Technical Report UD-CIS-2011/01 January 21, 2011

The Toolkit for Accurate Scientific Software

Stephen F. Siegel and Timothy K. Zirkel
Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark, DE 19716, USA

The Toolkit for Accurate Scientific Software

Stephen F. Siegel and Timothy K. Zirkel

Abstract. The Toolkit for Accurate Scientific Software (TASS) is a suite of integrated tools for the formal verification of programs used in computational science, including numerically-intensive message-passing-based parallel programs. While TASS can verify a number of standard safety properties (such as absence of deadlocks and out-of-bound array indexing), its most powerful feature is the ability to establish that two programs are functionally equivalent. These properties are verified by performing an explicit state enumeration of a model of the program(s). In this model, symbolic expressions are used to represent the inputs and the values of variables. TASS uses novel techniques to simplify the symbolic representation of the state and to reduce the number of states explored and saved. The TASS front-end supports a large subset of C, including (multi-dimensional) arrays, structs, dynamically allocated data, pointers and pointer arithmetic, functions and recursion, and other commonly used language constructs. A number of experiments on small but realistic numerical programs show that TASS can scale to reasonably large configurations and process counts. TASS is open source software distributed under the GNU Public License. The Java source code, examples, experimental results, and reference materials are all available at `http://vsl.cis.udel.edu/tass`.

1. Overview

1.1. Introduction

The *Toolkit for Accurate Scientific Software* (TASS) [27, 36] is a new suite of integrated tools for the formal verification of programs used in computational science, including parallel programs using the Message Passing Interface (MPI) [22]. TASS uses symbolic execution and model checking techniques [2, 21, 33] to verify a number of standard safety properties, including those listed in Fig. 1. Perhaps its most powerful feature is the ability to establish that two programs are functionally equivalent. This is particularly useful in the numerically-intensive scientific computing domain. Developers in this domain often start with a simple sequential encoding of an algorithm, then gradually transform it 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.

TASS verifies these properties by automatically constructing a model from the C source code(s) and exploring the reachable states of that model (see Fig. 2). For the search to terminate, the model must have a finite number of states. In practice, this means finite bounds must be placed on some program inputs and parameters. A specific value must be provided for the number of processes, and array lengths and values that determine the number of iterations of a loop must be restricted to finite

This material is based upon work supported by the National Science Foundation under Grants CCF-0733035 and CCF-0953210, and by the University of Delaware Research Foundation.

1. Every array index must lie within the array's bounds.
2. Every user-specified assertion must hold on all executions.
3. There are no memory leaks.
4. Every pointer dereferenced is a valid pointer.
5. There is no division by 0 (including with the integer modulus operator).
6. For each process, no MPI function is invoked before `MPI_Init`; if `MPI_Init` is invoked then `MPI_Finalize` will be invoked before termination; no MPI function will be invoked after `MPI_Finalize`.
7. The program can never enter a state of potential or absolute deadlock.
8. In MPI functions involving "count" arguments, such arguments are always nonnegative; if a "count" argument is positive, the corresponding buffer argument is a valid non-null pointer.
9. Any rank argument used in an MPI function call lies between 0 and $m - 1$ (inclusive), where m is the number of processes in the communicator used in that call. (Exceptions: `source` may be `MPI_ANY_SOURCE`, `source` or `dest` may be `MPI_PROC_NULL`.)
10. The element type of any message received is compatible with the type specified by the receive statement.
11. Assuming weak fairness, every message sent is eventually received.
12. Any message received does not overflow the specified receive buffer.
13. For any communicator, all processes belonging to the communicator issue the same sequence of collective calls on that communicator, in the same order, and with compatible arguments (`root`, `op`, etc.).
14. The program is functionally equivalent to another given program (the "specification").

FIGURE 1. Some of the properties verified by TASS.

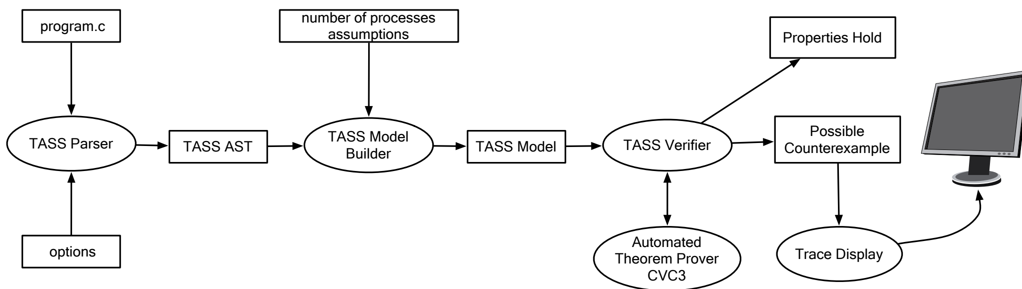


FIGURE 2. Dataflow through TASS tool chain.

ranges. Within these bounds, the analysis performed by TASS is *conservative*: if it reports a property holds, the property must hold on any execution within the specified region of the input space.

While these bounds are typically much smaller than those the program would encounter in use, they are far from trivial (see §6). Moreover, using the TASS approach, it is not necessary to scale to production-level configurations in order to detect most defects. This contrasts with testing-based verification approaches. Consider, for example, the search for deadlocks in an MPI program that uses the standard send and receive operations. If the program reaches a state in which no progress is possible without buffering a message, it may deadlock; such a state is said to be *potentially deadlocked.* Whether or not it actually deadlocks depends on choices made by the MPI implementation; the MPI Standard [22] permits an implementation to use any policy to determine whether a message should be buffered or forced to be delivered synchronously. In practice, most implementations will perform the buffering operation as long as there are sufficient resources to do so. Often, these resource limits are not reached at small scales, and so the tests do not reveal the deadlock. But when the program is run at full scale, the deadlock suddenly appears, perhaps in the middle of a long-running simulation. By contrast, TASS explores all possible behaviors permitted by the MPI Standard, and would

therefore explore the "forced synchronization" choice, and detect the deadlock, at even the smallest scales.

If a property is violated, an explicit counterexample is returned to the user in the form of a step-by-step trace through the program(s) showing the values of variables at each state. This greatly facilitates defect isolation and correction ("debugging"). It also demonstrates another advantage over standard testing and debugging techniques: providing the user with small (even minimal) counterexamples. If testing can only manifest a defect in a configuration involving thousands of processes and tens of thousands of execution steps, the user has little hope of stepping through the failing trace and arriving at an understanding of the problem.

1.2. Example: Summing the Elements of an Array

We now illustrate the use of TASS on a simple example. Figure 3 shows two C programs for summing the elements of an array. The first is a straightforward sequential program, the second a parallel version in which each process sums a contiguous section of the array. The goal is to show the two programs are functionally equivalent.

The source code has been annotated with TASS pragmas which specify the input and output signatures of the programs. These pragmas are ignored by the standard compiler that will be used to compile the code for execution, but provide information to the TASS front-end. The `input` pragma precedes a global variable declaration or preprocessor object-like macro definition to indicate that the variable or object should be considered an input to the program. In the case of macros, the literal value, such as the "`10`" used for `B` in this example, is ignored when `B` is declared to be an input. In this case, both programs have 3 inputs: integers `B` and `n`, and an array `a` of doubles of length `n`. Assumptions about the inputs may be included in the pragma; here we have assumed $0 \leq \texttt{n} \leq \texttt{B}$. Output pragmas are similar; in this case each program has one output, `sum`. Two programs with the same input-output signatures are candidates for *comparison* by TASS.

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 number of processes is obtained by the call to `MPI_Comm_size`, which stores this value in `nprocs`. A unique PID ("rank") for the process is obtained by calling `MPI_Comm_rank` and the result stored in `myrank`. Each process then determines the segment of the array for which it is responsible and sums the elements in that segment. These results are then sent to the process of rank 0, which receives them one at a time and adds them to variable `result`, which is finally assigned to the output variable `sum`.

To verify the functional equivalence of the sequential code with the 10-process instantiation of the parallel code, whenever the input array length is at most 100, the user would issue the command

```
tass compare -np2=10 -inputB=100 adder_seq.c adder_par.c
```

Several seconds later, TASS returns with a message that all properties hold, together with statistics on the number of states explored, and so on. (See Figs. 8(a) and 10 for detailed scaling data on this example.)

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 perfectly precise, i.e., spurious counterexamples are possible. The imprecision results from the need to decide certain questions, such as whether a boolean-valued symbolic expression is satisfiable, or a real-valued expression must necessarily be non-zero. For expressions involving all of the standard operations of real and integer arithmetic, these questions are undecidable in general, and hence a definitive answer is not always possible. TASS reports all possible violations, but also categorizes each by a *certainty* level ("possible" or "provable") to help the user prioritize the alarms.

```
#pragma TASS input int
#define B 10
#pragma TASS input {n>=0 && n<=B} int
#define n 10
#pragma TASS input
double a[n];
#pragma TASS 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.c`, sequential version serving as specification.

```
#include<mpi.h>
#pragma TASS input int
#define B 10
#pragma TASS input {n>=0 && n<=B} int
#define n 10
#pragma TASS input
double a[n];
#pragma TASS output
double sum;
int myrank, nprocs; double localSum = 0.0;

double computeGlobalSum() {
  double result = localSum, buffer; int i;
  for (i=1; i<nprocs; i++) {
    MPI_Recv(&buffer, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    result += buffer;
  }
  return result;
}

void main() {
  int argc; char **argv; int first, afterLast, i;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  first = n*myrank/nprocs;
  afterLast = n*(myrank+1)/nprocs;
  for (i=first; i<afterLast; i++) localSum += a[i];
  if (myrank == 0) sum = computeGlobalSum();
  else MPI_Send(&localSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
  MPI_Finalize();
}
```

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

FIGURE 3. Two functionally equivalent programs to add the elements of an array.

This example raises the issue of round-off error. Floating-point addition is not associative (even when satisfying the IEEE 754 Standard), so the results returned by the two programs will not really be identical for all possible inputs. However, TASS interprets the floating-point operations as taking place in the ideal mathematical real numbers and therefore returns a positive result. (A similar comment holds for the integers.) Would it not be better to interpret these instructions exactly as they are implemented on a real machine? It depends on the question one wants to answer. When a developer of numerical software asks whether two programs are functionally equivalent, she almost always has in mind "ignoring round-off error." Otherwise, the answer will almost always be "no," as the adder example illustrates, and a tool that only reported this negative result would not be of much use.[1]

A related issue arises when a modification is made to a complex numerical code. Tests are run, and the results turn out to be slightly different from those of the original version. The developer wants to know the reason for the difference: is it due solely to round-off error, or has the modification introduced a deeper defect? Clearly, a tool that only compares on the level of floating-point arithmetic cannot help answer this question: the results of the test already show that the two versions are not floating-point equivalent. But a tool such as TASS can provide a precise answer.

1.3. Summary

Section 2 discusses the architecture of the TASS system itself. A good deal of attention has been paid to design issues. In particular, each module has a limited, well-defined interface and can be largely understood and developed in isolation. This section summarizes the primary responsibilities of each module as well as the important relations among modules.

Section 3 deals with model extraction: the TASS representation of a model, the process by which source code is translated into that representation, the various pragmas used to inform the translation, and libraries are some of the important issues discussed.

Symbolic execution is an abstract interpretation of a program in which symbolic expressions are used in place of concrete values. Section 4 presents a formal description of symbolic execution in a general setting. Theorems are proved which bear on the soundness of several of the techniques used by TASS, including why the fact that a property holds on the symbolic level implies it must hold for all concrete executions, and why the symbolic representation of the state can be "simplified" at any point in the search.

The management of states is the subject of Section 5. The precise ways in which states are represented, stored and manipulated are key to the scalability of TASS. Care must be taken to balance concerns such as the memory footprint and the time required to execute a transition. These engineering trade-offs, and the novel solution taken by TASS are discussed here.

We have performed a number of experiments and scalability studies using the benchmarks of the FEVS suite [35]. The results of these experiments are discussed in Section 6.

Section 7 deals with related work. Conclusions and future work are discussed in Section 8.

2. Architecture

The TASS system has been designed in a modular way, with clear module boundaries having well-defined interfaces. The benefits of this approach are well-known. TASS is written in Java, and each module is implemented using one or more Java packages. Each module M has a designated sub-package (named "`IF`") providing all elements exported by M. These are the only elements of M used by other modules.

The *uses relation* on the set of modules is the irreflexive binary relation consisting of all pairs (M_1, M_2) where M_1 has at least one static reference to (i.e., use of) M_2. The TASS uses relation

[1] In fact, the output of almost any MPI program with a reduction operation using floating-point addition will not be a deterministic function of its input. This is because the values from different processes can be added in any order. Such a program will not be functionally equivalent to any program, using the strict definition.

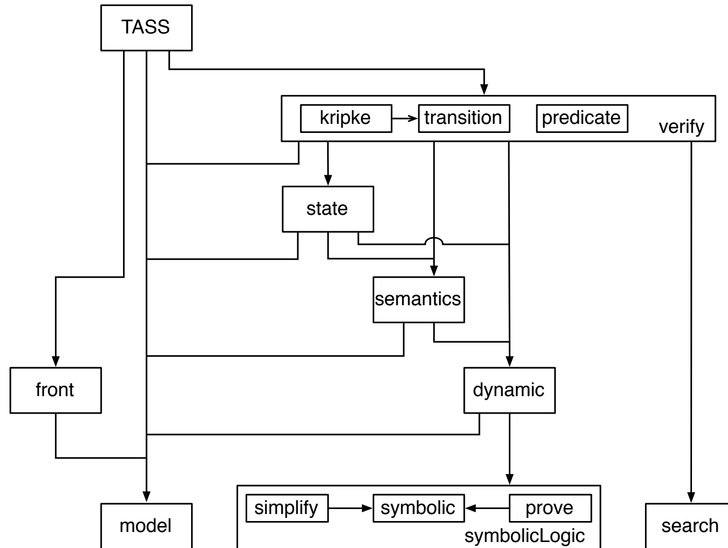


FIGURE 4. "Uses" relation on the primary modules of TASS. Modules `verify` and `symbolicLogic` are further decomposed into submodules.

is shown in Figure 4. The relation is a hierarchy, i.e., the graph contains no cycles. The hierarchical constraint is generally considered to provide many software engineering advantages [23].

We hope that these engineering decisions will encourage other researchers to contribute to the development of TASS or modify it for their use in other projects.

We now describe the responsibilities of each module in turn, working from the bottom up.

Module search provides a generic framework for performing a depth-first search of a transition system T, finding all states that satisfy a given predicate. The classes representing the states and transitions of T are generic parameters in this module. The module provides interfaces which specify a method to return an ordered set of enabled transitions from a state, a method to return the next state from a given state and transition, and so on. The user of this module need only define classes implementing these interfaces.

Module symbolicLogic is responsible for managing and reasoning about symbolic expressions. The core submodule symbolic manages the creation of symbolic expressions and provides basic operations on them. The expressions are typed. The type system includes three primitive types, *boolean*, *integer*, and *real*, and compound types *tuple*, *array*, and *function*.

Submodule prove is responsible for proving theorems expressed as boolean-valued symbolic expressions. It provides a *theorem prover* interface. The key method in this interface takes a boolean-valued symbolic expression ϕ and attempts to prove it is *valid*, i.e., that it holds for all possible assignments of concrete values to the free variables (symbolic constants) occurring in ϕ. This method returns one of three values: *yes*, *no*, or *maybe*. Nothing can be concluded from a *maybe* result. Two implementations of the theorem prover interface are currently provided: one uses CVC3 [4], the other is a fast but less precise (i.e., more likely to return *maybe*) light-weight prover. The prover ultimately used by TASS is a hybrid of these two: the light-weight prover is invoked first, and if it returns *maybe*, the CVC3 prover is called.

Submodule simplify provides methods to simplify symbolic expressions. Given an *assumption*, a boolean-valued symbolic expression ϕ, this method returns a *simplifier* for ϕ. The simplifier provides a method that takes any symbolic expression e and returns a simplified expression e' so that $\phi \Rightarrow e = e'$ is valid. For example, if ϕ is $x = 3$, and e is $x+y-1$, then e' will be $y+2$. This facility

is essential for reasoning efficiently and precisely about symbolic expressions, and for transforming each state into a canonical form.

The model module provides the TASS intermediate representation of a program. Classes are provided to represent and construct processes, procedures, expressions, statements, and so on. This module may be thought of as providing the *syntax* of a TASS model.

Module front, described in §3, is responsible for constructing a model from C source.

The dynamic module is responsible for representing all runtime values, types, and variables. A single static local variable declared in a function f will correspond to many dynamic variables at an execution state if f occurs on the call stack several times. Calls to `malloc` and `free` also lead to the creation and destruction of dynamic variables. Dynamic values generally wrap symbolic expressions but in some cases use them in more complex ways to provide certain kinds of values that may arise in C, such as pointers. Pointers are represented as ordered pairs $(i, \langle j_1, \ldots, j_m \rangle)$ where i is the ID number of a dynamic variable and the second component navigates to a point "inside" a variable of compound type. For example, if a is a dynamic variable with ID 3 and type `int[7][3]`, then $(3, \langle 1, 2 \rangle)$ is a pointer to $a[1][2]$. The same notation works for C structs, whose fields are ordered and can therefore be referenced by integers in the same way.

While the model module defines the syntax of all model elements, the semantics module defines the semantics of these elements. This module provides an *environment* interface, which is a high-level abstraction of the state of a model. An environment has methods to get and set the values of dynamic variables, to push a function onto the call stack of a process, to allocate variables on the heap, enqueue or dequeue a message, and so on. The expression semantics are defined by a method which takes an expression and an environment and returns a value; this is provided in an *evaluator* class implemented in this module. All TASS expressions are side-effect free, so the environment is not modified by this method. The statement semantics are defined by a method which takes a statement and an environment and modifies the environment accordingly. The environment interface hides the details about how states are managed. For example, one could make states immutable and implement the environment by giving it a state field and having each modification method create a new state, replacing the old one with the new. Or states could be mutable and the modification methods of the environment could modify the state accordingly. As we will see, the strategy followed by TASS is to flow between these two extremes in order to get advantages of both.

The state module provides a representation of the system state. The system state has a complex, hierarchical structure, described in §5. It comprises one or more model states and other components. This module provides an implementation of the environment interface defined in semantics.

The verify module is responsible for performing verification of a TASS system S composed of one or more models. It brings together the given model(s) and a state predicate, forms a transition system T, and then performs the depth-first search of T using the search module. Submodule kripke defines the transition system of S, providing a next-state function, and so on, in order to implement the interfaces defined in search. The transitions come in two types: a *simple* transition wraps a single statement in a model, while a *synchronous* transition comprises a paired send and receive statement in two processes. Submodule predicate provides commonly-used predicates on the state of a TASS system, including freedom from potential or absolute deadlock, and a *comparison predicate* which holds if the output variables of two models agree.

Module TASS provides a command-line interface for TASS.

3. Model Extraction

Model extraction is the process by which TASS constructs its internal representation from source code. In this section we discuss some of the salient features of this process. We begin by describing the target of this translation, the TASS *model*, and the mapping from source to model elements. This

is followed by a detailed description of the annotations the user may insert into the source code. Finally, we discuss how libraries are dealt with in the model construction process.

3.1. Structure of a TASS Model

A model consists of a set of *shared variables* and a sequence of *processes*.

A shared variable is one that is shared by all processes in the model. There are three types: *input*, *output*, and *proper* shared variables. Input variables are read-only and output variables are write-only. Hence neither can be used for inter-process communication. These variables are typically added for verification purposes only, and will not necessarily exist in the actual production code—their role is similar to that of a test harness in testing approaches. They specify the input-output signature of the program, which is required if two programs are to be compared for functional equivalence. The proper shared variables are not used at this time, though they may be in the future if TASS is extended to deal with shared-memory parallelism.

A process consists of a set of *process-global* variables and a non-empty set of *functions*, one of which is designated the *main* function. The scope of a process-global variable is the process to which it belongs; it corresponds to the usual notion of "global variable" in an MPI program. The main function is the one invoked at startup.

A function comprises a set of local variables (including the formal parameters), a return type, and a *guarded transition system*. This is a directed graph in which the nodes are *locations* and the edges *statements*. Each statement has an associated *guard*, a boolean-valued expression specifying when the statement is enabled. One location is specified to be *initial*. Statement types include *noop*, *assign*, *assert*, *assume*, *invoke*, *return*, *send*, and *receive*. Every function must include at least one return statement.

3.2. Translation

The source is parsed by a Java parser generated by the ANTLR Parser Generator [24]. The grammar used to generate the parser is a modified C99 grammar augmented to deal with pragmas and preprocessor directives. Note that the source code is not run through the C preprocessor; this is because many of the preprocessor directives are incorporated into the TASS model. For example, the user may specify that certain "constants" defined as object-like macros (e.g., `#define N 10`) be considered input variables for the purposes of verification. The object that results from the parsing phase is a TASS *abstract syntax tree*.

At each phase of translation, a detailed mapping from model elements back to the original source code is preserved, including start and end line and column numbers. This facilitates precise error reporting if a property violation is found. Also, constants that originated from macro definitions are reported using the original macro name (`N`) rather than the value (10) to enhance the readability of the reports.

Currently, all floating-point types in the source code are mapped to the single *real* type in TASS. Similarly, all integer types (`short`, `long`, etc.) are mapped to the single *integer* type. An exception is made for C's `char` type, which has a separate corresponding type of the same name in TASS. A value of this type can be displayed as an integer or as an ASCII character, depending on context.

Not every aspect of the C99 language is covered at this time. For example, there is no support for C's bit-wise operations. Use of these constructs will result in a syntax error.

3.3. Annotations

The user may annotate the source with certain pragmas to provide additional information to TASS. The pragmas are as follows:

Input-output. Normally, a variable declared in the global scope is considered a process-global variable, i.e., one instance of the variable is created for each process, and the scope of each such instance is the entire process. However, if the *input pragma* `#pragma TASS input` is placed before the declaration, this variable will be instantiated only once, as a shared input variable. Output variables are likewise declared by the pragma `#pragma TASS output`.

An optional boolean-valued expression may be attached to the input declaration. These assumptions are added to the path condition (see §4) in the initial state. They are typically used to place constraints on the input space. For example,

```
#pragma TASS input {N>0}
int N;
```

declares that `N` is an input variable and its value must be positive. The boolean-valued expression may refer to any input variables previously declared, not just the variable being declared as an input.

Certain object-like preprocessor macros can also be considered input variables. For these, the type must also be specified in the pragma, since it is not specified in the macro:

```
#pragma TASS input {N>0} int
#define N 10
```

This tells TASS that for the purpose of verification `N` should be treated as an input variable; the value 10 will be ignored.

Assertions. TASS will verify standard C assertions in the input code, but assertions can also be specified using the pragma language. The advantage of the latter is that the assertion language has many features not available in C, such as existential and universal quantifiers. For example,

```
#TASS pragma assert forall {int i | 0<=i && i<N-1} a[i]<=a[i+1]
```

asserts that the array `a` is non-decreasing. The general syntax for a quantifier expression is

(`forall` | `exists`) '`{`' (`int` | `real`) *identifier* ('`|`' *constraint*)? '`}`' *predicate*

The optional *constraint* is a boolean-valued expression used to restrict the possible values of the bound variable named *identifier*.

TASS also provides a *collective assertion* mechanism [37]. A collective assertion is analogous to a collective operation in MPI. It involves one assertion in each process. Conceptually, when control reaches a collective assertion location, a "snapshot" of the process state is taken and sent into a queue. When one snapshot from each process involved in the assertion has been queued, these snapshots are dequeued and put together to form an (artificial) global state. The asserted expressions in each process are then evaluated in this state; the collective assertion holds if and only if these hold on every process. Many useful properties of MPI programs can be expressed as collective assertions; for details and several examples, see [37]. The general syntax is

```
#pragma TASS collective assert IDENTIFIER expression
```

where `IDENTIFIER` gives a name to the assertion, which is used to match assertion statements. The expressions may refer to variables in other processes: the notation `PROC`[e], where e is an integer-valued expression, denotes the process of rank e. This can be pre-appended to any variable name to refer to a variable in that process, e.g., `PROC[(myrank+1)%nprocs].a` refers to the variable `a` in the process of rank `(myrank+1)%nprocs`. The keyword `joint` can be used in place of `collective` to indicate that the collective assertion spans two models that are being compared.

Assumptions. An assume pragma is used to add an assumption to the path condition at any point of execution. For example, `#pragma TASS assume x+y>z` causes TASS to evaluate the expression `x+y>z` and use the resulting symbolic expression p to update the path condition pc by replacing it with $pc \wedge p$. Whereas an assertion is something the user believes must hold and asks TASS to check, an assumption is the user's way of telling TASS to assume that a condition holds.

Functions. There are two different pragmas that may precede the declaration (without body) of a function f. The first, `#pragma TASS system`, declares f to be a system function; this facility is used primarily for modeling libraries and is described in §3.4. The second is used to declare f to be an abstract, side-effect-free mathematical function and is written `#pragma TASS abstract`. An abstract function is treated as an uninterpreted operation and can be used in expressions.

3.4. Libraries

Almost any non-trivial program uses libraries. Each library defines some set of constants, types, and functions. Conceptually, we may divide the functions into two categories: *standard functions* and *system functions*. Standard functions have bodies written in C and can be treated by TASS like any other function. System functions require special attention. They are typically provided by libraries that must conform to published standards, and source code may not even be available for them. Standard libraries in C include `stdlib`, `stdout`, and `math`.

When the TASS front end processes a library inclusion directive (e.g., `#include <math.h>`), it looks in certain directories for an implementation of this library. TASS provides implementations for commonly-used libraries (e.g., `math.c`) and the user may of course provide their own. Each library function can be defined in the usual way (in C), or can be declared a system function. In the latter case, no body is included and the declaration is preceded by the system function pragma. The semantics of the system function must be provided in a Java class associated with the library. That class describes the transformation performed by the function on an environment. In essence, it defines a new kind of primitive statement that will be executed as a single atomic transition. A guard can also be specified in the pragma, as in `#pragma TASS system guard x>0.0`. This means a call to that function will block until global variable `x` is positive, at which point the function may execute atomically.

Even in the case where a particular source code implementation of a function is available, there are advantages to defining these on the system level. Rather than restricting the analysis to a particular implementation of `math.h`, for example, the use of a generic model of that library enables platform-independent verification of the user's code. Also, since TASS is concerned with real (rather than floating-point) arithmetic, information can be included in the TASS model that is not available in an actual implementation, such as the axiom $\forall x \in \mathbb{R}. \sin(x)^2 + \cos(x)^2 = 1$.

4. Symbolic Execution

We now turn to the formal description of the main technique used by TASS, symbolic execution. Our approach borrows ideas from *abstract interpretation* [9, 10], though the description here is self-contained, including proofs of all claims, some of which can be found in the Appendix.

For sets X and Y, let $\mathsf{Func}(X, Y)$ denote the set of all functions from X to Y. Let Val be a set of *values*. Assume $\mathbb{B} = \{\mathit{true}, \mathit{false}\} \subseteq \mathsf{Val}$.

4.1. Program Graphs and the Concrete Transition System

Given a set of program variables V, let $\mathsf{Eval}(V) = \mathsf{Func}(V, \mathsf{Val})$. Let $\mathsf{Expr}(V)$ denote the set of expressions over V. The exact syntax of expressions is not important. The semantics of expressions are defined by a function $\mathsf{eval}_V \colon \mathsf{Expr}(V) \times \mathsf{Eval}(V) \to \mathsf{Val}$. Let $\mathsf{BoolExpr}(V)$ be the subset of $\mathsf{Expr}(V)$ consisting of all expressions of boolean type; for any $g \in \mathsf{BoolExpr}(V)$ and $\eta \in \mathsf{Eval}(V)$, $\mathsf{eval}_V(g, \eta) \in \mathbb{B}$.

Definition 1. A *program graph* over V is a tuple $(\mathsf{Loc}, \mathsf{Act}, \mathsf{effect}, \mathsf{Tran}, \mathsf{Loc}_0, g_0)$ where

1. Loc is a set of *locations* and Act is a set of *actions*,
2. $\mathsf{effect} \colon \mathsf{Act} \times \mathsf{Eval}(V) \to \mathsf{Eval}(V)$ is the *effect function*,
3. $\mathsf{Tran} \subseteq \mathsf{Loc} \times \mathsf{Cond}(V) \times \mathsf{Act} \times \mathsf{Loc}$ is the *conditional transition relation*,

4. $\mathsf{Loc}_0 \subseteq \mathsf{Loc}$ is a set of *initial locations*, $g_0 \in \mathsf{Cond}(V)$ is the *initial condition*.

The set of *states* of a program graph is the set $\mathsf{State} = \mathsf{Loc} \times \mathsf{Eval}(V)$. A state $s = \langle l, \eta \rangle$ is *initial* if $l \in \mathsf{Loc}_0$ and $\mathsf{eval}_V(g_0, \eta) = \textit{true}$. The set of initial states is denoted State_0. The *next-state function* $\mathsf{next}\colon \mathsf{State} \times \mathsf{Tran} \to \mathcal{P}(\mathsf{State})$ is defined as follows: let $s = \langle l, \eta \rangle \in \mathsf{State}$, $t = \langle l'', g, \alpha, l' \rangle \in \mathsf{Tran}$, and set

$$\mathsf{next}(s,t) = \begin{cases} \{\langle l', \mathsf{effect}(\alpha, \eta)\rangle\} & \text{if } l = l'' \text{ and } \mathsf{eval}_V(g, \eta) = \textit{true} \\ \emptyset & \text{otherwise.} \end{cases} \tag{1}$$

Define $\mathsf{Next}\colon \mathsf{State} \to \mathcal{P}(\mathsf{State})$ by $\mathsf{Next}(s) = \bigcup_{t \in \mathsf{Tran}} \mathsf{next}(s,t)$. Let $\textsc{next} = \{(s, s') \in \mathsf{State} \times \mathsf{State} \mid s' \in \mathsf{Next}(s)\}$. Let $\textsc{next}*$ denote the reflexive transitive closure of $\textsc{next}$. A state s is *reachable* if $(s_0, s) \in \textsc{next}*$ for some $s_0 \in \mathsf{State}_0$. The set of all reachable states is denoted Reach.

The notion of program graph is quite general. A parallel program may be represented as a program graph by taking Loc to be tuples whose components are the program counters for the processes. The elements of Val may represent arrays or other complex data structures. The elements of Var could be used to model heaps or call stacks. A single program graph may also represent two or more TASS models that run concurrently without interacting; this is the approach taken by TASS when comparing two models.

4.2. The Symbolic Transition System

Let $\mathcal{X}$ be a new set of variables that has no elements in common with any of the objects introduced so far. We call the elements of $\mathcal{X}$ *symbolic constants*. Let SExpr denote a set of *symbolic expressions* over $\mathcal{X}$. As in the case with program expressions, we do not specify every possible kind of symbolic expression, but instead assume we are given a function

$$\mathsf{eval}_{\mathcal{X}}\colon \mathsf{SExpr} \times \mathsf{Func}(\mathcal{X}, \mathsf{Val}) \to \mathsf{Val}$$

that specifies the semantics of symbolic expressions. Let SBoolExpr be the symbolic expressions of boolean type; these satisfy $\mathsf{eval}_{\mathcal{X}}(\phi, \theta) \in \mathbb{B}$ for any $\phi \in \mathsf{SBoolExpr}$ and $\theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val})$. We also require that SExpr contain at least the boolean literals and the symbolic constants themselves, and that these have the obvious interpretation. I.e., assume $\mathbb{B} \subseteq \mathsf{SBoolExpr}$ and $\mathcal{X} \subseteq \mathsf{SExpr}$, and that $\mathsf{eval}_{\mathcal{X}}(b, \theta) = b$ and $\mathsf{eval}_{\mathcal{X}}(X, \theta) = \theta(X)$ for all $b \in \mathbb{B}$ and $\theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val})$. Finally, we need the symbolic expressions to support conjunction. Namely, assume there is an operator $\wedge\colon \mathsf{SBoolExpr} \times \mathsf{SBoolExpr} \to \mathsf{SBoolExpr}$ with the property

$$\mathsf{eval}_{\mathcal{X}}(\phi_1 \wedge \phi_2, \theta) = \mathsf{eval}_{\mathcal{X}}(\phi_1, \theta) \wedge \mathsf{eval}_{\mathcal{X}}(\phi_2, \theta). \tag{2}$$

for any $\phi_1, \phi_2 \in \mathsf{SBoolExpr}$ and $\theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val})$.

The paragraph above specifies some facts about symbolic expressions in isolation. To be useful for symbolic execution we must relate these to program expressions. Symbolic execution works by associating symbolic expressions to program variables and we need a way to evaluate a program expression in this context to obtain a symbolic expression. We therefore assume we are given another function

$$\mathsf{seval}\colon \mathsf{Expr}(V) \times \mathsf{Func}(V, \mathsf{SExpr}) \to \mathsf{SExpr}$$

and that this function is consistent with the concrete evaluation semantics, i.e.,

$$\mathsf{eval}_V(e, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi) = \mathsf{eval}_{\mathcal{X}}(\mathsf{seval}(e, \xi), \theta) \tag{3}$$

for all $e \in \mathsf{Expr}(V)$, $\xi \in \mathsf{Func}(V, \mathsf{SExpr})$, and $\theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val})$. Here, by $\mathsf{eval}_{\mathcal{X}}(-, \theta)$, we mean the function from SExpr to Val which maps ϕ to $\mathsf{eval}_{\mathcal{X}}(\phi, \theta)$.

Example. Let $V = \{a, b\}$ and $\mathsf{Val} = \mathbb{Z} \cup \mathbb{B}$. Assume $a + b \in \mathsf{Expr}(V)$ and this expression is given the obvious semantics. Let $\mathcal{X} = \{X, Y\}$ and say SExpr contains all integer-linear combinations of the elements of $\mathcal{X}$. We use $\oplus$ for the additive operator in SExpr. Suppose $\mathsf{seval}(a + b, \xi) = \mathsf{seval}(a, \xi) \oplus \mathsf{seval}(b, \xi)$ for any $\xi \in \mathsf{Func}(V, \mathsf{SExpr})$. Consider the $\xi \in \mathsf{Func}(V, \mathsf{SExpr})$ defined by $\xi(a) = 2X \oplus Y$ and $\xi(b) = -Y$. Suppose $\theta(X) = 2$ and $\theta(Y) = 3$. Then

$$\begin{aligned}\mathsf{eval}_V(a + b, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi) &= \mathsf{eval}_V(a, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi) + \mathsf{eval}_V(b, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi)\\ &= \mathsf{eval}_{\mathcal{X}}(\xi(a), \theta) + \mathsf{eval}_{\mathcal{X}}(\xi(b), \theta)\\ &= \mathsf{eval}_{\mathcal{X}}(2X \oplus Y, \theta) + \mathsf{eval}_{\mathcal{X}}(-Y, \theta)\\ &= 7 - 3 = 4,\end{aligned}$$

whereas

$$\begin{aligned}\mathsf{eval}_{\mathcal{X}}(\mathsf{seval}(a + b, \xi), \theta) &= \mathsf{eval}_{\mathcal{X}}(\mathsf{seval}(a, \xi) \oplus \mathsf{seval}(b, \xi), \theta)\\ &= \mathsf{eval}_{\mathcal{X}}(2X \oplus Y \oplus -Y, \theta)\\ &= \mathsf{eval}_{\mathcal{X}}(2X, \theta) = 4.\end{aligned}$$

Hence seval satisfies (3) for these specific values (and of course in general).

The set of *symbolic states* is the set

$$\mathsf{SState} = \mathsf{SBoolExpr} \times \mathsf{Loc} \times \mathsf{Func}(V, \mathsf{SExpr}).$$

The first component is the *path condition*, the second a location, and the third a function assigning a symbolic expression to each program variable.

The set of *symbolic initial states* is defined as follows. For each program variable $v \in V$, we choose a symbolic constant $X_v \in \mathcal{X}$ in such a way that the mapping $\xi_0 \colon V \to \mathsf{SExpr}$ defined by $\xi_0(v) = X_v$ is injective. (Assume $|\mathcal{X}| \geq |V|$, so this is possible.) Let

$$\mathsf{SState}_0 = \{\langle \phi_0, l_0, \xi_0 \rangle \mid l_0 \in \mathsf{Loc}_0\},$$

where $\phi_0 = \mathsf{seval}(g_0, \xi_0)$.

In the same way that we needed to define the semantics of symbolic expression evaluation by relating it to program expression evaluation, we must also define the semantics of statement execution on the symbolic level. Hence we assume we are given a *symbolic effect function*

$$\mathsf{seffect} \colon \mathsf{Act} \times \mathsf{Func}(V, \mathsf{SExpr}) \to \mathsf{Func}(V, \mathsf{SExpr})$$

which is consistent with the concrete effect function, i.e., which satisfies

$$\mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \mathsf{seffect}(\alpha, \xi) = \mathsf{effect}(\alpha, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi) \tag{4}$$

for all $\alpha \in \mathsf{Act}$, $\xi \in \mathsf{Func}(V, \mathsf{SExpr})$, and $\theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val})$.

The symbolic next-state function $\mathsf{snext} \colon \mathsf{SState} \times \mathsf{Tran} \to \mathcal{P}(\mathsf{SState})$ is defined as follows: given $\hat{s} = \langle \phi, l, \xi \rangle \in \mathsf{SState}$ and $t = \langle l'', g, \alpha, l' \rangle \in \mathsf{Tran}$, define

$$\mathsf{snext}(\hat{s}, t) = \begin{cases} \{\langle \phi \wedge \mathsf{seval}(g, \xi), l', \mathsf{seffect}(\alpha, \xi) \rangle\} & \text{if } l = l'' \\ \emptyset & \text{otherwise.} \end{cases} \tag{5}$$

Define $\mathsf{SNext} \colon \mathsf{SState} \to \mathcal{P}(\mathsf{SState})$ by $\mathsf{SNext}(\hat{s}) = \bigcup_{t \in \mathsf{Tran}} \mathsf{snext}(\hat{s}, t)$. Let $\textsc{snext} = \{(\hat{s}, \hat{s}') \in \mathsf{SState} \times \mathsf{SState} \mid \hat{s}' \in \mathsf{SNext}(\hat{s})\}$. Let $\mathsf{SReach} = \{\hat{s} \in \mathsf{SState} \mid \exists \hat{s}_0 \in \mathsf{SState}_0 . (\hat{s}_0, \hat{s}) \in \textsc{snext}*\}$.

4.3. Concretization

The purpose of this section is to relate the symbolic state space to the concrete one. Each symbolic state $\hat{s}$ determines a set of concrete states $\gamma(\hat{s})$, where $\gamma\colon \mathsf{SState} \to \mathcal{P}(\mathsf{State})$ is defined by

$$\gamma(\langle \phi, l, \xi \rangle) = \{\langle l, \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi \rangle \mid \theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val}) \wedge \mathsf{eval}_{\mathcal{X}}(\phi, \theta)\}. \tag{6}$$

Note that $\gamma(\langle \phi, l, \xi \rangle) = \emptyset$ if and only if ϕ is unsatisfiable, i.e., $\mathsf{eval}_{\mathcal{X}}(\phi, \theta)$ is *false* for all θ.

The following lemma relates the initial states in the symbolic and concrete worlds. It says that γ maps initial states to sets of concrete initial states, and all concrete initial states are covered in this way. The proof of this lemma and other theorems in this paper are given in the Appendix.

Lemma 2. $\bigcup_{\hat{s}_0 \in \mathsf{SState}_0} \gamma(\hat{s}_0) = \mathsf{State}_0$.

The next goal is to relate the symbolic and concrete *next-state* functions. Starting with a symbolic state and a transition, there are two ways one may arrive at a set of concrete "next states": concretize first, then gather all the next states of those concrete states, or apply the symbolic next state operator and then concretize. The following assert that both approaches yield the same result:

Lemma 3. *Let* $\hat{s} \in \mathsf{SState}$ *and* $t \in \mathsf{Tran}$. *Then*

$$\bigcup_{\hat{s}' \in \mathsf{snext}(\hat{s}, t)} \gamma(\hat{s}') = \bigcup_{s \in \gamma(\hat{s})} \mathsf{next}(s, t). \tag{7}$$

Lemma 4. *Let* $\hat{s} \in \mathsf{SState}$. *Then* $\bigcup_{\hat{s}' \in \mathsf{SNext}(\hat{s})} \gamma(\hat{s}') = \bigcup_{s \in \gamma(\hat{s})} \mathsf{Next}(s)$.

Proof. This follows from Lemma 3 by applying the union over all $t \in \mathsf{Tran}$ to both sides of (7). □

Together these imply a precise relation between the reachable concrete and symbolic states:

Theorem 5. $\bigcup_{\hat{s} \in \mathsf{SReach}} \gamma(\hat{s}) = \mathsf{Reach}$.

The practical consequence of Theorem 5 is that we may use symbolic execution to verify certain safety properties of the program graph. Suppose π is a predicate on State representing some "bad" quality, and the goal is to verify that $\pi(s)$ does not hold on any $s \in \mathsf{Reach}$. Assume there is some way to "lift" π to a predicate $\hat{\pi}$ on SState such that for all $\hat{s} \in \mathsf{SState}$,

$$(\exists s \in \gamma(\hat{s}).\pi(s)) \Rightarrow \hat{\pi}(\hat{s}). \tag{8}$$

If one can show that $\hat{\pi}(\hat{s})$ does not hold for any $\hat{s} \in \mathsf{SReach}$, one may conclude from Theorem 5 that the desired property holds. We record this as

Corollary 6. *Let* π *be a predicate on* State *and* $\hat{\pi}$ *a predicate on* SState *satisfying* (8). *If* $\exists s \in \mathsf{Reach}.\pi(s)$ *then* $\exists \hat{s} \in \mathsf{SReach}.\hat{\pi}(\hat{s})$.

Proof. Assume $s \in \mathsf{Reach} \wedge \pi(s)$. By Theorem 5, $s \in \gamma(\hat{s})$ for some $\hat{s} \in \mathsf{SReach}$. By (8), $\hat{\pi}(\hat{s})$. □

The problem is that in general it is not possible to lift π to some $\hat{\pi}$ such that the converse of (8) also holds. An example illustrates this. Let $b \in \mathsf{Loc}$ and define π by $\pi(\langle l, \eta \rangle) \Leftrightarrow l = b$. We can define $\hat{\pi}(\langle \phi, l, \xi \rangle) \Leftrightarrow l = b$ and this satisfies (8). It does not necessarily satisfy the converse, because it is possible that ϕ is not satisfiable, so $\gamma(\langle \phi, l, \xi \rangle)$ is empty. The upshot is that symbolic execution may return a spurious counterexample to a safety property. However, if it concludes the property holds on all reachable symbolic states, it must hold on all reachable concrete states.

4.4. Vacuity, Equivalence, and the Role of the Theorem Prover

We now turn our attention to concepts that are vital for improving the precision of symbolic reasoning and the scalability of symbolic execution. The first is *vacuity*, the detection of which can limit the false alarms discussed above. The second is a notion of *equivalence* on symbolic states, which can be used to greatly reduce the number of symbolic states explored.

4.4.1. Vacuity and the Theorem Prover. A symbolic state $\hat{s} = \langle \phi, l, \xi \rangle \in \mathsf{SState}$ is *vacuous* if ϕ is not satisfiable. From (6) it is clear that $\hat{s}$ is vacuous if and only if $\gamma(\hat{s}) = \emptyset$. Suppose we have a procedure that can determine in some cases that a symbolic expression of boolean type is unsatisfiable. (This is the crucial service provided by the automated theorem provers used in symbolic execution.) Represent this procedure as a function $\mathsf{nsat}\colon \mathsf{SBoolExpr} \to \mathbb{B}$ that is *conservative*: if $\mathsf{nsat}(\phi)$ is *true* then ϕ is not satisfiable. Note that nsat need not be fully precise, i.e., it is possible that $\mathsf{nsat}(\phi)$ is *false* but ϕ is not satisfiable. Using nsat, one can replace the symbolic next-state function (5) with

$$\mathsf{snext}(\hat{s}, t) = \begin{cases} \emptyset & \text{if } l \neq l'' \vee \mathsf{nsat}(\phi \wedge \mathsf{seval}(g, \xi)) \\ \{\langle \phi \wedge \mathsf{seval}(g, \xi), l', \mathsf{seffect}(\alpha, \xi)\rangle\} & \text{otherwise.} \end{cases} \tag{9}$$

Hence we eliminate transitions that lead to vacuous states. It is not hard to see that Lemma 3 remains valid with this new definition: we have simply removed occurrences of $\emptyset$ from the union on the left hand side of (7), while the right hand side is unchanged. Hence Corollary 6 remains valid as well. The more precise nsat, the less likely symbolic execution is to return a spurious counterexample.

One can also use nsat to lift certain properties, such as assertions, from the concrete to the symbolic space. Suppose $e \in \mathsf{BoolExpr}$, $l_1 \in \mathsf{Loc}$, and π is the predicate on State defined by $\pi(\langle l, \eta \rangle) \Leftrightarrow (l = l_1 \wedge \mathsf{eval}(e, \eta))$. Define a predicate $\hat{\pi}$ on SState by $\hat{\pi}(\langle \phi, l, \xi \rangle) \Leftrightarrow (l = l_1 \wedge \neg \mathsf{nsat}(\mathsf{seval}(e, \xi)))$. We claim $\hat{\pi}$ satisfies (8). To see this, suppose $s = \langle l, \eta \rangle \in \gamma(\hat{s})$, where $\hat{s} = \langle \phi, l, \xi \rangle$, and $\pi(s)$ holds. By (6), there is some $\theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val})$ such that $\eta = \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi$. Since $\pi(s)$ holds, $l = l_1$ and $\mathsf{eval}(e, \eta)$ holds. Hence

$$\mathsf{eval}_{\mathcal{X}}(\mathsf{seval}(s, \xi), \theta) = \mathsf{eval}_V(e, \mathsf{eval}_{\mathcal{X}}(-, \theta) \circ \xi) = \mathsf{eval}_V(e, \eta)$$

holds as well, which shows that $\mathsf{seval}(s, \xi)$ is satisfiable. Since nsat is conservative, this implies $\mathsf{nsat}(\mathsf{seval}(s, \xi)) =$ *false*, i.e., $\hat{\pi}(\hat{s})$ holds, proving (8). All of the properties checked by TASS (see Fig. 1) fall into this category, and are verified in this way.

4.4.2. Equivalence. The symbolic states $\hat{s}, \hat{s}' \in \mathsf{SState}$ are *equivalent*, written $\hat{s} \sim \hat{s}'$, if $\gamma(\hat{s}) = \gamma(\hat{s}')$. Clearly, $\sim$ is an equivalence relations on SState and any two vacuous states are equivalent. Let $\mathsf{QState} = \mathsf{SState}/\sim$. Write $[\hat{s}]$ for the equivalence class containing $\hat{s}$. Define $\bar{\gamma}\colon \mathsf{QState} \to \mathsf{State}$ by $\bar{\gamma}([\hat{s}]) = \gamma(\hat{s})$. Define $\mathsf{qnext}\colon \mathsf{QState} \times \mathsf{Tran} \to \mathcal{P}(\mathsf{QState})$ by

$$\mathsf{qnext}(\sigma, t) = \{\sigma' \in \mathsf{QState} \mid \exists \hat{s} \in \sigma, \hat{s}' \in \sigma'. \hat{s}' \in \mathsf{snext}(\hat{s}, t)\}. \tag{10}$$

Let $\mathsf{QState}_0 = \{[\hat{s}_0] \mid \hat{s} \in \mathsf{SState}_0\}$, $\mathsf{QNext}(\sigma) = \bigcup_{t \in \mathsf{Tran}} \mathsf{qnext}(\sigma, t)$, $\textsc{qnext} = \{(\sigma, \sigma') \in \mathsf{QState} \times \mathsf{QState} \mid \sigma' \in \mathsf{QNext}(\sigma)\}$, and $\mathsf{QReach} = \{\sigma \in \mathsf{QState} \mid \exists \sigma_0 \in \mathsf{QState}_0. (\sigma_0, \sigma) \in \textsc{qnext}*\}$. We can now show the following, the proof of which is given in the Appendix:

Theorem 7. $\bigcup_{\sigma \in \mathsf{QReach}} \bar{\gamma}(\sigma) = \mathsf{Reach}$.

The importance of Theorem 7 is that it allows a symbolic state to be replaced by an equivalent state at any point in the search of the symbolic state space. Doing so still guarantees that at least one representative from each equivalence class of symbolic states will be explored. TASS takes advantage of this fact by replacing a state with a "simplified" version of the state whenever a state is saved. The goal is to increase the precision of the analysis and possibly decrease the number of states explored by making it more likely to recognize a state as equivalent to one seen before. This is made precise in the nondeterministic procedure of Fig. 5.

This procedure is not guaranteed to terminate, since the number of reachable symbolic states may be infinite, whence the need for bounds on certain variables. Even if QState is finite, the procedure may not terminate, since poor choices at line 14 could lead to repeated visits to the same equivalence class. Effective choices approximate, to the extent possible, a canonical representation of the symbolic state in order to decrease the number of such repeated visits. No matter what choices are made, however, the algorithm is "safe," in the following sense:

```
1  procedure main(PG, π̂): 𝔹 is
2      Seen ← ∅;
3      foreach l_0 ∈ Loc_0 do
4          if dfs(⟨φ_0, l_0, ξ_0⟩) then
5              return true;
6      return false;
```

```
7  procedure dfs(ŝ = ⟨φ, l, ξ⟩): 𝔹 is
8      if ŝ ∈ Seen then return false;
9      Seen ← Seen ∪ {ŝ};
10     if π̂(ŝ) then return true;
11     foreach t ∈ {⟨l_1, g, α, l'⟩ ∈ Tran | l_1 = l} do
12         if ¬nsat(φ ∧ seval(g, ξ)) then
13             ŝ' ← ⟨φ ∧ seval(g, ξ), l', seffect(α, ξ)⟩;
14             choose ŝ'' ∈ SState such that ŝ'' ∼ ŝ';
15             if dfs(s'') then return true;
16     return false;
```

FIGURE 5. The symbolic execution procedure. Given a program graph PG and a predicate $\hat{\pi}$ on symbolic states, performs a depth-first search of QState, returning *true* if a state satisfying $\hat{\pi}$ is found.

Theorem 8. *Let* PG *be a program graph,* π *a predicate on* State*, and* $\hat{\pi}$ *a predicate on* SState *satisfying* (8)*. Consider any execution of the algorithm of Fig.* 5*. If* main *returns false, then* $\pi(s)$ *does not hold for any* $s \in$ Reach.

Proof. If main returns false then the algorithm has completed a depth-first search (possibly visiting some states more than once) of the quotient space QState, checking $\hat{\pi}$ in at least one representative from each $\sigma \in$ QState. It follows that for all such σ, there is some $\hat{s} \in \sigma$ such that $\hat{\pi}(\hat{s})$ is *false*. From (8), $\pi(s)$ is *false* for all $s \in \gamma(\hat{s})$. By Theorem 7, $\pi(s)$ is *false* for all $s \in$ Reach. □

Note that there is no guarantee that $\hat{\pi}$ takes the same value on every element of a class $\sigma \in$ QState. But if $\hat{\pi}$ is *true* for one element and *false* for another, then π holds at no concrete state represented by σ—this is implied by (8) as long as there is one element of σ for which $\hat{\pi}$ is *false*. The *true* value is an instance of imprecision, and will lead to a "false alarm." Again, one of the goals of simplification is to reduce the likelihood of such alarms.

In the next section, we examine the various ways TASS transforms states into equivalent forms.

4.5. Symbolic Expressions in TASS

As described in §2, the symbolic expressions used in TASS are typed. The concrete primitive values $\mathbb{B}$, $\mathbb{Z}$, and $\mathbb{Q}$ are all included in SExpr. Here $\mathbb{Z}$ and $\mathbb{Q}$ are disjoint and represent the mathematical integers and rationals, respectively. These are represented with infinite precision in TASS (using Java's `BigInteger` class); rationals are represented as a quotient of two integers with greatest common divisor 1.

Most of the symbolic operations correspond directly to operator expressions in the model language, including $+$, $-$, $*$, $/$, $\%$, array indexing, and so on. As the (concrete) semantics of a TASS model uses the mathematical integers and reals, the requirements (3) and (4) are easily seen to hold in most cases.

Arrays are somewhat more involved. An array type $T = E[\nu]$ is specified by an element type E and the array *extent* ν, a symbolic expression of integer type. TASS provides array-read and array-write expressions that correspond directly to those program operations. However, it also provides a *concrete array expression* $\chi\{\lambda_0, \ldots, \lambda_n\}$, where χ is any expression of type T, n is a (concrete) nonnegative integer, and each λ_i is either an expression of type E or the special symbol null. This expression represents the result of modifying χ by assigning λ_i to index i for each i for which $\lambda_i \neq$ null. This kind of expression is redundant, in the sense that such an expression is

always equivalent to one expressed as a sequence of array-write operations. However, it is included to improve performance and the precision of reasoning about arrays. The array modification operation seffect will return a concrete array expression whenever a concrete value can be extracted from the given index expression. In the extreme case where all such indices are concrete, the performance of reading and modifying the array is comparable to that of an ordinary concrete array.

TASS uses various techniques to replace a state with an equivalent state, in order to approximate a canonical representation of the state. One way it does this is by transforming each symbolic expression into a canonical form. Suppose $\chi_1, \chi_2 \in \mathsf{SExpr}$ are *equivalent*, i.e., $\mathsf{seval}(\chi_1, \theta) = \mathsf{seval}(\chi_2, \theta)$ for all $\theta \in \mathsf{Func}(\mathcal{X}, \mathsf{Val})$. Suppose $\hat{s}$ is a symbolic state which assigns χ_1 to variable v. Let $\hat{s}'$ be the state which assigns χ_2 to v, but which is otherwise the same as $\hat{s}$. It follows that $\hat{s} \sim \hat{s}'$. Hence we may always replace a symbolic expression with an equivalent one without changing the equivalence class of the state.

TASS uses a specific canonical form for boolean, integer, and real expressions. The canonical form is not guaranteed to produce a unique representative of the equivalence class of the expression, but it provides sufficient precision to solve most problems that arise in practice.

For expressions of boolean type, a variant of conjunctive normal form is used for the canonical form. Each such expression has the form $\bigwedge_i \bigvee_j b_{i,j}$, where each $b_{i,j}$ is either a *literal*, *quantifier*, or *relational* expression. A literal expression is either p or $\neg p$, where p is a *primitive* boolean expression. A boolean primitive is either a symbolic constant, a record-read, array-read, or functional evaluation of boolean type. A quantifier expression has the form $\forall x.e$ or $\exists x.e$, where e is a boolean expression and x is a symbolic constant of either integer or real type. A relational expression has one of the following forms: $f > 0$, $f = 0$, $f \geq 0$, or $f \neq 0$, where f is a numeric expression. A total order is imposed on SExpr and this is used to order the $b_{i,j}$ in a canonical way. Further simplifications take place within this form, e.g., $\forall x.(f_1 \wedge f_2)$ is transformed to $(\forall x.f_1) \wedge (\forall x.f_2)$.

An expression of integer type is represented as a polynomial $\sum_{i_1,\ldots,i_n} a_i x_1^{i_1} \cdots x_n^{i_n}$, where each $a_i \in \mathbb{Z}$ and the x_i are integer primitive expressions, i.e., any expression other than an addition, subtraction, multiplication, or concrete expression.

An expression of real type is represented as a quotient p/q of two polynomials. TASS makes an effort to find and cancel common factors in p and q, but does not do this with full precision because of the expense involved in factoring polynomials. Instead, factorizations are stored with each polynomial, and when two are multiplied the resulting factorization is stored with the product. Again, this compromise appears to provide sufficient precision for most problems that actually arise.

The transformation of every symbolic expression into canonical form occurs whenever any symbolic operation is performed. A more expensive procedure is *state simplification*, which is only performed at certain times, such as before a state is saved. It is also guaranteed to produce an equivalent state, but not just by replacing the symbolic values of variables with equivalent values. An example illustrates this:

Example. Suppose $V = \{a, b\}$, $\mathcal{X} = \{X_0, X_1, X_2\}$, the path condition ϕ is $X_0 + X_1 + X_2 \leq 6 \wedge X_0 + X_1 + X_2 \geq 6 \wedge X_0 - 2X_1 + X_2 = 18$, $\xi(a) = X_0^2 + X_1^2 + X_2^2$, $\xi(b)$ is the boolean expression $X_1 > 0$, and $\hat{s} = \langle \phi, l, \xi \rangle$. The TASS simplify routine will return the state $\hat{s}' = \langle \phi', l, \xi' \rangle$, where ϕ' is $X_0 + X_2 = 10$, $\xi'(a) = X_0^2 + X_2^2 + 16$, and $\xi'(b) = \mathit{false}$. Note $\hat{s}' \sim \hat{s}$ and $\hat{s}'$ has replaced X_1 with -4 wherever it occurs in the state.

Simplification begins by parsing the path condition ϕ to produce a *simplifier*, an object used to store information extracted from ϕ and to cache results of simplifications made under assumption ϕ. Among other data, this produces an interval analysis for all monic polynomials extracted from ϕ. In the example, this introduces the equality $X_0 + X_1 + X_2 = 6$. The set of all such equalities extracted from ϕ is considered to be a linear system in the monomials. This system is simplified using Gaussian elimination to determine concrete values for certain monomials; in the example this

determines $X_1 = -4$. The path condition is simplified by substituting the concrete value for the monomial. All of this information is stored in the simplifier for ϕ.

The simplifiers themselves are cached, so when a state is to be simplified, TASS looks to see if there is already a simplifier for the path condition used in that state. In any case, the simplifier is then used to simplify each expression e occurring in the state, using the results of the interval analysis and other information stored in the simplifier. The result of the simplification of e is cached in the simplifier.

The built-in "light-weight" theorem prover used by TASS also relies on simplification. Given as assumption ϕ and predicate ψ, both in SBoolExpr, the goal is to determine whether $\phi \Rightarrow \psi$ is valid, i.e., true for all assignments of concrete values to symbolic constants. TASS obtains the simplifier for ϕ, applies it to ψ, and if the result is *true* or *false*, returns the result. Otherwise, it invokes the "heavy-weight" theorem prover (CVC3). In any case, the result of the query is cached.

A small experiment indicates the benefits of simplification. A simple parallel 1d-diffusion simulation is one of the standard examples used in our scalability study (§6). We used TASS to verify this code with 5 processes, the number of cells bounded above by 20, and an upper bound of 4 on the number of time steps. With simplification turned on, there are a total of 88 distinct queries (i.e., assumption-predicate pairs) and 73 calls to CVC3, with a total runtime of 7.5 seconds. Without simplifying the state or using the simplifier to resolve queries, there are 4200 distinct queries, 4010 calls to CVC3, and runtime has increased to 37.7 seconds. The number of distinct queries is much larger because many queries, if simplified, reduce to the same query. There are many more calls to CVC3 because simplification often reduces the predicate in a query to *true* or *false*, making a theorem-prover call unnecessary.

5. States

The TASS execution engine performs a depth-first search of the reachable symbolic states of a TASS system. Systems with hundreds of millions of states are not uncommon, and the ability to efficiently store states and execute transitions is key to the scalability of the approach. We begin by describing the structure of a state, then turn to a discussion of the engineering issues related to the sharing and mutability of state components. The solution used by TASS, which we call the *morphic pattern*, is then described.

5.1. Structure of the State

The state has a hierarchical structure, as depicted in Fig. 6. At the highest level is the *system state*, which consists of the state of one or more models, the path condition, and the state of the queue used to check collective assertions. (Currently, the number of models is either one or two, depending on whether TASS is used in *verify* or *compare* mode.) The path condition is a symbolic value of boolean type which records the branch choices made by all processes (see §4).

Each entry in the collective queue is a *collective record* which records a collective assertion (associated with a unique string identifier) and an array of process states, some of which may be null. Each time a process enters a collective assertion the queue is updated by taking a snapshot of the process' state and inserting it in the correct place in (or adding a new entry to) the queue. When an entry is complete, it is dequeued and the assertion checked. For details, see [37].

The description of the shared variables and process state is clear. An entry in the stack is created each time a function is called (invoked), and popped when the function returns. The heap is an array of values, each corresponding to a single call to `malloc`. When a heap-allocated object is freed, its entry becomes null. The null entries are cleaned up and the heap objects placed in a canonical order (and the references to them updated) in a garbage collection/heap canonicalization procedure that occurs when the state is simplified.

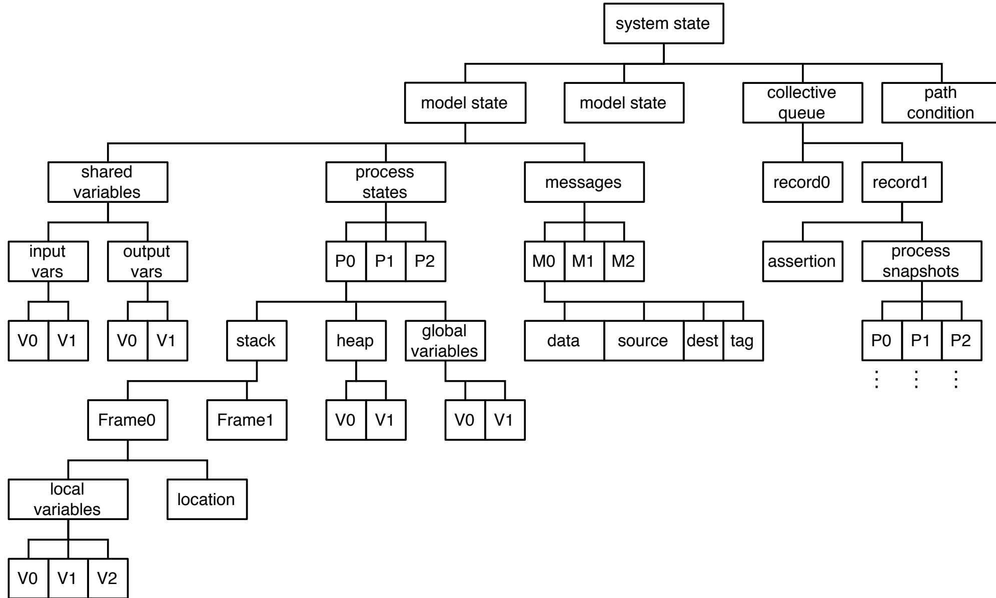


FIGURE 6. Structure of the State.

All the buffered messages are organized into a single array. They are ordered first by sender, then by receiver, then by the order in which they were issued.

5.2. Sharing, Mutability, and Flyweighting

In this section we describe how TASS manages the creation, modification, and storage of states. Model checking tools deal with these issues in various ways, making different time-memory trade-offs. Memory is often a limiting factor because of the need to record states that have been seen; total execution time is often determined by the time it takes to execute a transition. In a typical explicit-state model checker, a large component of the time is devoted to computing hash functions on the state. We first discuss several engineering choices and their impact on time and memory, and in the next section present the approach taken by TASS.

Sharing. The memory required to store states seen in the depth-first search is often a limiting factor in model checking. This is a particular challenge for TASS, where the states are hierarchical objects with a large number of components. At the same time, there are many cases where two states differ in only one or two components. Hence there is enormous advantage is allowing states to share components. Sharing can also decrease the time required to execute a transition, since instead of copying all of the common data from the old state to the new, the new state can just use a reference to the common components. This is illustrated in Fig. 7(a). The disadvantage to sharing arises when one needs to modify a component: if the modification is not intended to apply to all objects that could possibly be sharing the component, a new object must be created with the modified data. For example, if a state has been stored in a set of seen states or the DFS stack, none of its components should be modified, so it cannot be shared by any object that might perform a modification.

Mutability. The question here is whether all states and their components should be immutable, i.e., not modifiable after creation. In place of any action that would modify a component, a new component must be created. The advantages of the immutable pattern are well-known: e.g., there is no problem sharing common substructures among different objects or concurrently executing threads; the hash function of a component can be computed once and cached as a field in that component, since it will never change. Immutability can also reduce the time devoted to hashing. In a hierarchical

structure such as a TASS state, the hash can be defined as a function (such as the sum) of the hashes of the children. When creating a new "parent" object from given "children" nodes, if the hash of the children are cached, the time required to compute the hash of the new object is proportional only to the number of children (and not all descendants).

The main disadvantage of immutability is the time it takes to perform a modification. Since the state cannot be modified, each such action must produce a new state. Suppose one wishes to change the value of the local variable 0 in frame 0 of process 0 of model state 0 in Fig. 6. To do so requires the creation of a new local variable array, a new frame, a new stack, a new process state, a new process state array, a new model state, and finally a new system state. In general, a modification to a node requires that every ancestor node in the hierarchy be re-created. Imagine a case in which 200 such modifications are made in row, and there is no need to share, store, or hash the intermediate 199 states. In such a scenario, the immutable pattern leads to a large amount of unnecessary work. In contrast, if the state (and all its components) are mutable, the work required to perform the 200 modifications is minimized: no new objects need to be instantiated, only the appropriate fields updated. But such an object cannot be safely shared, and its hash function cannot be safely cached.

Flyweighting. It is often the case that two distinct objects in a class are considered to represent the "same" conceptual object. This is the case with TASS states and their components. The notion of "same" defines an equivalence relation $\sim$ on the set S of objects belonging to the class. In Java, this relation is codified by implementing an `equals` method for the class. The hash function must be defined so that it is compatible with `equals`, i.e., such that $x \sim y \Rightarrow \mathsf{hash}(x) = \mathsf{hash}(y)$. The *flyweight pattern* provides a factory which returns a unique representative from each equivalence class. It can be implemented using a hash table in which each key-value pair has the form (y, y) where y is a canonical representative of its equivalence class. When a method generates a new object x, it looks to see if there is already an entry in the hash table for key x, i.e., an entry (y, y) where $x \sim y$. If there is such an entry, the method returns y. Else it adds (x, x) to the table and returns x.

The obvious advantage of flyweighting is conservation of memory. Another advantage arises when there is a need to associate certain data to an equivalence class, rather than to a particular member. For example, we might want a boolean flag to indicate whether a state in this equivalence class has been seen before, or is currently on the DFS stack. This *extrinsic data* can be stored in the canonical representative of the equivalence class; we call the data *extrinsic* because it is not used in the hash or equals methods of the class. There are many places where extrinsic data is needed for state components in TASS; in addition to the flags mentioned above, symbolic simplification results are cached in each boolean (symbolic) value used as an assumption. The disadvantage is the cost associated to executing the `hashCode` and `equals` functions when an object is flyweighted. However, if a state is to be saved then something like that must be done at some point.

5.3. The Morphic Pattern

The morphic pattern provides a way to organize the creation, sharing, and flyweighting of objects organized in a hierarchy. It aims to achieve the following goals:

1. a hash function is only computed when it is needed and the result is cached whenever it is safe to do so,
2. flyweighting is supported at every node in the state hierarchy, and in such a way that the time required to flyweight an object is proportional to the number of its components not yet flyweighted, and
3. all sub-structures that can be safely shared are shared; those that cannot are not.

The classes making up the hierarchy are *morphic classes* and instances of those classes are *morphic objects*. Each morphic class has an associated *morphic factory* used to create and manipulate the

objects of its class. Each field in a morphic class is either *intrinsic* or *extrinsic*; the latter can have no bearing on the equals or hash methods.

A morphic object starts in a mutable state, but it and its descendants become immutable when a method `commit` is invoked. A method `isCommitted` tells whether the object has been committed. Any method in the morphic class that can modify intrinsic data first checks that the object is mutable; if not, an exception is thrown. Extrinsic data can be modified in any case.

The hash method is implemented in such a way that if the object is mutable, the function is computed as usual, typically by summing the hashes of its children and other intrinsic data. If the object is immutable, the method first determines if the hash has been cached and, if so, returns the cached value; else it computes the result and caches it.

The factory provides methods to "modify" the objects it controls, where "modify" may involve creating new components if certain components are immutable. The details are hidden from the user. Returning to the example of the local variable modification, assume that the stack (and its descendants) are committed, but process state 0 is not committed. A modification to the local variable would entail the creation of a new local variable array, a new frame, and a new stack. But a new process state will not be created; instead, the stack field will be assigned the newly created stack, and that completes the modification.

The idea is that when a morphic object is used in a context where sharing is not a concern (e.g., if the object is only reachable by a single method, or single thread), it can be mutable. If the object needs to be "released" into an environment where it may be shared, it must first be committed.

At any time, each equivalence class of morphic objects has at most one object which is the canonical representative of that equivalence class. Such an object is called *canonic*. A canonic object is always committed. Furthermore, all descendants of a canonic object are canonic. A method `isCanonic` in the morphic class is used to determine whether an object is canonic.

Every morphic factory provides a method `canonic` which takes a morphic object and returns the canonic representative of its equivalence class. These canonic objects are the ones stored by the factory. If the given object is not committed, it may be committed by this method; also, the method may return the given object. It is implemented recursively, by obtaining the canonic representative of each child (if it is not already canonic) and building up a new object from those children if necessary.

The equals method is optimized for canonic objects: if two elements are canonic, this method simply tests whether they are the same object.

We now describe an example of how the morphic pattern is used in TASS. In performing the depth-first search of a TASS system, there are often cases where a sequence of transitions can be executed as a single atomic action. This is the case, for example, if these operations only involve the state of a single process and there are no nondeterministic choices or transitions that could block. Only the final state resulting from this sequence needs to be "released" to the DFS stack. The intermediate states will never be shared, and so the sequence of transitions can all take place on mutable objects. Only the final state is committed and then returned to the stack.

A second example illustrates the use of canonicalization: when a new state s has been computed, TASS must first determine if s has been seen before and, if not, record that it has now been seen. This is implemented by invoking the canonic method on s to return the canonic representative t. A boolean field `seen` in t is then read to determine if the state has been seen before. If not, the flag is set to *true*.

The morphic pattern is supported in TASS by a module that provides an interface `Morphic` to be implemented by all morphic objects; a generic interface `MorphicFactoryIF<E extends Morphic>` for morphic factories, and abstract classes providing partial implementations of those interfaces. The module also provides morphic classes and factories corresponding to compound data structures, such as arrays, vectors, and sets. This greatly simplifies the creation of new morphic classes from old.

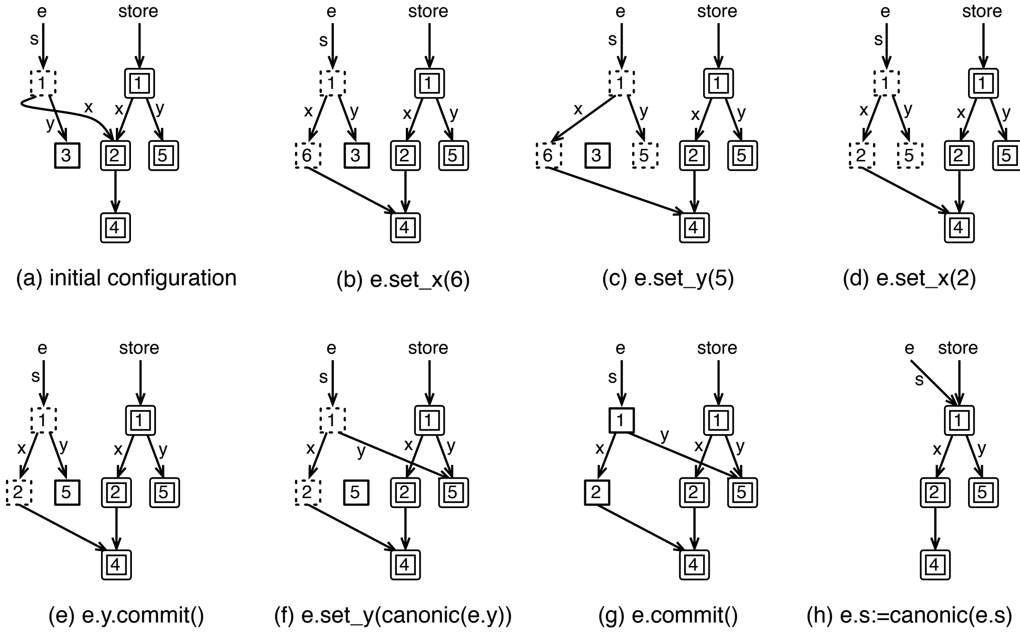


FIGURE 7. Morphic pattern. An environment e wraps a state s whose components are updated through a sequence of method invocations on e. Dashed boxes represent mutable components; solid boxes immutable (committed) components; double edged boxes canonical representatives. The *store* contains the stored, flyweighted states.

Fig. 7 illustrates these concepts. In the initial configuration (a), the state s wrapped by environment e shares a substructure with a stored, canonical state. When e receives a request to modify component x (b), it must create a new x node, but since the parent of that node (i.e., s) is not committed, that is the only new node that must be created. A similar process takes place upon processing the modification to y (c); the old y component (3), if not shared by any other object, will eventually be swept up by the garbage collector. A second modification to x (d) requires no new objects since x is mutable. The y component is then committed (e), and then is replaced by its canonic representative (f). After this, the entire state is committed (g) and replaced by its canonic representative (h).

5.4. Reductions

We now discuss several modifications to the standard DFS algorithm (Fig. 5) used by TASS to improve performance: partial order reduction, and heuristics guiding when to save a state, or push a state on the DFS stack.

Partial order reduction techniques determine a subspace of the full reachable state space that is guaranteed to contain a violation to some property if a violation is present in the full space [17]. Various schemes have been developed for different models and for preserving various classes of properties. TASS uses the *urgent* POR scheme [28], which targets models of MPI-based programs and preserves any property expressed as a predicate on potentially halted states, such as potential deadlock. In the reduced graph, the transitions enabled at a state will consist of either all enabled transitions in the full graph, or the transitions associated to one process. (For this purpose, a synchronous transition is considered to belong to the receiving process.) In the latter case, line 11 of Fig. 5 is modified to iterate over the proper subset of enabled transitions. The reduction in the number of states explored can be quite dramatic; for details, see [28]. In order to preserve potential deadlocks,

the reduced case excludes a process attempting to send a message which will be buffered (i.e., the destination process is not at a state from which it can receive the message synchronously). If potential deadlocks are not a concern, this restriction can be relaxed to obtain even further reduction. TASS provides options to either (a) check for all potential deadlocks, (b) check for only absolute deadlocks, or (c) ignore deadlock altogether, and adjusts the POR scheme appropriately.

The second optimization concerns when to save a state, i.e., canonicalize and mark it as "seen," corresponding to line 9 of Fig. 5. As we have seen, the saving of states is one of the primary costs, in both time and memory, of explicit-state model checking. Note that the safety of the algorithm does not depend on saving states: the procedure will not terminate unless every state (in the reduced space) has been explored. The danger is that states that are not saved may be explored more than once, or, in the worst case, the search will never terminate because there are cycles on which no state is saved. The goal is to find effective heuristics to determine when a state should be saved. For example, if a state has no chance of being seen again (i.e., it has only one incoming edge in the reduced state graph), a good heuristic will determine the state should not be saved.

The state-saving heuristic used by TASS uses knowledge of both the partial order reduction scheme and the structure of the original program (the AST) represented by the program graph. For example, a sequence of local (non-communication) statements without branches in a single process will essentially be treated as a single atomic transition by the POR scheme, so there is no need to save the intermediate states. Similar reasoning can be applied to communication statements in certain cases, but the analysis is more involved.

We first require a definition. We say a state s is *essentially nondeterministic* (END) if s has at least two outgoing edges in the reduced space, and the guards associated to those edges are not mutually exclusive. For example, if s has two outgoing transitions corresponding to the *true* and *false* branches from an `if` statement in one process, s is not END, since the guards will have the form ϕ and $\neg\phi$. Suppose a state s has two outgoing edges, and there are two paths in the reduced space departing from s along either edge, and these two paths meet at a common, non-vacuous state. Then it must be the case that s is an END state: since the path condition is only strengthened along a path, if the two guards had been mutually exclusive they could never have led to the common state.

Now suppose at some point during the DFS search through the reduced space, there is no END state on the stack. Let s be the state at the top of the stack, and s_0 the initial state at the bottom of the stack. The stack defines a path ρ from s_0 to s. We claim that any path from s_0 to s must have ρ as a prefix: if there were another such path ρ', then the state preceding the first divergence from ρ would be an END state on the stack, contradicting the assumption.

We can now define the TASS state-saving heuristic. Suppose execution of transition t in process p leads from a state s to a new state s'. Then s' will *not* be saved if and only if all of the following hold:

1. the reduced set of transitions T enabled in s' all lie in one process q; say q is at location l in s',
2. $p = q$ and the transitions of T are all local, or there is no END node on the stack, and
3. either there is only one incoming edge for l in the program graph for process q, or l is the join node ending an `if` block, or l is a loop location for a `for` loop.

The conditions above make it highly unlikely for s' to have a second incoming edge in the reduced state space. A possible exception arises if l is a `for` loop location and after some number of iterations of this loop, the process state is revisited. In such a case it would be more common to use a `while` loop. The most common situations requiring state-saving are wildcard receives, which lead to violations of the first two conditions, and `while` loops, which violate the third condition. In many programs that avoid these constructs, the state space is a tree, and there is no need to save any states.

The third and final optimization involves pushing states onto the DFS stack, corresponding to the recursive call of line 15 in Fig. 5. TASS will only push a state onto the stack if that state has at least two enabled transitions in the reduced space. Otherwise, it simply executes the next transition

without committing the state, keeping the state component mutable for as long as possible. (This also saves the time required to do a large number of unnecessary pops of the stack.) When a state with more than one enabled transition is finally reached, it is first committed, then pushed. It is not even necessary to save such a state, though there seems to be some advantage to simplifying them (which requires saving, as simplification results are always cached), so by default TASS does so.

6. Experimental Results

In this section we present the results of experiments applying TASS to seven program pairs from the FEVS verification suite [35]. While these programs are certainly simpler that those used in state-of-the-art scientific practice, they do capture many of the patterns common in real applications. The experiments are designed to shed light on questions such as *how far can the problems be scaled in a "reasonable" amount of time?*, *how does performance vary when scaling different parameters?*, *what is the relative cost of checking potential vs. absolute (or no) deadlock?*, *what is the relative cost of checking collective assertions?*, *how does the growth of states compare to time?*, *how do the number of states saved compared to the number seen?*, *how often is the theorem prover invoked?*, and *how much memory is used in general?*.

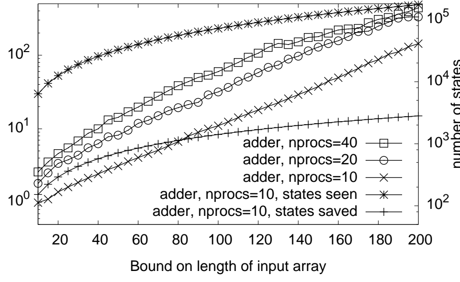
The programs were selected for several reasons. First, they represent various coding patterns. Second, the parallel programs use only blocking MPI calls. (Non-blocking MPI functions are not yet supported by TASS.) Finally, they are easily modified to be compatible with TASS. The modifications made to the programs were minor, and primarily involved the addition of TASS input/output pragmas, since TASS does not yet support reading from files or printing to files or the screen.

The programs are: (1) **adder**, with sequential and parallel versions described in §1.2, (2) **factorial**, two sequential programs to compute $n!$, one using iteration, the other recursion, (3) **diffusion**, a 1-dimensional diffusion equation solver in sequential and parallel (block-distributed) versions; (4) **manager-worker**, a program to multiply an $N \times L$ and an $L \times M$ matrix, in a standard sequential version and a parallel version using the manager-worker pattern, where the number of tasks is N; (5) **tiling**, a program to multiply two $N \times N$ matrices, in a standard sequential version and another sequential version that tiles the three loops; (6) **laplace**, a 2d-Laplace equation solver which iterates until convergence, in a sequential version and parallel row-distributed version; and (7) **integrate**, a numerical integrator using the trapezoid rule with a 1-dimensional adaptive mesh refinement, in a simple sequential version and a parallel version which distributes tasks using a manager-worker pattern.

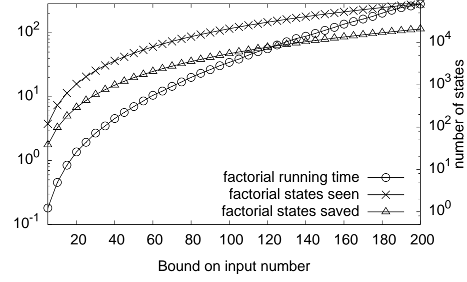
TASS was applied to these program pairs to verify functional equivalence and the other standard properties; for the most part, the data we present deals with functional equivalence verification. Each experiment focuses on scaling one parameter. All were run on a 2.8GHz quad-core Intel i7 iMac with 16GB RAM.

Figs. 8 and 9 give the results of some of these experiments, while Fig. 10 gives detailed statistics on some of the largest configurations of each program. Some experiments (e.g., **adder** and **factorial**) were scaled until we felt a reasonable input space had been explored. Others were scaled until time became prohibitive. For some programs, multiple experiments are performed scaling different parameters. Some observations about the results follow.

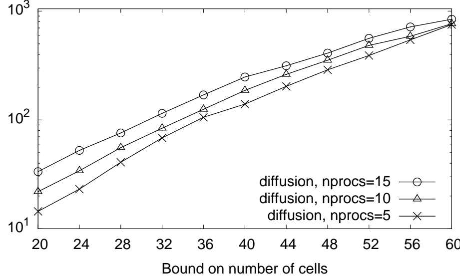
In general, how well the verification scales is dependent on the level of nondeterminism present in the programs. All of the programs have some nondeterminism from the input sizes, which are generally given as a bound rather than a fixed value. Programs where these are the only source of nondeterminism (e.g., **adder**, **factorial**, **diffusion**, **laplace**, and **tiling**) scale very well. In many cases, for fixed input size bound there is little change when varying the number of processes, much as one would expect from executing the programs on a single CPU.



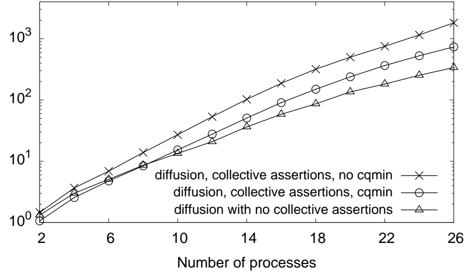
(a) Verification of functional equivalence for **adder**: sequential vs. parallel versions.



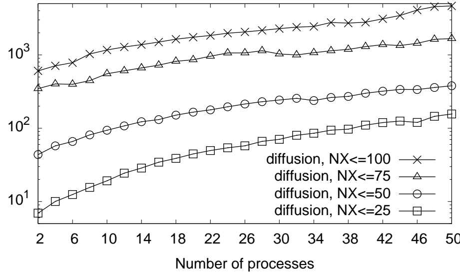
(b) Verification of functional equivalence for **factorial**: recursive vs. iterative versions.



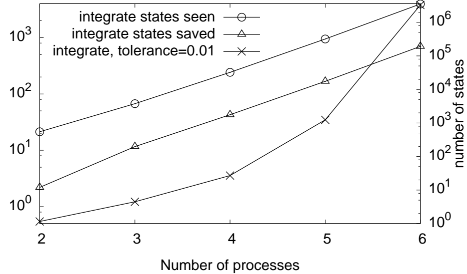
(c) Verifying parallel version of **diffusion** to be free of potential deadlocks.



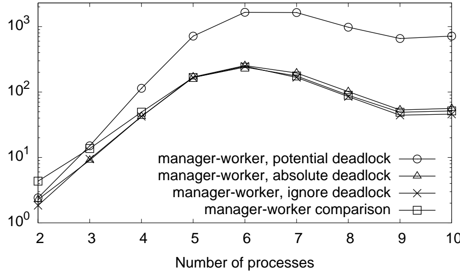
(d) Verification of functional equivalence for **diffusion** using collective assertions to check the correctness of the ghost cell exchange. NX≤3*nprocs.



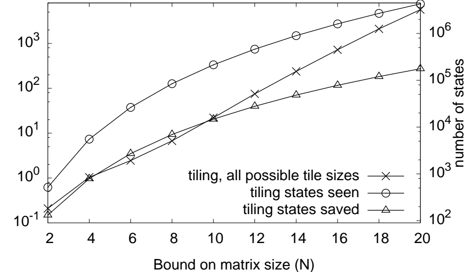
(e) Verification of functional equivalence for **diffusion** with bounded input array size and scaling nprocs.



(f) Verification of functional equivalence for **integrate**. The number of intervals is twice the number of processes.

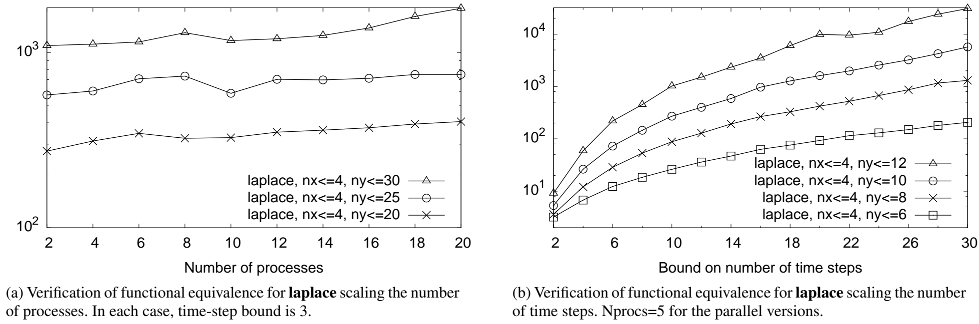


(g) Verification of functional equivalence and deadlock freedom for **manager-worker**. For each experiment, L≤2, M≤ 2, N≤ 8.



(h) Verification of functional equivalence for **tiling**.

FIGURE 8. TASS performance verifying programs from the FEVS suite. In all graphs, left y-axis is total verification time in seconds, on log-scale.



(a) Verification of functional equivalence for **laplace** scaling the number of processes. In each case, time-step bound is 3.

(b) Verification of functional equivalence for **laplace** scaling the number of time steps. Nprocs=5 for the parallel versions.

FIGURE 9. TASS performance, cont. Y-axis is total verification time (sec.), log scale.

On the other hand, programs that introduce additional nondeterminism (e.g. **manager-worker**, **integrate**) scale poorly, though TASS is still able to reach quite non-trivial configurations, especially in the first case. Both of these programs contain MPI receive calls using `MPI_ANY_SOURCE`. Each time one of these receives is reached, TASS must nondeterministically explore all possible source processes, resulting in an exponential blowup of the number of states, though the urgent POR scheme does mitigate the explosion somewhat [28]. The integration program has yet another layer of nondeterminism resulting from the adaptive mesh. At each step, the program must choose whether or not to subdivide each interval, and TASS must check both possibilities.

Fig. 10 and Fig. 8(a,b,f,h) present some evidence that the TASS state-saving heuristic is effective. In all experiments thus far, the number of states saved is significantly smaller than the number of states seen. For most large configurations, the number of states seen is at least 10 times the number saved. For many of these cases, we have checked that the number of states seen does not drop when all states are saved, indicating that there would be no advantage in saving the states the heuristic chooses to not save.

The state-saving heuristic and the morphic pattern contribute to TASS's efficient memory use. No experiment used more than 2 GB of memory.

Fig. 8(g) compares the running times for different deadlock detection strategies. Checking for potential deadlocks incurs a substantial performance penalty, due to the stricter POR strategy (see §5.4). However, checking for absolute deadlocks is very efficient; the amount of time required beyond ignoring deadlocks is negligible.

Fig. 8(d) compares running times for verifying functional equivalence of **diffusion** with and without collective assertions. The collective assertions are verified with and without *cqmin*, a collective queue minimization heuristic [37]. Checking collective assertions takes additional time, but the use of the minimization heuristic greatly reduces the penalty.

Even on the large configurations in Fig. 10, there are very few calls to CVC3. This is due in large part to TASS's simplification abilities. As discussed in §4.5, without this simplification we see many more calls to CVC3.

Fig. 8(g) is interesting because the running times first rise with the number of processes, and then decrease. In this experiment, the input bounds are fixed while the number of processes is scaled. As the number of processes increases, a greater fraction of the work is distributed in the initial, deterministic phase of the manager-worker algorithm. Thus with more processes, nondeterminism, and therefore verification time, decrease.

In Fig. 8(a,h) we see running times that appear to grow exponentially, but numbers of states seen and saved that are clearly sub-exponential (in fact, in the case of (a), polynomial). The reasons for this discrepancy are not fully understood at this time, and will require further investigation.

program	bounds	nprocs	time (s)	seen	saved	values	proofs	mem
adder	$n \leq 200$	30	443	411044	6832	12784	401	221
factorial	$n \leq 200$	1	282	82410	20903	2008	802	1900
diffusion	$n_x \leq 100 \wedge n_t \leq 2$	50	4653	5788608	1186	39133	204	380
tiling	$n \leq 20$	1	3448	2692861	119933	183840	377	720
manager-worker	$l \leq 2 \wedge m \leq 2 \wedge n \leq 10$	7	13157	151752013	2167303	14650	72	1736
manager-worker	$l \leq 2 \wedge m \leq 2 \wedge n \leq 10$	11	242	10543295	90147	15515	72	137
integrate	$intervals = 12 \wedge tol = 0.01$	6	3788	3511123	191492	1468	25	305
laplace	$n_x \leq 4 \wedge n_y \leq 12 \wedge n_t \leq 30$	5	30911	3384271	20336	16626	1199	1947
laplace	$n_x \leq 4 \wedge n_y \leq 30 \wedge n_t \leq 3$	20	1787	875577	5021	18510	371	746

FIGURE 10. TASS performance, verifying functional equivalence on maximal configurations: nprocs is number of processes, seen is the total number of states seen, saved is the number of states saved, values is the total number of symbolic expressions saved, proofs is the number of calls to the theorem prover CVC3, mem is amount of memory used in MB.

7. Related Work

Symbolic execution was originally proposed in the context of program testing [7, 21]. Since then it has seen many extensions and generalizations, such as its combination with model checking [15,20]. *Comparative symbolic execution*, the method combining model checking with symbolic execution to verify the functional equivalence of two programs, was introduced in [32, 33].

The formal description of symbolic execution given in Section 4 uses ideas from *abstract interpretation* [9, 10]. To formulate our approach in the AI framework, the concrete domain consists of sets of concrete states; the abstract domain consists of sets of symbolic states. The concretization function maps a set S of symbolic states to $\cup_{\hat{s}\in S}\gamma(\hat{s})$. We have not defined an abstraction map α in the opposite direction, because it is not needed for our purposes (and not always for the general theory either [11]). Our consistency assumptions (3) and (4) are used to prove Theorem 5, an approach that corresponds roughly to the use of the "local hypothesis" [9, (6.5)] to establish the "global hypothesis" [9, (6.0)].

SPIN [19] is one of the most widely-used model checking tools, and introduced a large array of techniques to reduce the time and memory consumed by explicit state model checking. For example, SPIN's "collapse" compression algorithm allows global states to share common process states to reduce the memory footprint. However, SPIN's input language, Promela, lacks many of the challenging constructs found in most commonly-used programming languages, such as procedures and recursion, pointers, and dynamic memory management.

Bandera [8] and the related tool Bogor [25] innovated many methods in software model checking and were among the first tools to apply these techniques to Java programs. The design of TASS—in particular, its general approach to the separation of concerns—was heavily influenced by studying both tools. Bogor's "collapse" compression extended SPIN's technique by allowing states to share sub-structures at various levels of the state hierarchy [26]. The morphic pattern described in §5 has much in common with this approach. The main difference is that in Bogor, states are encoded (typically as bit vectors) before being saved and/or checked for being seen, to conserve memory. Also, instead of storing states on the DFS stack, Bogor stores transitions, and a method to invert a transition to obtain the previous state is used when popping the stack. (A similar technique is used by SPIN.) Essentially, this allows Bogor to maintain only one uncompressed, mutable state during the search. In contrast, TASS avoids the computational expense associated with compressing states and inverting transitions by maximizing opportunities for sharing among stored states—at every node in the state hierarchy, flyweighting is used to obtain a unique representative of the equivalence class for that node—and limiting the number of states saved and pushed onto the stack. We have presented

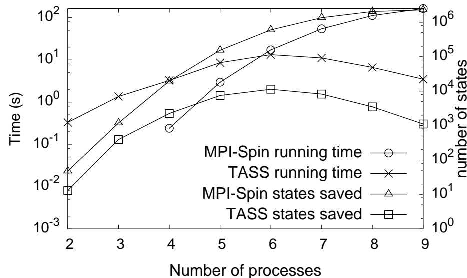


FIGURE 11. Verifying functional equivalence for **manager-worker** using MPI-SPIN and TASS. $L = M = 2, N = 8$. For 2 and 3 processes, MPI-SPIN times are under 0.01 sec. and are not reported by SPIN.

some evidence for the effectiveness of these engineering choices in the case of TASS: for all the scalability experiments conducted so far, memory has never exceeded 2GB.

Among the many tools for verifying properties of numerical programs, we mention only a few that are most relevant to TASS.

Comparative symbolic execution for MPI programs was used in the predecessor to TASS, MPI-SPIN [29–31,34], an extension to SPIN. Unlike TASS, MPI-SPIN requires the manual translation of the program into Promela. It also does not support the MPI-specific partial order reduction, the morphic pattern, or the sophisticated symbolic reasoning techniques used by TASS. For these reasons, MPI-SPIN does not in general scale as well as TASS. Fig. 11 illustrates this fact on the manager-worker matrix multiplication example; other examples reveal an even greater discrepancy.

ASTRÉE is an abstract interpretation-based static analyzer for a subset of C [12]. It can reason precisely about floating-point and limited-precision integer arithmetic and verify absence of many runtime errors, but does not deal with dynamic memory allocation or recursion; its main applications have been to real-time embedded software. FLUCTUAT [14] is another AI-based static analyzer providing information about rounding errors in C programs.

KLEE [5] 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 example, it does not deal with parallel programs, and it uses "bit-precise" reasoning instead of mathematical real arithmetic. As discussed in §1, this is not as useful for equivalence checking of numerical programs since they are rarely expected to be bit-equivalent. TVOC [3] is a tool for checking the correctness of compiler optimizations for sequential programs; it takes a functional equivalence verification approach based on a set of pre-defined transformation patterns. BLAST [18] is a C program verification system to check safety properties based on predicate abstraction, but has not targeted numerical or parallel software, or the problem of functional equivalence. The Why/Krakatoa/Caduceus [16] and Frama-C [1] frameworks provide a set of tools for checking Java and C programs. These use special comments or JML-style annotations to specify numerical accuracy requirements. None of these tools applies to message-passing based parallel programs or the problem of functional equivalence. ISP [38], 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.

8. Conclusions and Future Work

We have presented some evidence that formal verification techniques can be effectively applied to numerical software, including MPI-based parallel programs. Questions that often plague developers,

such as *can the code deadlock?* and *is this program functionally equivalent to that one?*, can be decided by TASS on significant portions of the program's state space. A variety of new symbolic and model-checking techniques have been incorporated into TASS, enabling it to scale to non-trivial configurations of these programs.

Our future work on TASS includes several major improvements. First, we plan to use a mature compiler infrastructure, such as ROSE or LLVM, to construct a more robust model extractor for C/C++ and Fortran source code. We will deal with some C constructs that we have so far ignored, such as the bit-wise operators, and expand the library functions modeled. Precise modeling of I/O is another challenge: real programs of course read and write files, rather than the artificial global variables used in TASS; a symbolic representation of these files will therefore be required. We must also expand the subset of MPI which TASS can handle, beginning with the widely-used nonblocking operations. None of these problems is insurmountable, and in fact all have been dealt with in various contexts before.

We hope the TASS framework will be used by other researchers, as well as ourselves, to explore new verification techniques for numerical software. We have already used TASS to investigate a new kind of assertion for message-passing programs [37]. We are currently extending that approach to deal with "collective loop invariants." These enable TASS to verify properties without bounds on many parameters. Other promising avenues include using TASS to reason about performance of programs, rather than just correctness, and using TASS to automatically transform programs to improve their performance.

References

[1] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 127–141. Springer Berlin / Heidelberg, 2010.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[3] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *LNCS*, pages 291–295. Springer-Verlag, 2005.

[4] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2007.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[6] Franck Cappello, Thomas Hérault, and Jack Dongarra, editors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007, Proceedings*, volume 4757 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2:215–222, May 1976.

[8] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE '00: Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, USA, 2000. ACM.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[10] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[11] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *The Journal of Logic and Computation*, 2(4):511–547, 1992.

[12] Patrick Cousot, Rashia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyser. In Mooly Sagiv, editor, *Proc. European Symposium on Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 21–30, Edinburgh, April 2–10 2005. Springer.

[13] Werner Damm and Holger Hermanns, editors. *CAV 2007*, volume 4590 of *LNCS*. Springer, 2007.

[14] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer Berlin / Heidelberg, 2009.

[15] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*, pages 157–166. IEEE Computer Society, 2006.

[16] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm and Hermanns [13], pages 173–177.

[17] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *SPIN 2003*, volume 2648, pages 235–239, 2003.

[19] Gerard J. Holzmann. *The* SPIN *Model Checker*. Addison-Wesley, Boston, 2004.

[20] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[21] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[22] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, version 2.2, September 4, 2009. `http://www.mpi-forum.org/docs/`, 2009.

[23] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, 5:128–138, March 1979.

[24] Terence Parr. ANTLR Parser Generator web page. `http://www.antlr.org`.

[25] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, 2003.

[26] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic software. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

[27] S. F. Siegel et al. The Toolkit for Accurate Scientific Software web page. `http://vsl.cis.udel.edu/tass`, 2010.

[28] Stephen F. Siegel. Efficient verification of halting properties for MPI programs with wildcard receives. In Radhia Cousot, editor, *VMCAI 2005*, volume 3385 of *LNCS*, pages 413–429, 2005.

[29] Stephen F. Siegel. Model checking nonblocking MPI programs. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14–16, 2007, Proceedings*, volume 4349 of *LNCS*, pages 44–58, 2007.

[30] Stephen F. Siegel. Verifying parallel programs with MPI-SPIN. In Cappello et al. [6], pages 13–14.

[31] Stephen F. Siegel and George S. Avrunin. Verification of halting properties for MPI programs using non-blocking operations. In Cappello et al. [6], pages 326–334.

[32] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In Lori L. Pollock and Mauro Pezzé, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17–20, 2006*, pages 157–168. ACM, 2006.

[33] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin, and Lori A. Clarke. Combining symbolic execution with model checking to verify parallel numerical programs. *ACM Transactions on Software Engineering and Methodology*, 17(2):Article 10, 1–34, 2008.

[34] Stephen F. Siegel and Louis F. Rossi. Analyzing BlobFlow: A case study using model checking to verify parallel scientific software. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 15th European PVM/MPI User's Group Meeting, Proceedings*, volume 5205 of *LNCS*. Springer, 2008.

[35] Stephen F. Siegel and Timothy K. Zirkel. A Functional Equivalence Verification Suite. `http://vsl.cis.udel.edu/fevs`.

[36] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of MPI-based parallel programs (poster). In *The 16th ACM SIGPLAN Annual Symposium on Principles and Practices of Parallel Programming (PPoPP)*. ACM, 2011. To appear.

[37] Stephen F. Siegel and Timothy K. Zirkel. Collective assertions. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation: 12th International Conference, VMCAI 2011*, volume 6538 of *LNCS*, pages 387–402, 2011. To appear.

[38] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 285–286, 2008.

Appendix A. Proofs of Theorems

Proof of Lemma 2. Let $l_0 \in \mathsf{Loc}_0$. Then

$$\gamma(\langle \phi_0, l_0, \xi_0\rangle) = \{\langle l_0, \mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\xi_0\rangle \mid \theta \in \mathsf{Func}(\mathcal{X},\mathsf{Val}) \wedge \mathsf{eval}_{\mathcal{X}}(\phi_0,\theta)\}.$$

Now for any $\theta \in \mathsf{Func}(\mathcal{X},\mathsf{Val})$ for which $\mathsf{eval}_{\mathcal{X}}(\phi_0,\theta) = \mathit{true}$, (3) and the definition of ϕ_0 imply

$$\mathsf{eval}_V(g_0, \mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\xi_0) = \mathsf{eval}_{\mathcal{X}}(\mathsf{seval}(g_0,\xi_0),\theta) = \mathsf{eval}_{\mathcal{X}}(\phi_0,\theta) = \mathit{true}.$$

By definition of State_0, this means $\langle l_0, \mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\xi_0\rangle \in \mathsf{State}_0$.

Conversely, given any $\langle l_0, \eta\rangle \in \mathsf{State}_0$, we have $\mathsf{eval}_V(g_0,\eta) = \mathit{true}$. Let $\theta\colon \mathcal{X} \to \mathsf{Val}$ be any map such that $\theta(X_v) = \eta(v)$ for all $v \in V$. Hence for any $v \in V$,

$$(\mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\xi_0)(v) = \mathsf{eval}_{\mathcal{X}}(\xi_0(v),\theta) = \mathsf{eval}_{\mathcal{X}}(X_v,\theta) = \theta(X_v) = \eta(v).$$

That is, $\mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\xi_0 = \eta$, and so

$$\mathsf{eval}_{\mathcal{X}}(\phi_0,\theta) = \mathsf{eval}_V(g_0, \mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\xi_0) = \mathsf{eval}_V(g_0,\eta) = \mathit{true}.$$

It follows that $\langle l_0,\eta\rangle \in \gamma(\langle \phi, l_0, \xi_0\rangle)$. □

Proof of Lemma 3. Write $\hat{s} = \langle \phi, l, \xi\rangle$. We may assume t has the form $\langle l, g, \alpha, l'\rangle$ (else both sides are empty). By (5), $\mathsf{snext}(\hat{s}, t) = \{\hat{s}'\}$, where $\hat{s}' = \langle \phi \wedge \mathsf{seval}(g,\xi), l', \mathsf{seffect}(\alpha,\xi)\rangle$. Applying (6), one has

$$\gamma(\hat{s}') = \{\langle l', \mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\mathsf{seffect}(\alpha,\xi)\rangle \mid \theta \in \mathsf{Func}(\mathcal{X},\mathsf{Val}) \wedge \mathsf{eval}_{\mathcal{X}}(\phi \wedge \mathsf{seval}(g,\xi),\theta)\}. \quad (11)$$

By (4), $\mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \mathsf{seffect}(\alpha,\xi) = \mathsf{effect}(\alpha, \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi)$. By (2) and (3),

$$\begin{aligned} \mathsf{eval}_{\mathcal{X}}(\phi \wedge \mathsf{seval}(g,\xi),\theta) &= \mathsf{eval}_{\mathcal{X}}(\phi,\theta) \wedge \mathsf{eval}_{\mathcal{X}}(\mathsf{seval}(g,\xi),\theta) \\ &= \mathsf{eval}_{\mathcal{X}}(\phi,\theta) \wedge \mathsf{eval}_{V}(g, \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi). \end{aligned}$$

Combining these with (11) yields

$$\gamma(\hat{s}') = \bigcup_{\substack{\theta \in \mathsf{Func}(\mathcal{X},\mathsf{Val}) \\ \mathsf{eval}_{\mathcal{X}}(\phi,\theta) \wedge \mathsf{eval}_V(g, \mathsf{eval}_{\mathcal{X}}(-,\theta)\circ\xi)}} \{\langle l', \mathsf{effect}(\alpha, \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi)\rangle\}.$$

Applying the definitions of next and of γ (equations (1) and (6)) to the above yields

$$\gamma(\hat{s}') = \bigcup_{\substack{\theta \in \mathsf{Func}(\mathcal{X},\mathsf{Val}) \\ \mathsf{eval}_{\mathcal{X}}(\phi,\theta)}} \mathsf{next}(\langle l, \mathsf{eval}_{\mathcal{X}}(-,\theta) \circ \xi\rangle, t) = \bigcup_{s \in \gamma(\hat{s})} \mathsf{next}(s,t).$$

□

Proof of Theorem 5. Let $S = \bigcup_{\hat{s} \in \mathsf{SReach}} \gamma(\hat{s})$. We first show $S \subseteq \mathsf{Reach}$. To do this we will show (a) for any $\hat{s}_0 \in \mathsf{SState}_0$, $\gamma(\hat{s}_0) \subseteq \mathsf{Reach}$, and (b) if $\hat{s} \in \mathsf{SState}$, $\gamma(\hat{s}) \subseteq \mathsf{Reach}$, and $\hat{s}' \in \mathsf{SNext}(\hat{s})$, then $\gamma(\hat{s}') \subseteq \mathsf{Reach}$. But Lemma 2 implies $\gamma(\hat{s}_0) \in \mathsf{State}_0 \subseteq \mathsf{Reach}$, whence (a). Too see (b), note Lemma 4 implies $\gamma(\hat{s}') \subseteq \bigcup_{s \in \gamma(\hat{s})} \mathsf{Next}(s)$. But for any $s \in \gamma(\hat{s}) \subseteq \mathsf{Reach}$, $\mathsf{Next}(s) \subseteq \mathsf{Reach}$.

To show $\mathsf{Reach} \subseteq S$, we show (a) for any $s_0 \in \mathsf{State}_0$, $s_0 \in S$, and (b) if $s \in S$ and $s' \in \mathsf{Next}(s)$ then $s' \in S$. By Lemma 2, there is some $\hat{s}_0 \in \mathsf{SState}_0 \subseteq \mathsf{SReach}$ such that $s_0 \in \gamma(\hat{s}_0)$, whence (a). As for (b), since $s \in S$, $s \in \gamma(\hat{s})$ for some $\hat{s} \in \mathsf{SReach}$. By Lemma 4, $s' \in \gamma(\hat{s}')$, for some $\hat{s}' \in \mathsf{SNext}(\hat{s}) \subseteq \mathsf{SReach}$. Hence $s' \in S$. □

Proof of Theorem 7. Let $S = \bigcup_{\hat{s} \in \mathsf{SReach}} \gamma(\hat{s})$. By Theorem 5, $S = \mathsf{Reach}$.

We first show the right hand side is contained in the left. If $\hat{s} \in \mathsf{SReach}$ then there exist $\hat{s}_0, \ldots, \hat{s}_n \in \mathsf{SState}$ such that $\hat{s}_0 \in \mathsf{SState}_0$, $\hat{s}_n = \hat{s}$, and $\hat{s}_i \in \mathsf{Next}(\hat{s}_{i-1})$. Taking $\sigma_i = [\hat{s}_i]$ shows $\sigma_n \in \mathsf{QReach}$ and $\gamma(\hat{s}) = \bar{\gamma}(\sigma_n)$.

To see that the left hand side is contained in S, we show (a) given any $\sigma_0 \in \mathsf{QReach}_0$, $\bar{\gamma}(\sigma_0) \in S$, and (b) if $\sigma \in \mathsf{QState}$, $\sigma' \in \mathsf{QNext}(\sigma)$, and $\bar{\gamma}(\sigma) \subseteq S$, then $\bar{\gamma}(\sigma') \subseteq S$. To see (a), say $\sigma_0 = [\hat{s}_0]$, where $\hat{s}_0 \in \mathsf{SState}_0$. Then $\bar{\gamma}(\sigma_0) = \gamma(\hat{s}_0) \in S$, since $\mathsf{SState}_0 \subseteq \mathsf{SReach}$.

For (b), by definition of QNext there exist $\hat{s} \in \sigma$ and $\hat{s}' \in \sigma'$ such that $\hat{s}' \in \mathsf{SNext}(\hat{s})$. By assumption, $\gamma(\hat{s}) \subseteq S = \mathsf{Reach}$. By Lemma 4, $\gamma(\hat{s}') \subseteq \bigcup_{s \in \gamma(\hat{s})} \mathsf{next}(s)$. But if $s \in \gamma(\hat{s}) \subseteq \mathsf{Reach}$, then $\mathsf{Next}(s) \subseteq \mathsf{Reach}$, so $\bar{\gamma}(\sigma') = \gamma(\hat{s}') \subseteq \mathsf{Reach}$, as required. □

Stephen F. Siegel
e-mail: siegel@cis.udel.edu
Verified Software Laboratory, Department of Computer & Information Sciences, University of Delaware, Newark DE 19716, USA

Timothy K. Zirkel
e-mail: zirkeltk@udel.edu
Verified Software Laboratory, Department of Computer & Information Sciences, University of Delaware, Newark DE 19716, USA