The Toolkit for Accurate Scientific Software

Stephen F. Siegel, Timothy Zirkel, Yi Wei

Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, DE, USA

Third International Workshop on Numerical Software Verification
Edinburgh, Scotland
15 Jul 2010

Post & Votta, *Physics Today*, 2005

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

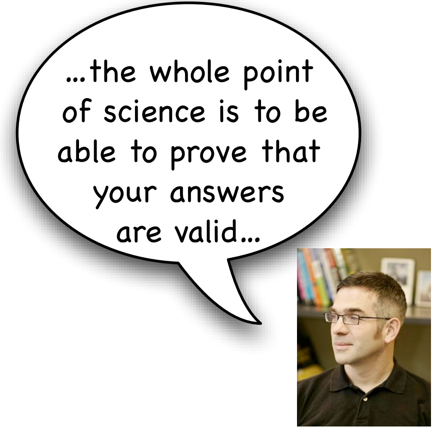
Computers have become indispensable to scientific research. They are essential for collecting and analyzing experimental data, and they have largely replaced pencil and paper as the theorist's main tool. Computers let theo-

efficiently exploit the capacities of the increasingly complex computers. The prediction challenge is to use all that computing power to provide answers reliable enough to form the basis for important decisions.

The performance challenge is being met, at least for the next 10 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 rela-

"...diligence and alertness are far from a guarantee that the code is free of defects. Better verification techniques are desperately needed."

Greg Wilson, *American Scientist*, 2009



Survey of ~ 2000 **Scientists**

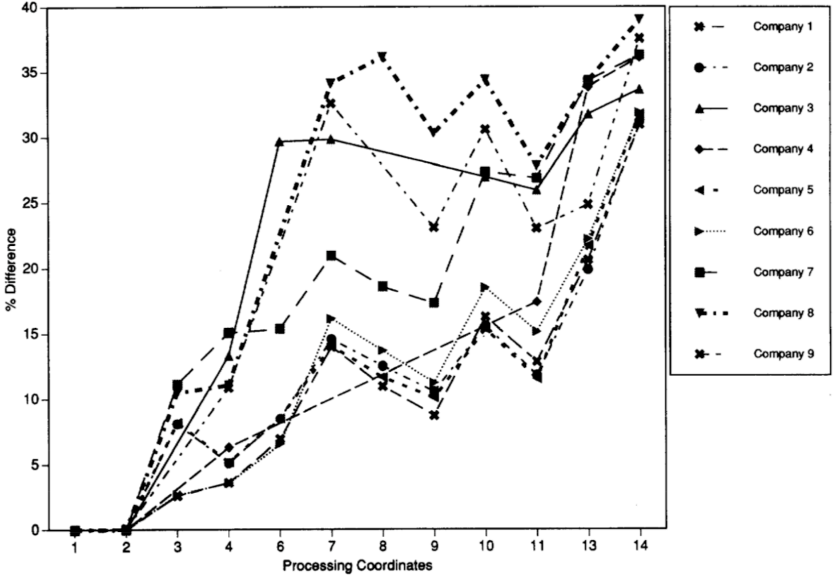
Top 3 topics about which respondents felt they did not know as much as they should:

1. software construction
2. verification
3. testing

Les Hatton, IEEE *Computer*, 2007

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 Performace 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 propeties
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 Performace 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 propeties
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

1. Cadar, Dunbar, Engler, KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs SOSDI 2008
2. Barrett, Fang, Goldberg, Hu, Pnueli, Zuck, TVOC: A Translation Validator for Optimizing Compilers, CAV 2005
3. Beyer, Henzinger, Jhala, Majumdar, The Software Model Checker BLAST: Applications to Software Engineering, IJSTTT 2007
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`

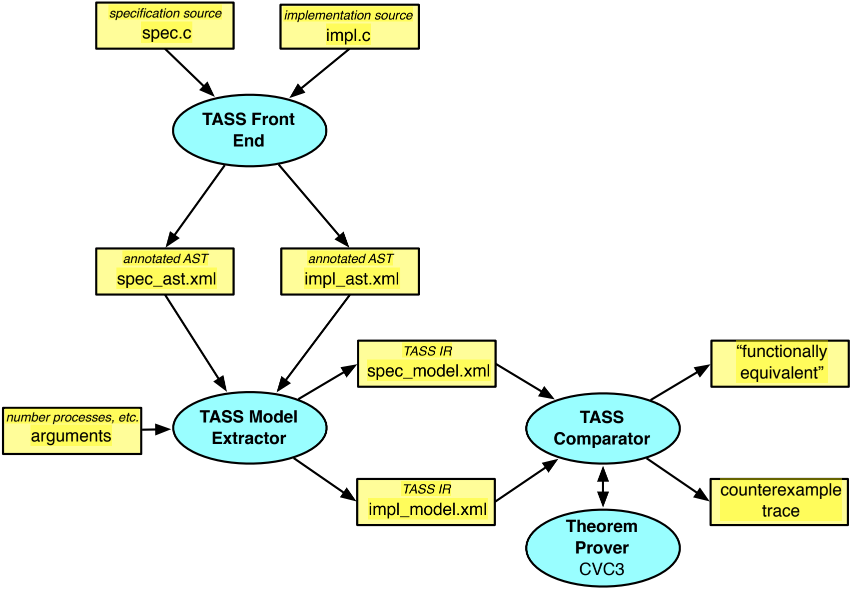
```
#pragma TASS assert forall {int j | 0 <= j && j < n} a[j] == 1;
```

- 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.

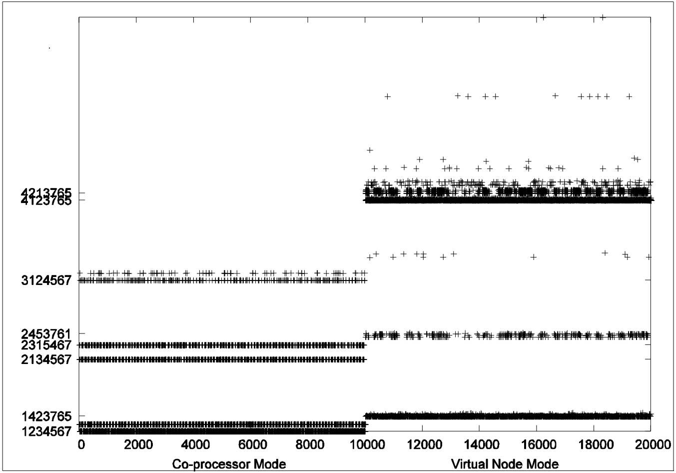
TASS Tool Chain



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

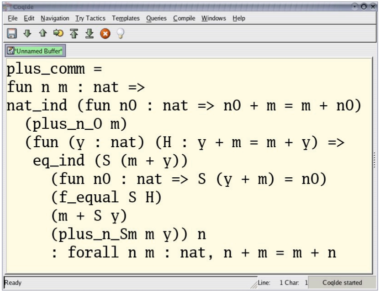
PADTAD'06 (slide from presentation)

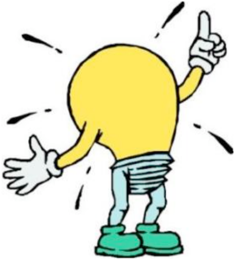
Symbolic execution

- 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

Theorem Proving Considered Difficult (James Iry)

Q: How many Coq programmers does it take to change a lightbulb?



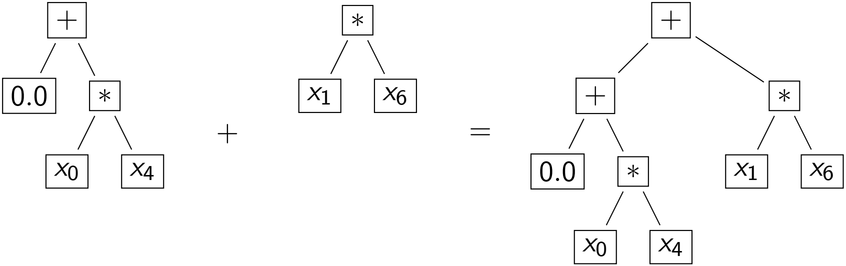


A: Are you kidding? It takes 2 post-docs six months just to prove that the bulb and the socket are both threaded in the same direction.

Symbolic execution

Input: symbolic constants $x_0, x_1, \ldots$

Output: symbolic expressions in the x_i



$$0.0 + (x_0 x_4) \quad + \quad x_1 x_6 \quad = \quad (0.0 + (x_0 x_4)) + x_1 x_6$$

The path condition

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

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, $p \leftarrow \mathit{true}$

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, $p \leftarrow \mathit{true}$
- make a nondeterministic choice between *true* and *false* branch
  - if you choose the *true* branch, update path condition:
    - $p \leftarrow p \wedge x_0 \neq 0$
  - if you choose the *false* branch, update path condition:
    - $p \leftarrow p \wedge x_0 = 0$

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:
    - $p \leftarrow p \wedge x_0 \neq 0$
  - if you choose the *false* branch, update path condition:
    - $p \leftarrow p \wedge x_0 = 0$
- 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:
    - $p \leftarrow p \wedge x_0 \neq 0$
  - if you choose the *false* branch, update path condition:
    - $p \leftarrow p \wedge x_0 = 0$
- 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:
    - $p \leftarrow p \wedge x_0 \neq 0$
  - if you choose the *false* branch, update path condition:
    - $p \leftarrow p \wedge x_0 = 0$
- 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:

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

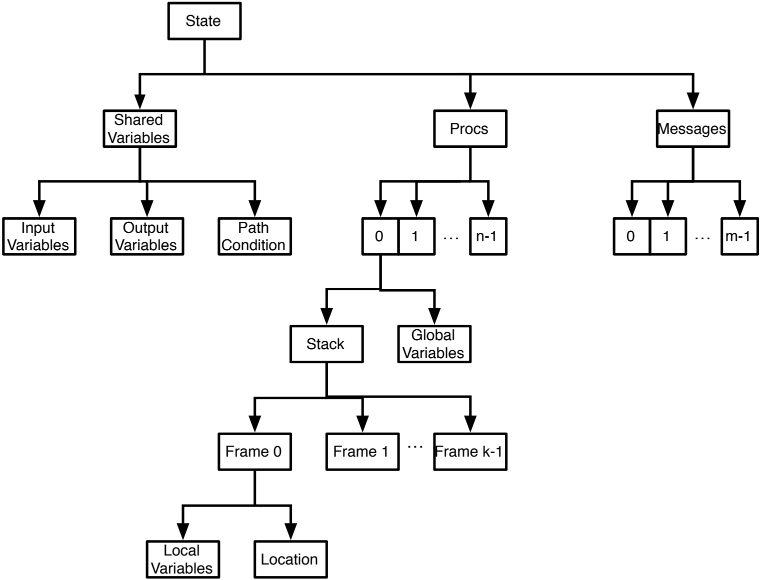
Result of symbolic execution for Gaussian elimination

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

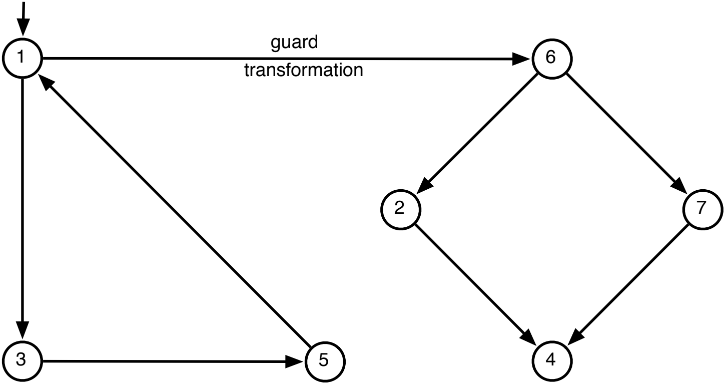
$$\mathbf{x} = \begin{pmatrix} x_0 & x_1 \\ x_2 & x_3 \end{pmatrix} \quad \rightarrow \quad \mathbf{y} = \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \end{pmatrix}$$

$$\mathbf{y} = \begin{cases} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 = 0 \wedge x_1 = 0 \wedge x_3 = 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 = 0 \wedge x_1 = 0 \wedge x_3 \neq 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 = 0 \wedge x_1 \neq 0 \\ \begin{pmatrix} 1 & x_3/x_2 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 \neq 0 \wedge x_1 = 0 \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } x_0 = 0 \wedge x_2 \neq 0 \wedge x_1 \neq 0 \\ \begin{pmatrix} 1 & x_1/x_0 \\ 0 & 0 \end{pmatrix} & \text{if } x_0 \neq 0 \wedge x_3 - x_2(x_1/x_0) = 0 \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & \text{if } x_0 \neq 0 \wedge x_3 - x_2(x_1/x_0) \neq 0 \end{cases}$$

Structure of the State of a TASS Model



Function Body: Guarded Transition System



Statement Types

statement type	example guard	example transformation
ASSIGN	*true*	$x[i] \leftarrow (y * z)/7.2$
NOOP	$x \neq y + z$	*identity*
SEND	$nfull(source, dest)$	$send(source, dest, tag, data)$
RECV	$\ldots$	$\ldots$
ASSERT		
ASSUME		
INVOKE		
RETURN		

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 $p \wedge q$ is satisfiable then there is a transition from s to a new state s'
  - the path condition in s' is $p \wedge q$ 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 $f(e_1, \ldots, e_n)$,
  - ... (any operation other than $*$, $+$, $-$)
- any expression formed from numeric primitives and concrete integers using $*$, $+$, $-$ 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
  - ...yiedling 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

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