The Toolkit for Accurate Scientific Software

Stephen F. Siegel, Yi Wei, and Timothy K. Zirkel*

Verified Software Laboratory, Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716, USA
{siegel|ywei|zirkeltk}@udel.edu http://vsl.cis.udel.edu

1 Overview

The *Toolkit for Accurate Scientific Software* (TASS) is a new suite of integrated tools for the formal verification of programs used in computational science, including message-passing-based parallel programs. TASS uses symbolic execution [1] and model checking techniques to verify a number of standard safety properties (such as absence of deadlocks and out-of-bound references), but its most innovative feature is the ability to establish that two programs are functionally equivalent. This is particularly useful in scientific computing, where developers often start with a simple sequential encoding of an algorithm, then gradually transform this into a production code by introducing parallelism and a host of other optimizations by hand. The final implementation is intended to be functionally equivalent to the original specification, but this is generally extremely difficult to check.

For the most part, TASS is used to verify small, bounded configurations of programs. However, it performs an exhaustive exploration of all possible inputs and behaviors within the specified bounds. Moreover, with some additional help from the user, TASS can use novel symbolic techniques to verify the equivalence of programs with unbounded loops.

TASS is open source software distributed under the GNU Public License. The Java source code, examples, JUnit test suite, and reference materials are all available at `http://vsl.cis.udel.edu/tass`.

Using TASS. Fig. 1 shows the steps involved in using TASS to check the functional equivalence of two programs. The TASS *model extractor* translates the program source codes into the TASS intermediate representation. The number of processes of each program is fixed at this step. The IR is designed to be an easy translation target for programming languages commonly used by computational scientists. It supports parallelism; functions; boolean, integer, and real types (support for finite-precision integer and floating-point types is in progress); (multi-dimensional) array and record types; dynamically allocated data; and pointers and pointer arithmetic. Currently, the extractor accepts programs in the language MiniMP. A dialect of C with simple message-passing commands, MiniMP is closely related to the TASS IR. Front-ends for full C, C++, and Fortran, with support for the Message-Passing Interface (MPI), are in progress.

* Supported by the U.S. National Science Foundation grant CCF-0733035

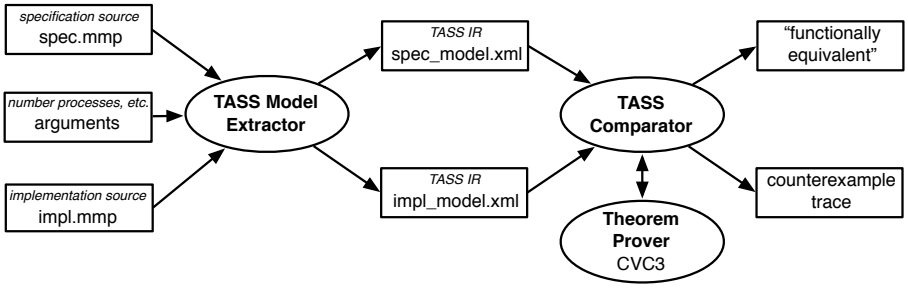

Fig. 1. Verification of functional equivalence using TASS

A simple example for summing the elements of an array is shown in Figure 2. The MiniMP `input` and `output` type qualifiers specify which variables are to be used for program input and output; two models with equivalent input/output signatures can be compared for functional equivalence. *Assumptions* restricting the input values can be added in curly braces. In this case the assumption $\mathtt{n} \geq 0$ has been added so TASS will not complain about the array declaration `double[n] a`. An upper bound on `n` is required in order for verification to terminate. As in typical MPI programs, the parallel version is expressed as code for a single generic process which is instantiated multiple times at model extraction. The built-in macros `PID` and `NPROCS` give the unique process ID number and the number of processes, respectively.

```
input int n {n>=0 && n<=100}; input double a[n]; output double sum;
void main() {
  double result = 0.0; int i;
  for (i=0; i<n; i++) result += a[i];
  sum = result;
}
```

(a) `adder_seq.mmp`, sequential version serving as specification

```
input int n {n>=0 && n<=100}; input double a[n]; output double sum;
double localSum = 0.0;
double computeGlobalSum() {
  double result = localSum; double buffer; int i;
  for (i=1; i<NPROCS; i++) { recv(buffer, i, 0); result += buffer; }
  return result;
}
void main() {
  int first = n*PID/NPROCS; int afterLast = n*(PID+1)/NPROCS; int i;
  for (i=first; i<afterLast; i++) localSum += a[i];
  if (PID == 0) sum = computeGlobalSum(); else send(localSum, 0, 0);
}
```

(b) `adder_par.mmp`, parallel version. Each process adds one section of the array; the partial results are sent to process 0, where they are summed.

Fig. 2. Two functionally equivalent MiniMP programs to add the elements of an array

To verify the functional equivalence of the sequential code with the 10-process instantiation of the parallel code, the user could issue the command

```
tass compare -reduce=urgent adder_seq.mmp adder_par.mmp -np 10
```

(Option `reduce` specifies the partial order reduction strategy, in this case the "urgent" reduction strategy of [2].) For this example, TASS reports that the two models are functionally equivalent (after exploring 23,936 states in 21 seconds on a 2.2 GHz MacBook Pro). This means that for any input satisfying $0 \leq \mathtt{n} \leq 100$, assuming "infinite precision" arithmetic, both the sequential and the 10-process parallel program will terminate, and at termination their outputs will agree.

In other cases, TASS reports that the two models *may not* be equivalent, and produces program traces showing the symbolic values of all variables at each step, and finally the values of the two output variables that disagree. The *may* is significant because the analysis performed by TASS is conservative, but not necessarily precise, i.e., spurious counterexamples are possible.

2 Verification Techniques used by TASS

The main verification technique used by the TASS back-end combines symbolic execution with a depth-first search of the state space (cf. [3]).

Values are represented as symbolic expressions: input variables are assigned symbolic constants in the initial state, and every operation returns a symbolic expression. Branches are modeled using non-deterministic choice; all choices are explored in the course of the DFS. An additional boolean-valued *path condition* variable *pc* keeps track of assumptions and branch choices. For example, if input variable `n` of `adder_seq` is assigned symbolic constant X, in the initial state *pc* is $X \geq 0 \wedge X \leq 100$. Upon reaching the `for` loop the first time, a choice is made on the expression `i<n`, which evaluates to $0 < X$. If the *true* branch is chosen, the value of *pc* becomes $X \geq 0 \wedge X \leq 100 \wedge 0 < X$ (which TASS immediately simplifies to $X \geq 1 \wedge X \leq 100$). If the *false* branch is chosen *pc* becomes $X \geq 0 \wedge X \leq 100 \wedge 0 \leq X$ (which simplifies to $X = 0$). An automated theorem prover (currently, CVC3 [4]) is used to determine whether the execution of a transition would lead to an unsatisfiable *pc*, in which case that transition is disabled. The theorem prover is also used to check a number of assertions, including user-specified assertions in the source code, array indices are within bounds, and denominators in division operations are non-zero.

Termination is verified by checking there are no reachable cycles in the state space.

Equivalence is verified using *comparative symbolic execution* [5]. The specification's state space is explored as described above, except that each time a terminal state s is reached, a property $\pi(\alpha, \mathbf{y})$ is checked, where α is the value of *pc* in s and $\mathbf{y}$ is the tuple of output values in s. Property π asserts that all executions of the implementation that begin with the same input symbolic constants and with initial path condition α will terminate with output tuple values equivalent to $\mathbf{y}$. This is verified by a separate DFS search in which α is used

for the initial value of the implementation's path condition variable. The equivalence of the output variables is determined by the theorem prover, using as the assumption the terminal value of the implementation's path condition variable.

Handling Unbounded Loops. TASS can verify the equivalence of two programs with unbounded loops using a technique based on *loop joint invariants*. A joint invariant is like an ordinary loop invariant but relates the state of one program to the state of another on corresponding loop iterations. This approach requires the user to provide a joint invariant as a code annotation. Fig. 3 shows an excerpt from two programs that compute the mean of the elements of an array.

```
l:                      l@main { s*(i-1)==spec.s && i-1==spec.i } :
while (i<N) {           while (i<=N) {
  s=s+a[i];               s=((i-1)*s+a[i-1])/i;
  i=i+1;                  i=i+1;
 }                       }
```

Fig. 3. Loop Joint Invariant. Left: loop in specification. Right: loop in implementation. The joint invariant specifies a relation between the variables in the implementation and those in the specification on corresponding iterations.

When TASS encounters a joint invariant during the search, it essentially constructs an inductive proof of the equivalence of the two loops. This proof relies on an abstraction that is similar to a predicate abstraction: it records just enough information to know how the variables in the two programs correspond, without recording the exact values of those variables. Upon exiting the loop, the variables that have been modified in the loop bodies hold fresh symbolic constants that are unrelated to the original input symbolic constants, but are related to each other in accordance with the joint invariant. This is all that is needed in order to establish equivalence.

If the user provides an incorrect invariant or one that is not strong enough to establish equivalence, the worst that can happen is that TASS will report a possible violation and produce a spurious counterexample. The approach is still conservative, so a positive result always implies equivalence holds.

The loop technique works for more complex loop transformations as well, including nested loops, loop tiling, and loop skewing. In some cases, the technique has allowed us to find a defect that we could not find with an ordinary (bounded) technique without exhausting memory resources. These examples and a detailed explanation of the technique can be found on [6].

3 Experimental Results

We have applied TASS to successively more complex numerical programs. While these programs are certainly simpler that those used in the current scientific

practice, they do capture many of the patterns common in the real codes. They include a time-stepping diffusion equation simulation, a 2d-Laplace equation solver using Jacobi iteration, and matrix multiplication schemes using loop tiling, manager-worker parallelism, and other optimizations.

Some experimental data is shown in Fig. 4 for various bounds on four different parallel programs. In each case, TASS was used to establish functional equivalence with a simple sequential specification. Note that the constraints on the inputs sizes are bounds, so for example the case $n \leq 100$ means that equivalence was verified for all $n \in \{0, 1, \ldots, 100\}$. The bounds in the Laplace example are the number of discrete points in the x and y directions and the number of time steps. In the 1d-diffusion case, the number of discrete points and number of time steps; in matrix multiplication, the number of rows and columns in the first input matrix and number of columns in the second matrix.

program	bounds	nprocs	time (s)	states	values	messages	proofs
adder	$n \leq 100$	10	11.1	23936	17580	873	500
adder	$n \leq 100$	30	135.6	40096	18381	2523	500
laplace	$n_x \leq 5 \wedge n_y \leq 7 \wedge B \leq 3$	12	131.2	73499	22136	1419	269
laplace	$n_x \leq 6 \wedge n_y \leq 8 \wedge B \leq 3$	3	1649.1	61935	26955	856	509
diffusion	$n_x \leq 10 \wedge n_t \leq 4$	7	543.3	3746952	14717	1590	98
diffusion	$n_x \leq 16 \wedge n_t \leq 4$	8	5523.9	27151911	33556	3255	152
diffusion	$n_x \leq 20 \wedge n_t \leq 6$	6	755.3	2735221	78478	6230	236
matrix	$l \leq 3 \wedge m \leq 6 \wedge n \leq 3$	3	4.2	39785	21769	600	121
matrix	$l \leq 4 \wedge m \leq 8 \wedge n \leq 4$	4	91.0	977112	390024	2208	219
matrix	$l \leq 5 \wedge m \leq 5 \wedge n \leq 5$	5	1761.6	17317811	5050494	3448	215

Fig. 4. Experimental Results for Verifying Functional Equivalence: nprocs is number of processes, values is the total number of symbolic expressions generated during the search, messages in the total number of messages sent, proofs is the number of calls to the theorem prover CVC3.

4 Related and Future Work

Among the many tools for verifying properties of numerical programs, we mention only a few that are most relevant to TASS. KLEE [7] is a symbolic execution tool for generating tests to improve test coverage; it can also check functional equivalence in some cases. It differs from TASS in several ways. For one, KLEE uses "bit-precise" reasoning instead of mathematical real arithmetic; this is not as useful for equivalence checking of numerical programs since they are rarely expected to be bit-equivalent; the adder programs considered above form a typical example. TVOC [8] is a tool for checking the correctness of compiler optimizations by functional equivalence verification based on set of pre-defined transformation patterns. BLAST [9] is a C program verification system to check safety properties based on predicate abstraction. Caduceus [10] is a set of tools

for checking Java and C programs; it uses special comments or JML style annotations to specify the accuracy requirements. None of these tools applies to message-passing based parallel programs. ISP [11], on the other hand, is geared specifically for MPI programs. It uses a modified runtime system to explore all relevant interleavings, but like ordinary testing only operates on concrete inputs, so cannot establish functional equivalence.

Our future work on TASS includes several major improvements. First, we will use a mature compiler infrastructure, such as ROSE or LLVM, to construct a front-end that can parse C, C++ and Fortran source code and generate an IR model. Second, we are working on the optimization of the symbolic package to dramatically reduce the number of calls made to the theorem prover, thus lowering the overall run time. Last but not the least, we will develop methods to extract and infer the loop joint invariants automatically, which will reduce the effort on the user side and enable TASS to verify unbounded models fully automatically.

References

1. King, J.C.: Symbolic execution and program testing. Comm. ACM **19**(7) (1976) 385–394
2. Siegel, S.F.: Efficient verification of halting properties for MPI programs with wildcard receives. In Cousot, R., ed.: VMCAI 2005. Volume 3385 of LNCS. (2005) 413–429
3. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: TACAS 2003. Volume 2619 of LNCS., Springer (2003) 553–568
4. Barrett, C., Tinelli, C.: CVC3. [12] 298–302
5. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM TOSEM **17**(2) (2008) Article 10, 1–34
6. S. F. Siegel et al.: The Toolkit for Accurate Scientific Software web page. `http://vsl.cis.udel.edu/tass` (2010)
7. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (2008)
8. Barrett, C., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.: TVOC: A translation validator for optimizing compilers. In Etessami, K., Rajamani, S.K., eds.: CAV 2005. Volume 3576 of LNCS., Springer (2005) 291–295
9. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: SPIN 2003. Volume 2648. (2003) 235–239
10. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. [12] 173–177
11. Vakkalanka, S., Sharma, S.V., Gopalakrishnan, G., Kirby, R.M.: ISP: A tool for model checking MPI programs. In: Principles and Practices of Parallel Programming (PPoPP). (2008) 285–286
12. Damm, W., Hermanns, H., eds.: CAV 2007. In Damm, W., Hermanns, H., eds.: CAV 2007. Volume 4590 of LNCS., Springer (2007)