Computational Science Demands a New Paradigm

The field has reached a threshold at which better organization becomes crucial. New methods of verifying and validating complex codes are mandatory if computational science is to fulfill its promise for science and society.

Douglass E. Post and Lawrence G. Votta

Computers have become indispensable to scientific research. They are essential for collecting and analyzing experimental data, and they have largely replaced pencil and paper as the theorist’s main tool. Computers let theorists extend their studies of physical, chemical, and biological systems by solving difficult nonlinear problems in a few minutes that once took years. Processor speed continues to increase, and massive parallelization is augmenting that speed, albeit at the cost of increasingly complex computer architectures. Massively parallel computers with thousands of processors are becoming widely available at relatively low cost, and larger ones are being developed.

Part of the problem is simply that it’s hard to decide whether a code result is right or wrong. Our experience as referees and editors tells us that the peer review process in computational science generally doesn’t provide as effective a filter as it does for experiment or theory. Many things that a referee cannot detect could be wrong with a computational science paper. The code could have hidden defects, it might be applying algorithms improperly, or its spatial or temporal resolution might be inappropriately coarse.

“...diligence and alertness are far from a guarantee that the code is free of defects. Better verification techniques are desperately needed.”
...the whole point of science is to be able to prove that your answers are valid...

Survey of ~ 2000 Scientists

Top 3 topics about which respondents felt they did not know as much as they should:

1. software construction
2. verification
3. testing
Many scientific results are corrupted, perhaps fatally so, by undiscovered mistakes in the software used to calculate and present those results.
Hatton & Roberts: average distance from mean
Goals of TASS

1. verification & debugging of programs used in computational science
2. High Performance Computing
   - parallel programs: Message Passing Interface (MPI)
3. automatic (mostly)
   - produce useful results with no effort
   - more effort (code annotations) $\rightarrow$ stronger results
4. functional equivalence for real arithmetic
5. verify generic safety properties
6. support real code, including standard libraries
7. good engineering:
   - usability, documentation, open-source, automated testing, clear module boundaries, well-documented interfaces, easily extended/modified
Goals of TASS

1. verification & debugging of programs used in computational science
2. High Performance Computing
   • parallel programs: Message Passing Interface (MPI)
3. automatic (mostly)
   • produce useful results with no effort
   • more effort (code annotations) → stronger results
4. functional equivalence for real arithmetic
5. verify generic safety properties
6. support real code, including standard libraries
7. good engineering:
   • usability, documentation, open-source, automated testing, clear module boundaries, well-documented interfaces, easily extended/modified

Version 1.0 available now: http://vsl.cis.udel.edu/tass
Some Related Work


4.Boldo, Filliâtre, Formal Verification of Floating-Point Programs, ARITH-18 2007 (Caduceus)

5. Vakkalanka, Sharma, Gopalakrishnan, ISP: A Tool for Model Checking MPI Programs, PPoPP 2008
TASS: Properties Verified

1. functional equivalence
2. absence of user-specified assertion violations
3. freedom from deadlock
4. absence of buffer overflows (MPI, pointer arithmetic, array indexing, …)
5. no reading uninitialized variables
6. no division by zero
7. proper use of malloc/free
8. absence of memory leaks
9. proper use of MPI_Init, MPI_Finalize, …
10. (ordinary) loop invariants
11. loop joint invariants
TASS: Input Language

- currently: a subset of C99 + MPI + pragmas
- including
  1. functions
  2. types: real, integer, boolean, arrays, structs, pointers, functions
  3. dynamic allocation (malloc/free)
  4. &, *, pointer arithmetic
  5. assert

#pragma TASS assert forall {int j | 0 <= j && j < n} a[j] == 1;

- excluding (for now)
  1. bit-wise operations
  2. nested scopes
  3. support for many standard libraries (math.h,...)
TASS: Restrictions

- small configurations
  - small number of processes, bounds on inputs, etc.
  - but: exhaustive exploration of all possible behaviors within the bounds
- limits on input language
- does not deal with floating-point issues (currently)
- limits due to automated theorem proving
  - theorem prover(s) might not be able to prove valid assertions
  - but: TASS is conservative: reports anything that could possibly be wrong
- categorizes errors: proveable, maybe, etc.
Basic Techniques used by TASS

- symbolic execution
- state space exploration ("model checking")
  - MPI-specific "partial order reduction" techniques to reduce the number of states explored
- comparative symbolic execution
  - Siegel, Mironova, Avrunin, Clarke, Using model checking with symbolic execution to verify parallel numerical programs, ISSTA 2006
“Bias in occurrence of message orderings: BG/L”

R. Vuduc, M. Schulz, D. Quinlan, B. de Supinski

Improving distributed memory applications testing by message perturbation

PADTAD’06 (slide from presentation)
Symbolic execution

- J.C. King, *Symbolic execution and program testing*, CACM 1976
- addresses the problem of sampling the inputs
  - many test cases can be grouped together into a single test
- useful for sequential as well as parallel programs
- useful for reasoning about numerical properties
- can be automated
Q: How many Coq programmers does it take to change a lightbulb?

A: Are you kidding? It takes 2 post-docs six months just to prove that the bulb and the socket are both threaded in the same direction.
Symbolic execution

**Input:** symbolic constants \( x_0, x_1, \ldots \)

**Output:** symbolic expressions in the \( x_i \)

\[
0.0 + (x_0 \times x_4) + x_1 \times x_6 = (0.0 + (x_0 \times x_4)) + x_1 \times x_6
\]
The path condition

• how do you execute a conditional statement?!  
  
  • if (\(x_0 \neq 0\)) \{\ldots\} \textbf{else} \{\ldots\}
The path condition

• how do you execute a conditional statement?!
  • if \((x_0 \neq 0)\) \{...\} else \{...\}
  • add a boolean-value symbolic variable \(p\)
    • initially, \(p \leftarrow \text{true} \)
The path condition

- how do you execute a conditional statement?!
  - \textbf{if} \ (x_0 \neq 0) \ \{\ldots\} \ \textbf{else} \ \{\ldots\}
- add a boolean-value symbolic variable \( p \)
  - initially, \( p \leftarrow true \)
- make a \textbf{nondeterministic choice} between \textit{true} and \textit{false} branch
  - if you choose the \textit{true} branch, update path condition:
    - \( p \leftarrow p \land x_0 \neq 0 \)
  - if you choose the \textit{false} branch, update path condition:
    - \( p \leftarrow p \land x_0 = 0 \)
The path condition

- how do you execute a conditional statement?!
  - if \((x_0 \neq 0)\) \{\ldots\} else \{\ldots\}

- add a boolean-value symbolic variable \(p\)
  - initially, \(p \leftarrow true\)

- make a **nondeterministic choice** between *true* and *false* branch
  - if you choose the *true* branch, update path condition:
    - \(p \leftarrow p \land x_0 \neq 0\)
  - if you choose the *false* branch, update path condition:
    - \(p \leftarrow p \land x_0 = 0\)

- \(p\) encodes the condition on the input that had to be true in order for control to have followed the current path
The path condition

• how do you execute a conditional statement?!
  • if \((x_0 \neq 0)\) \{\ldots\} else \{\ldots\}

• add a boolean-value symbolic variable \(p\)
  • initially, \(p \leftarrow true\)

• make a nondeterministic choice between true and false branch
  • if you choose the true branch, update path condition:
    • \(p \leftarrow p \land x_0 \neq 0\)
  • if you choose the false branch, update path condition:
    • \(p \leftarrow p \land x_0 = 0\)

• \(p\) encodes the condition on the input that had to be true in order for control to have followed the current path

• now use a model checker to explore all possible nondeterministic choices
The path condition

- how do you execute a conditional statement?!
  - **if** \((x_0 \neq 0)\) \{\ldots\} **else** \{\ldots\}
- add a boolean-value symbolic variable \(p\)
  - initially, \(p \leftarrow true\)
- make a **nondeterministic choice** between *true* and *false* branch
  - if you choose the *true* branch, update path condition:
    - \(p \leftarrow p \land x_0 \neq 0\)
  - if you choose the *false* branch, update path condition:
    - \(p \leftarrow p \land x_0 = 0\)
- \(p\) encodes the condition on the input that had to be true in order for control to have followed the current path
- now use a **model checker** to explore all possible nondeterministic choices
- every time \(p\) is updated, invoke an **automated theorem prover** to check that \(p\) is **satisfiable**
  - if not, you are on an **infeasible path**: backtrack immediately
Result of symbolic execution for Gaussian elimination

Program transforms a matrix to its reduced row-echelon form:

\[
x = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix} \quad \rightarrow \quad y = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}
\]
Result of symbolic execution for Gaussian elimination

Program transforms a matrix to its reduced row-echelon form:

\[
\mathbf{x} = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix} \quad \rightarrow \quad \mathbf{y} = \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \end{pmatrix}
\]

\[
\mathbf{y} = \begin{cases} 
\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \land x_2 = 0 \land x_1 = 0 \land x_3 = 0 \\
\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \land x_2 = 0 \land x_1 = 0 \land x_3 \neq 0 \\
\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \land x_2 = 0 \land x_1 \neq 0 \\
\begin{pmatrix} 1 & x_3 \div x_2 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \land x_2 \neq 0 \land x_1 = 0 \\
\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } x_0 = 0 \land x_2 \neq 0 \land x_1 \neq 0 \\
\begin{pmatrix} 1 & x_1 \div x_0 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 \neq 0 \land x_3 - x_2(x_1 \div x_0) = 0 \\
\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } x_0 \neq 0 \land x_3 - x_2(x_1 \div x_0) \neq 0
\end{cases}
\]
Structure of the State of a TASS Model

- State
  - Shared Variables
    - Input Variables
    - Output Variables
    - Path Condition
  - Procs
    - 0 1 ... n-1
  - Messages
    - 0 1 ... m-1
- Stack
- Global Variables
  - Frame 0
  - Frame 1
  - ... Frame k-1
  - Local Variables
  - Location
Function Body: Guarded Transition System
### Statement Types

<table>
<thead>
<tr>
<th>statement type</th>
<th>example guard</th>
<th>example transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td>true</td>
<td>$x[i] \leftarrow (y \ast z)/7.2$</td>
</tr>
<tr>
<td>NOOP</td>
<td>$x \neq y + z$</td>
<td>identity</td>
</tr>
<tr>
<td>SEND</td>
<td>$nfull(source, dest)$</td>
<td>$send(source, dest, tag, data)$</td>
</tr>
<tr>
<td>RECV</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>ASSERT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASSUME</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVOKE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Execution Semantics of a TASS Model

• defined as a state transition system
• the set of states is defined as above
• given a state $s$, the set of transitions enabled from $s$ is determined as follows:
  • let $pc$ be the path condition in $s$
  • for each process $p$:
  • look at current location $l$ of $p$ in $s$
  • for each statement $(guard, transformation)$ departing from $l$:
  • let $q$ be the result of evaluating $guard$ at $s$
  • if $p \land q$ is satisfiable then there is a transition from $s$ to a new state $s'$
  • the path condition in $s'$ is $p \land q$ and the rest of the state is determined by applying $transformation$ to $s$. 
Symbolic Representations: Canonical Forms

- two symbolic expressions are equivalent if given any assignment of concrete values to symbolic constants, both expressions evaluate to the same concrete value
- if a state $s'$ is obtained from $s$ by replacing symbolic expressions with equivalence symbolic expressions
  - $s$ and $s'$ represent the same set of concrete states
  - say $s$ and $s'$ are equivalent
- so the components of the state may be considered as equivalence classes of symbolic expressions
- the ability to recognize that two expressions are equivalent can therefore reduce the number of states searched
- this is facilitated by placing every expression into a canonical form
  - boolean-valued: conjunctive normal form
  - integer-valued: polynomial form
  - real-valued: rational form
Canonical Form: Integer Expressions

- a symbolic expression $x$ of integer type is an integer primitive if $x$ has one of the following forms:
  - a symbolic constant $X$,
  - an array read expression $e_1[e_2]$,
  - a record member read expression $e_1.e_2$
  - an evaluated uninterpreted function expression $f(e_1, \ldots, e_n)$,
  - $\ldots$ (any operation other than $\ast$, $+$, $-$)

- any expression formed from numeric primitives and concrete integers using $\ast$, $+$, $-$ can be written as a polynomial:
  $$\sum_{i_1, \ldots, i_n} \lambda_{i_1, \ldots, i_n} x_1^{i_1} \cdots x_n^{i_n}$$

  where the $\lambda_{i_1, \ldots, i_n}$ are concrete integers.

- a total order can be placed on the primitives
  - $\ldots$yielding a total order on monic monomials

- arrange terms in order of increasing monics for the “canonical form”
 Canonical Form: Real Expressions

• a real primitive is defined similarly

• any expression formed from real primitives and concrete rational numbers using *, +, −, and / can be written as a rational function

\[ \frac{f(x)}{g(x)} \]

where \( f(x) \) and \( g(x) \) are polynomials in the primitives and \( g \) is monic.

• a factorization is associated to each polynomial

• common factors are canceled when dividing
### Evaluation

<table>
<thead>
<tr>
<th>program</th>
<th>bounds</th>
<th>nprocs</th>
<th>time (s)</th>
<th>states</th>
<th>values</th>
<th>m</th>
</tr>
</thead>
<tbody>
<tr>
<td>adder</td>
<td>$n \leq 100$</td>
<td>10</td>
<td>11.1</td>
<td>23936</td>
<td>17580</td>
<td></td>
</tr>
<tr>
<td>adder</td>
<td>$n \leq 100$</td>
<td>30</td>
<td>135.6</td>
<td>40096</td>
<td>18381</td>
<td></td>
</tr>
<tr>
<td>laplace</td>
<td>$n_x \leq 5 \land n_y \leq 7 \land B \leq 3$</td>
<td>12</td>
<td>131.2</td>
<td>73499</td>
<td>22136</td>
<td></td>
</tr>
<tr>
<td>laplace</td>
<td>$n_x \leq 6 \land n_y \leq 8 \land B \leq 3$</td>
<td>3</td>
<td>1649.1</td>
<td>61935</td>
<td>26955</td>
<td></td>
</tr>
<tr>
<td>diffusion</td>
<td>$n_x \leq 10 \land n_t \leq 4$</td>
<td>7</td>
<td>543.3</td>
<td>3746952</td>
<td>14717</td>
<td></td>
</tr>
<tr>
<td>diffusion</td>
<td>$n_x \leq 16 \land n_t \leq 4$</td>
<td>8</td>
<td>5523.9</td>
<td>27151911</td>
<td>33556</td>
<td></td>
</tr>
<tr>
<td>diffusion</td>
<td>$n_x \leq 20 \land n_t \leq 6$</td>
<td>6</td>
<td>755.3</td>
<td>2735221</td>
<td>78478</td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td>$l \leq 3 \land m \leq 6 \land n \leq 3$</td>
<td>3</td>
<td>4.2</td>
<td>39785</td>
<td>21769</td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td>$l \leq 4 \land m \leq 8 \land n \leq 4$</td>
<td>4</td>
<td>91.0</td>
<td>977112</td>
<td>390024</td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td>$l \leq 5 \land m \leq 5 \land n \leq 5$</td>
<td>5</td>
<td>1761.6</td>
<td>17317811</td>
<td>5050494</td>
<td></td>
</tr>
</tbody>
</table>