

  ...

d. if( j > 500)
e.     ...

1. Program (under Static Single Assignment form) as constraints
2. Control dependencies generation;
   \( j_1=100, \ i_3 \leq 1, \ j_3 > 500 \)
3. Global constraint reasoning
   \( j_1 \neq j_3 \) entailed \( \Rightarrow \) unroll the loop 400 times \( \Rightarrow i_1 \) in \( 401 \ldots 2^{31}-1 \)

No backtrack!
Program as constraints

- Type declaration:
  - signed long x; → x in \(-2^{31}..2^{31}-1\)

- Assignments:
  - i*=++i ; → i_2 = (i_1+1)^2

- Memory and array accesses and updates (Charreteur et al. JSS’09, Bardin et al. CPAIOR’12):
  - v=A[i] ( or p=Mem[&p] ) → variations of element/3

- Control structures: dedicated global constraints
  - Conditionnals (SSA) if D then C_1; else C_2 → ite/6
  - Loops (SSA) while D do C → w/5
Conditional as a global constraint: ite/6

$$\text{ite}( x > 0, j_1, j_2, j_3, j_1 = 5, j_2 = 18 ) \iff$$

- $$x > 0 \rightarrow j_1 = 5 \land j_3 = j_1$$
- $$\neg(x > 0) \rightarrow j_2 = 18 \land j_3 = j_2$$
- $$\neg(x > 0 \land j_1 = 5 \land j_3 = j_1) \rightarrow \neg(x > 0) \land j_2 = 18 \land j_3 = j_2$$
- $$\neg((\neg(x > 0) \land j_3 = j_2)) \rightarrow x > 0 \land j_1 = 5 \land j_3 = j_1$$
- $$\text{Join}( x > 0 \land j_1 = 5 \land j_3 = j_1, \neg(x > 0) \land j_1 = 18 \land j_3 = j_2)$$

Implemented as a global constraint: interface, awakening conditions, filtering algo.
While loop as a global constraint: w/5

\( v_3 = \phi(v_1, v_2) \)

while\( (\text{Dec}) \)

\[ w(\text{Dec}, V_1, V_2, V_3, \text{body}) \iff \]

\[ \text{Dec}_{V_3} \leftarrow V_1 \rightarrow \text{body}_{V_3} \leftarrow V_1 \land w(\text{Dec}, v_2, v_{\text{new}}, v_3, \text{body}_{V_2} \leftarrow v_{\text{new}}) \]

\[ \neg \text{Dec}_{V_3} \leftarrow V_1 \rightarrow v_3 = v_1 \]

\[ \neg (\text{Dec}_{V_3} \leftarrow V_1 \land \text{body}_{V_3} \leftarrow V_1) \rightarrow \neg \text{Dec}_{V_3} \leftarrow V_1 \land v_3 = v_1 \]

\[ \neg (\neg \text{Dec}_{V_3} \leftarrow V_1 \land v_3 = v_1) \rightarrow \text{Dec}_{V_3} \leftarrow V_1 \land \text{body}_{V_3} \leftarrow V_1 \land w(\text{Dec}, v_2, v_{\text{new}}, v_3, \text{body}_{V_2} \leftarrow v_{\text{new}}) \]

\[ \text{join}(\text{Dec}_{V_3} \leftarrow V_1 \land \text{body}_{V_3} \leftarrow V_1 \land w(\text{Dec}, v_2, v_{\text{new}}, v_3, \text{body}_{V_2} \leftarrow v_{\text{new}}), \neg \text{Dec}_{V_3} \leftarrow V_1 \land v_3 = v_1) \]
f( int i ) {
    j = 100;
    while( i > 1 )
    {
        j++;
        i--;
    }
    ...}
    if( j > 500)
    ...

w(Dec, V_1, V_2, V_3, body) :-
  Dec_{V_3\leftarrow V_1} \rightarrow body_{V_3\leftarrow V_1} \land w(Dec, V_2, v_{new}, V_3, body_{V_2\leftarrow v_{new}})
  \neg Dec_{V_3\leftarrow V_1} \rightarrow v_3 = v_1
  \neg (Dec_{V_3\leftarrow V_1} \land body_{V_3\leftarrow V_1}) \rightarrow \neg Dec_{V_3\leftarrow V_1} \land v_3 = v_1
  \neg (\neg Dec_{V_3\leftarrow V_1} \land v_3 = v_1) \rightarrow
  Dec_{V_3\leftarrow V_1} \land body_{V_3\leftarrow V_1} \land w(Dec, V_2, v_{new}, V_3, body_{V_2\leftarrow v_{new}})
  join(Dec_{V_3\leftarrow V_1} \land body_{V_3\leftarrow V_1} \land w(Dec, V_2, v_{new}, V_3, body_{V_2\leftarrow v_{new}}),
        \neg Dec_{V_3\leftarrow V_1} \land v_3 = v_1)

w(Dec, V_1, V_2, V_3, body) :-
  Dec_{V_3\leftarrow V_1} \rightarrow body_{V_3\leftarrow V_1} \land w(Dec, V_2, v_{new}, V_3, body_{V_2\leftarrow v_{new}})
  \neg Dec_{V_3\leftarrow V_1} \rightarrow v_3 = v_1
  \neg (Dec_{V_3\leftarrow V_1} \land body_{V_3\leftarrow V_1}) \rightarrow \neg Dec_{V_3\leftarrow V_1} \land v_3 = v_1
  \neg (\neg Dec_{V_3\leftarrow V_1} \land v_3 = v_1) \rightarrow
  Dec_{V_3\leftarrow V_1} \land body_{V_3\leftarrow V_1} \land w(Dec, V_2, v_{new}, V_3, body_{V_2\leftarrow v_{new}})
  join(Dec_{V_3\leftarrow V_1} \land body_{V_3\leftarrow V_1} \land w(Dec, V_2, v_{new}, V_3, body_{V_2\leftarrow v_{new}}),
        \neg Dec_{V_3\leftarrow V_1} \land v_3 = v_1)

i = 23, j_1 = 100 ?
  no
i in 401..2^{31}-1

w(i_3 > 1, (i,j_1), (i_2,j_2), (i_3,j_3), j_2 = j_3 + 1 \land i_2 = i_3 - 1)

i_3 = 1, j_3 = 122
i_3 = 10 ?
  j_1 = 100, j_3 > 500 ?
Features of the w/5 relation

✓ It can be imbricated with other relations (e.g., nested loops \( w(\text{cond}_1, v_1, v_2, v_3, w(\text{cond}_2, ...)) \)) – It handles unbounded loops

✓ Managed by the solver as any other constraint (its consistency is iteratively checked, awakening conditions, success/failure/suspension)

✓ By construction, w is unfolded only when necessary but \( w \) may NOT terminate! (only a semi-correct procedure)

✓ Join is implemented using Abstract Interpretation operators (interval union, Q-polyhedra weak-join operator, simple widening operators)

(Gotlieb et al. CL’2000, Denmat et al. CP’2006)
EUCLIDE: Automatic Test Case Generation for C Programs

void P_rad_eta()
{
    MEM_PEMORDR = PEMORDR;
    PEMORDR = 0x0;
    FM_PEMORDR = 0x0;
}
else
{
    if (TPCODRDR != 194)
    {
        if (TPCODRDR <= 0)
        {
            trait2_eta();
        }
        else
        {
            if (TPCODRDR <= (194 - 13))
            {
                if (DIALRDR)
                {
                    trait3_eta();
                }
                else
                {
                    local_merdr3g = TP_RDR_7R.merdr3g
                    if (((local_merdr3g & 0x0001) == 0x00001)
                    {
                        trait1_eta();
                    }
                }
            }
        }
    }
}

[Gotlieb ICST’09, KER’12]
Conclusions

- Global constraints (existing ones or user-defined) can efficiently and effectively tackle difficult software testing problems – experimental results and industrial case studies

- So far, only a few subset of existing global constraints have been explored for that purpose (e.g., nvalue, gcc, element, all_different,...)

- Some software testing problems require the creation of dedicated global constraints to facilitate disjunctive reasoning, case-based reasoning or probabilistic reasoning

→ there is room for research in that area!
Perspectives

- More industrial case studies for demonstrating the potential of global constraints for software testing applications

- Using GCC WITH COSTS to deal with bi-objective optimisation in test suite reduction (e.g., to also select test cases based on execution time in addition to requirement coverage)

- Test Case Execution Scheduling with CUMULATIVE
Thanks to my co-authors:

Bernard Botella (CEA)
Mats Carlsson (SICS)
Tristan Denmat (Inria)
Marius Liaeen (CISCO)
Dusica Marijan (SIMULA)
Alexandre Pétillon (SIMULA)