Automated Verification of Chapel Programs Using Model Checking and Symbolic Execution

Timothy K. Zirkel, Stephen F. Siegel, and Timothy McClory

Verified Software Laboratory, Department of Computer and Information Sciences
University of Delaware, Newark DE 19716, USA
{zirkeltk|siegel|tmcclory}@udel.edu

Abstract. Chapel is a new programming language targeting high performance computing. Chapel makes it easier to write parallel code, but is still subject to concurrency problems such as deadlocks, race conditions, and nondeterministic results. In theory, model checking and symbolic execution tools can help with these problems, but certain Chapel primitives are difficult to represent in the models used by existing tools. For example, some primitives dynamically create arbitrarily nested scopes with threads executing within those scopes. We present (1) a new formal model that naturally represents these dynamic concepts and (2) a new prototype model checking/symbolic execution tool for Chapel programs that uses this model as its intermediate representation. We describe how the tool translates Chapel into this IR and the results of applying the tool to several synthetic Chapel programs.

1 Introduction

Currently, most high performance scientific programs are written in C, C++, or Fortran, in combination with one or more concurrency extensions, such as the Message Passing Interface library [13] or OpenMP [14]. These approaches, based on old programming languages and concurrency models, pose well-known challenges to programmer productivity and to writing correct, efficient code. Much research effort has focused on ameliorating these problems with better debugging and analysis tools, but there have also been a number of recent initiatives introducing entirely new programming languages for HPC (e.g., [1, 3, 5, 18]).

These new languages include Chapel [5], a programming language designed for high productivity parallel programming. It incorporates high level constructs for expressing common parallel programming patterns. While these constructs simplify programs, they also make it easier to introduce unintended forms of nondeterminism. Such nondeterminism can lead to deadlock or unexpected results. We present a prototype verification tool for programs written in the subset of Chapel shown in Figure 1. The tool can automatically detect such defects or show that none exist within specified parameters.

The Chapel programming runtime spawns *tasks* (threads) that interact via shared memory. The `cobegin` statement creates a new task for each statement in a block. The `coforall` statement creates a new task for each iteration of a

```
prog ::= decl+
 decl ::= (config | extern)? (const | var | param) v : type (= expr)? ;
        | (iter | proc) f ( (v : type (, v : type)* )? ) (: type)? block
block ::= { (decl | stmt) * }
 type ::= sync? (real | int | void | bool | string | [ expr ] type)
 stmt ::= block | call | expr = expr ; | return expr? ; | yield expr ;
        | while ( expr ) block | if ( expr ) then stmt (else stmt)?
        | cobegin block
        | (for | forall | coforall) v in (expr .. expr | call) block
 expr ::= x | int_literal | float_literal | string_literal | true | false
        | call | expr [ expr ] | ( expr ) | (+ | - | !) expr
        | expr (|| | && | == | != | < | > | >= | <= | + | - | * | / | %) expr
 call ::= f ( (expr (, expr)*)? )
```

Fig. 1. Abstract syntax of Chapel subset.

loop. The `forall` statement partitions the iterations of the loop among one or more tasks. The number of tasks created depends on the runtime configuration. Execution is halted after `cobegin`, `coforall`, and `forall` statements until all created tasks have completed. An example of a `forall` statement is included in Figure 2.

The iteration domain of a loop is specified by a special type of function called an *iterator*. Ordinary functions that yield one return value are called *procedures*. Iterators are defined using the same syntax as a procedure except `return` statements are replaced with `yield` statements. An iterator definition is preceded by the keyword `iter` and a procedure definition is preceded by the keyword `proc`. Iterators are used to generate a sequence of values.

Constant variable declarations begin with the keyword `const` and must contain an initialization expression. Regular variables are declared with keyword `var`. Each variable is permitted to have one of two modifiers: `config` or `extern`. `Config` variables must be in the global scope. These variables may be initialized at compile time through some platform-dependent means. `Extern` variables are assumed to be defined outside of the program. No memory is allocated for `extern` variables and they are not initialized.

Tasks can be coordinated with `sync` variables. A `sync` variable is declared using the `sync` keyword. Each `sync` variable has a boolean flag associated with it, which indicates whether the variable is *empty* or *full*. A `sync` variable can only be read when full and can only be written to when empty. A write to a `sync` variable makes it full and a read from the variable makes it empty. A task that attempts to read or write a `sync` variable must wait until it is in the correct state. The flag is initially set to be empty.

The program in Figure 2 demonstrates some of the challenges for Chapel verification. The dynamic state illustrates how the tasks share certain memory regions. The tasks created have their own scopes for the variables that are declared in the body of the `forall` loop. However, they share all the variables

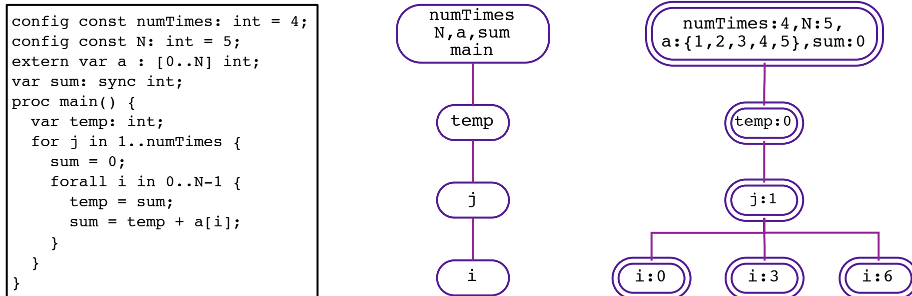


Fig. 2. `adderPar.chpl`. Left: This program computes the sum of the elements in `a` and stores the result in `sum`. This process is repeated `numTimes` times. If `sum` is not declared to be a `sync` variable then the program is incorrect; center: static scope tree for this program; right: the dynamic scope tree in a state.

above that point in the static scope tree. Hence the dynamic scope tree shown on the right. Standard model checking languages, such as Promela [10], provide a scope tree of height 1 (a global scope and a local scope for each process). Faithful representation of Chapel programs requires an arbitrary scope tree.

State space explosion is always an issue with model checking, but the dynamic nature of Chapel concurrency exacerbates the problems. For example, to execute a nested pair of `coforall` statements, each with an iteration space of size N, N^2 concurrent processes are instantiated.

Our Chapel Verification Tool (CVT) uses model checking with symbolic execution [4, 11, 12] to verify correctness properties of Chapel programs. It can also verify the functional equivalence of two programs, using comparative symbolic execution [16]. We constructed CVT by re-using certain components of the Toolkit for Accurate Scientific Software (TASS) [17]. TASS supports a subset of MPI, but not dynamic process creation, and the only scope shared by TASS processes is the one global ("shared") scope. Verifying Chapel programs requires a substantially different state representation.

2 CIR: The Chapel Intermediate Representation

In this section we decribe the Chapel Intermediate Representation (CIR). CIR is a "guarded command" style representation [7] that provides simple primitives for dynamic process creation, procedure calls, nondeterminism, and message-passing. It also adds to the usual model a notion of scopes, which have both a static and a dynamic aspect.

Spin [10] uses a similar model, and supports dynamic process creation, but lacks procedures and, as mentioned above, only supports a scope hierarchy of height one. The dSPIN [6] extension of Spin added procedures, arbitrary scope nests within a procedure, and other dynamic constructs, but only allowed function definitions in the top scope. CIR goes further by allowing procedure defi-

nitions in any scope. The “forking” of such procedures leads to states such as the one depicted in Figure 2(right). In that state, there are three processes, each with its own copy of `i`, but sharing `j` and variables in the scopes above. This general model of concurrency and scopes is similar to the threading model in some functional languages, such as Racket [8].

2.1 CIR models

A *CIR model* consists of the following components. First, there is a set Σ of (static) *scopes*, which has the structure of a rooted tree with root σ_0. These correspond to the lexical scopes in the source code, plus some additional scopes that may be added to translate complex statements (see Section 3). The root scope represents the outermost scope encompassing the entire program. If τ is a child of σ, the lexical scope represented by τ is immediately contained in that represented by σ.

The model associates to each $\sigma \in \Sigma$ a set of typed variables and a set of procedure symbols. We say these variables and procedure symbols are *declared in* σ. All of these sets are pairwise disjoint; in particular, the variables declared in σ do not include those declared in any child of σ. We say a variable or procedure symbol is *visible in* σ if it is declared in σ or an ancestor of σ.

Types include *boolean*, *real*, *int*, *string*, arrays of any element type, and a type *process* for process IDs. For simplicity, in this paper, the *real* and *int* types represent the mathematical real numbers and integers, though no fundamental changes are required in the model to incorporate finite-precision or other types.

For each procedure symbol f declared in the model, there is a *procedure scope*, which is a child of the scope in which f is declared. A scope can be the procedure scope of at most one procedure. The *root procedure* has σ_0 as its procedure scope, and is the only procedure that does not have a declaration scope.

Every scope σ “belongs to” a unique procedure: if σ is the procedure scope for some f then σ belongs to f, else σ belongs to the procedure to which the parent of σ belongs.

The model associates to each procedure symbol f a *procedure signature*, which consists of a return type (possibly “void”) and a sequence of parameter types. Finally, there is a *guarded transition system* associated to f. This system includes a set of locations, including a start location. Each location l has an associated scope $\mathsf{lscope}(l)$ which must belong to f; there is no other restriction on $\mathsf{lscope}(l)$. The location also has some number (possibly 0) of outgoing transitions. Each transition comprises (1) a *guard*, a boolean-valued expression specifying when the transition is enabled, (2) a destination location, and (3) an *atomic CIR statement*.

The kinds of atomic statements are listed below. In the list, v, w and b denote left-hand side (LHS) expressions, $e, \mathsf{dest}, \mathsf{src}, \mathsf{tag}, e_1, \ldots$ denote expressions, f is a procedure symbol, and x is a variable. In all cases, the variables and procedure symbols must be visible in $\mathsf{lscope}(l)$. An asterisk indicates an optional element. The semantics of these statements is given in Section 2.2.

1. $v = e$
2. return e^*
3. $v^* = f(e_1, \ldots, e_n)$
4. $v^* =$ fork $f(e_1, \ldots, e_n)$, where v has *process* type
5. join(e), where e has *process* type
6. $v =$ choose(e), where v and e have integer type
7. send(dest, e, tag), where dest has *process* type, tag has integer type
8. receive(src, v, tag) (like above, but src and tag may have form any(w))
9. write(e)
10. noop
11. sync-read(b, v, x), where b has boolean type
12. sync-write(b, x, e), where b has boolean type.

2.2 CIR semantics

We assume given a set of (typed) *values*. The values of type *process* are integers. Given this, the *state* of a CIR model comprises

1. a set Δ of *dyscopes* (dynamic scopes) which has the structure of a rooted tree with with root δ_0;
2. a function static: $\Delta \to \Sigma$ which respects tree structure, i.e., if $\delta_1, \delta_2 \in \Delta$ and δ_1 is a child of δ_2 then static(δ_1) is a child of static(δ_2). We say δ *is an instance of* static(δ);
3. for each $\delta \in \Delta$, a function which assigns a value (of the correct type) to each variable declared in static(δ);
4. a set of integers P (the *process IDs* used in the state);
5. a function which assigns to each $p \in P$ a *call stack*, which is a sequence of frames, each frame consisting of a location l in some procedure and a dyscope δ such that static$(\delta) =$ lscope(l);
6. a function which assigns to each ordered pair of processes a finite sequence of *messages*, where a message consists of a value for the message data and an integer tag.

Figure 3(right) shows a state of the model to its left. This state has two dyscopes which are instances of scope 1, no instance of scope 6, and 1 instance of each of the remaining scopes. There are 3 processes, whose call stacks are illustrated. (The locations are not shown.)

The semantics of a CIR model are specified in a small-step operational style using an interleaving view of concurrency. For a transition to be enabled, its guard must evaluate to *true*. In addition, certain statements have an *implicit guard* which must also hold; these are discussed below. For the most part, all of this is standard, so we limit the discussion to aspects of the semantics that may be particular to CIR.

To execute $v =$ choose(e), e is evaluated to yield an integer n. An integer in the range $[0, n-1]$ is chosen nondeterministically and assigned to v.

A send statement specifies the destination process, the data to be sent, and an integer tag. Tags are used by the receiver to select messages for reception.

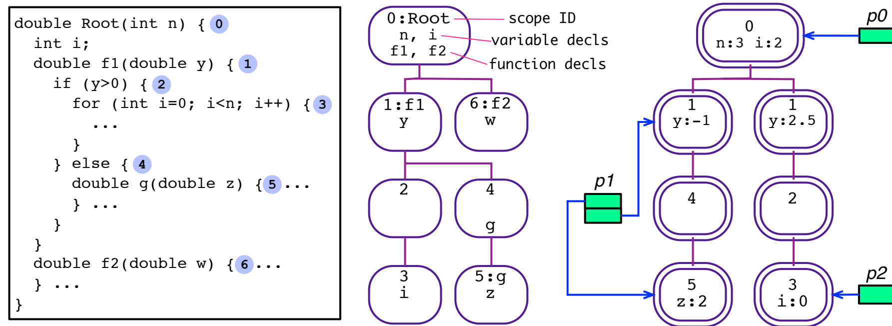


Fig. 3. CIR scopes. left: pseudocode representation of a CIR model with lexical scopes numbered; center: the static scope tree; right: a state consisting of 3 processes and 7 dynamic scopes.

The data may have any type, including an array type. The execution of the send creates a message and appends it to the message sequence for (p, q), where p is the process ID of the sender and q that of the receiver. The receive has an implicit guard which holds when a message matching the `tag` is available in an appropriate queue and pulls out the oldest message matching the `tag`. The "any" variant used as the source argument means the receive will nondeterministically choose one of the incoming queues which has a matching message, and then pull out the oldest matching message from that queue; the process ID of the sender will be stored in v. The use of "any" as the tag argument simply means the oldest message will be removed from the queue, regardless of its tag; the tag of that message will be stored in tag.

The sync-read and sync-write are used to model accesses to a `sync` variable x. The sync-read has implicit guard b and when executed it performs the assignment of x to v and sets b to *false* in one atomic step. A sync-write behaves dually.

When control moves from one location to another within a procedure's transition system, the scope may change. When this happens, new dyscopes are created and added to the state. This is carried out in such a way that the correspondence between dynamic and static scopes is preserved. The protocol requires computing the "join" in the static scope tree of the old and new scopes, considering the path from the old scope to the join to the new scope, and then creating a corresponding structure in the dynamic tree; see Figure 4(a–c).

A dyscope is *unreachable* if it does not occur in any frame and is not an ancestor of a dyscope occurring in a frame. Such a dyscope may be removed from the state; see Figure 4(d).

If a call or fork is executed in dyscope δ then since f is visible, it must be the case that f is declared in static(δ') where δ' is δ or an ancestor of δ. A new dyscope is created whose parent is δ' and whose scope is the procedure scope of f. A new frame is created referring to the new dyscope. For a call, the frame is

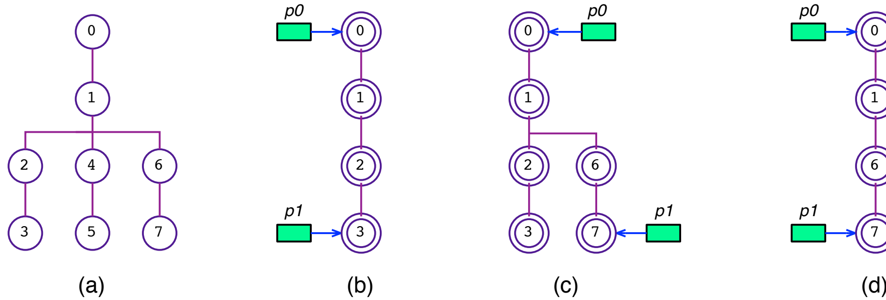


Fig. 4. Jump protocol. (a) a static scope tree; (b) a dyscope tree; p_1 is about to move from a location in scope 3 to a location in scope 7; (c) new dyscopes are added corresponding to the path from scope 1 (the join of 3 and 7) to 7; (d) dyscopes 2 and 3 became unreachable so were removed.

pushed onto the existing stack; for a fork, a new process is created whose stack consists of the single frame.

If a process has terminated and there are no references to that process in the state, it can be removed from the state. At any point, process IDs can also be re-assigned throughout the state, for example, to remove gaps or put the state into a canonical form.

Symbolic semantics. We have described the "concrete" semantics of a CIR model. Minimal changes are required to apply symbolic execution to CIR. These techniques are now well-known: values associated to variables become (typed) symbolic expressions, a path condition variable is added to the state, any time a guard *may* be *true* a transition is enabled in the symbolic space, and so on. Currently, our symbolic representation uses concrete representations of the dynamic scope tree, call stacks, and message queues (but the data in a message is symbolic), although it is conceivable that these structures could also be represented symbolically, thereby enabling reasoning for unbounded numbers of processes, dyscopes, and so on.

3 Translating from Chapel to CIR

In this section we present the translation of several Chapel structures into CIR. The translations are provided in a pseudocode for CIR. For readability, the pseudocode utilizes common sequential constructs (e.g., `for`, `if`) that do not exist in CIR, but are translated to the transition system in standard ways.

3.1 Built-in constants

CVT uses four built-in constants. These do not appear in the Chapel code being verified, but are used in the translation of several constructs. Three of these constants are used as message tags:

1. _CVT_NEXT_TAG indicates the next value yielded by an iterator.
2. _CVT_TERM_TAG indicates that an iterator has run out of values or that a process should terminate.
3. _CVT_FORALL_TAG indicates that the message data is the next value for a worker to use when running the body of a forall loop.

The fourth built-in constant is _CVT_MAX_WORKERS, an upper bound on the number of tasks to use when processing a forall loop. Each forall loop may be assigned any number of workers from 1 to _CVT_MAX_WORKERS, and CVT will explore executions of the code with each possible number.

3.2 Iterators

In the CIR model, an iterator is represented as a procedure. The procedure takes the same arguments as the original iterator plus an argument of process type to indicate the calling process. Yield statements are replaced by statements to send the yielded value to the calling process. Before the procedure returns, it sends a termination message to the caller.

```
iter foo(arg0, ..., argN) : T {...; yield e; ...}
```

is translated as

```
void foo(process caller, arg0, ..., argN) {
  ...;
  send(caller, e, _CVT_NEXT_TAG);
  ...;
  send(caller, NULL, _CVT_TERM_TAG);
}
```

When a range literal a..b is used as an iterator, CVT replaces this with a call to a built-in implementation of the range literal as an iterator.

```
iter _CVT_range_iterator(lower : int, upper : int) {
  var current : int;
  current = lower;
  while (current <= upper) {
    yield current;  current = current + 1;
  }
}
```

During model construction, the range literal iterator is then translated like any other user-defined iterator.

3.3 Loops

While loops are straightforward to translate into CIR. The various flavors of for loops have more complicated translations due to their use of iterators and implicit parallelism.

For loops. Recall that iterators are modeled as procedures that use send statements to yield values to the calling process. When translating a `for` loop, CVT forks a new process running the iterator, then executes the loop body on each value received from the iterator.

The expression `self` has type `process` and returns the ID of the process in which the expression is evaluated.

```
for x in f(...) S
```

is translated as

```
{ T x;  process p;  int tag;
  p = fork f(self, ...);
  while (true) {
    receive(p, x, any(tag));
    if (tag == _CVT_TERM_TAG) break;
    S
  }
}
```

Parallel loops. `Forall` loops exhibit the greatest degree of nondeterminism of the `for` loop varieties. CVT models `forall` loops using a manager-worker pattern:

```
forall x in f(...) S
```

is translated as

```
{ int numWorkers = 1 + choose(_CVT_MAX_WORKERS);
  int i;  process workers[numWorkers];
  void _CVT_tmp_1(process manager) {
    T x;  int tag;
    while (true) {
      receive(manager, x, any(tag));
      if (tag == _CVT_TERM_TAG) break;
      S
    }
  }
  for (i = 0..numWorkers-1) workers[i] = fork _CVT_tmp1(self);
  { T x;  process iterator;  int tag, dest;
    iterator = fork f(self, ...);
    while (true) {
      receive(iterator, x, any(tag));
      if (tag == _CVT_TERM_TAG) break;
      dest = choose(numWorkers);
      send(workers[dest], x, _CVT_FORALL_TAG);
    }
    for (i = 0..numWorkers-1)
      send(workers[i], NULL, _CVT_TERM_TAG);
    for(i = 0..numWorkers-1) join workers[i];
  }
}
```

The manager receives values from the iterator and assigns them to workers. All possible numbers of workers between one and `_CVT_MAX_WORKERS`, inclusive, are explored.

`Coforall` loops are translated similarly to `forall` loops, but there is always exactly one worker process for each iteration of the loop.

3.4 Cobegin

For each statement in the body of a `cobegin`, a new temporary procedure is created. Each temporary procedure takes no arguments and its body contains just the statement from the `cobegin`. The `cobegin` statement is then translated into a series of `fork` statements followed by a series of `join` statements.

```
cobegin{S1; ...; SN}
```

is translated as

```
{ process _CVT_tmp_procs[N];
  void _CVT_tmp_1() {S1} ... void _CVT_tmp_N() {SN}
  _CVT_tmp_procs[0] = fork _CVT_tmp_1();
  ...;
  _CVT_tmp_procs[N-1] = fork _CVT_tmp_N();
  join _CVT_tmp_procs[0]; ...; join _CVT_tmp_procs[N];
}
```

3.5 Sync variables

A `sync` variable has additional state information indicating whether it is full or empty. CVT tracks this information by introducing a new boolean-valued control variable (initially *false*) for each `sync` variable. For a `sync` variable `foo$` (the symbol `$` is by convention used in `sync` variable identifiers, but otherwise has no special meaning), the associated control variable is `_CVT_sync_foo$`. `foo$ = x;` is translated as `sync-write(_CVT_sync_foo$, foo$, x);` and `x = foo$;` is translated as `sync-read(_CVT_sync_foo$, x, foo$);`. We introduce additional temporary variables as needed to conform to this syntax.

3.6 Composite models

When comparing two programs, CVT must first create a composite CIR model. It does this by creating models for each individual program. Each of these will have an outermost procedure *system* which begins execution and has the outermost scope for that program. CVT then creates a new system procedure. The new system procedure's only task is to fork and join the individual models' system procedures. Variables in the outermost scope of the individual models are moved to the new outermost scope. Any variables with the `extern` modifier are considered to be inputs. Any other variables which are not `const` in the outermost scope are considered to be outputs. When this new model is executed and reaches a terminal state, it checks that all output variables with the same name have the same value.

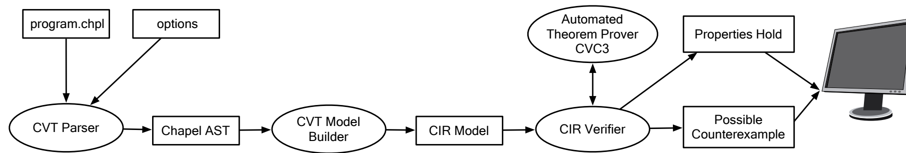


Fig. 5. Dataflow through CVT tool chain.

4 Evaluation

4.1 Tool characteristics

CVT supports two modes. It can be applied to a single Chapel program to verify (or find counterexamples to) certain safety properties, including absence of deadlocks, out-of-bounds array indexing, division by 0, and uses of uninitialized variables. It may also be used to compare two Chapel programs for functional equivalence. This comparison mode can also detect data races and other inappropriate nondeterminism, as such defects can cause two programs to produce different outputs on the same input. Like TASS, CVT uses symbolic execution to verify properties for all possible input values. When CVT compares two programs, it uses the same symbolic values for corresponding input variables. Thus the values of output variables can be compared at termination to check that they hold equivalent symbolic expressions.

CVT is conservative in its analysis. If it says that a program is correct or that two models are equivalent, then those properties must hold within the bounds specified by the program and `_CVT_MAX_WORKERS`. However, CVT is not necessarily precise: it may produce spurious counterexamples.

4.2 Tool structure and implementation

CVT is comprised of several components. These components, and the way data flows between them, are illustrated in Figure 5.

Front end. CVT contains a new front end for the currently supported subset of Chapel. A Java parser generated by the ANTLR parser generator [15] parses the source and builds a Chapel AST. The AST is processed (e.g., adding the implementation of the range literal iterator) to prepare for conversion to a CIR model.

CVT model builder. CVT converts the Chapel AST into a CIR model using the translations discussed in Sec. 3.

CIR verifier. The TASS components used by CVT provide a general framework for symbolic execution and utilize an external theorem prover (currently CVC3 [2]). CVT provides the symbolic execution framework with a representation of the state of a CIR model as described above, and also of guarded

transitions between states. A *simple transition* represents a deterministic move from one state to another. Provided the guard is satisfied, the state is modified based on the statement wrapped by the transition, and there is exactly one resulting state. A *choose transition* provides a nondeterministic choice between a number of values. These are used when a CIR statement utilizes the expression `choose(N)`. In a choose transition, CVT will explore the states resulting from each possible value of the choose expression.

4.3 Partial order reduction

The state space explosion problem is a major issue for formal verification of parallel programs. While any parallel code allows statement executions to interleave in many ways, a language such as Chapel provides the added challenge of dynamically creating an unknown number of processes when executing certain statements.

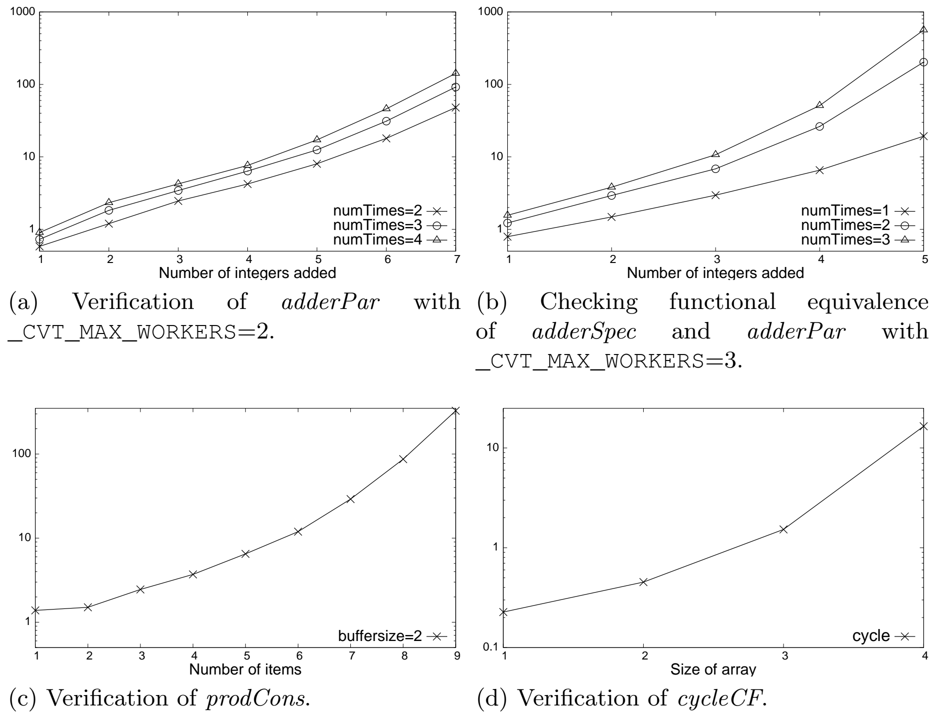
Partial order reduction [9] is a technique to mitigate the state space explosion problem. It relies on the idea that there are equivalence classes of the interleaved executions of a parallel program. That is, certain statements can have the order of their interleaving changed without affecting the outcome. If such a class can be identified, only one such interleaving needs to be explored by a model checker.

At any point in the execution, each process p is at location l_p and in a dyscope δ_p. The outgoing transitions of l_p may contain expressions which refer to variables in δ_p or in any dyscope along the path from δ_p to δ_0. Let δ'_p be the highest dyscope referred to by the outgoing transitions of l_p. Suppose we have some dyscope δ. Let P be the set of processes that can reach δ. If δ is an ancestor of δ'_p for each $p \in P$, then we know that the processes in P can be executed independently of other processes in the program.

4.4 Scaling experiments

We designed several synthetic Chapel programs to test CVT. Each program has several variants. Some variants are believed to be correct, while others contain known defects. While simple, the programs illustrate a number of realistic errors in Chapel codes. Figure 6 summarizes the results of scaling experiments doing both verifications of single programs and functional equivalence comparisons of correct versions. Figure 7 provides some statistics for experiments which were able to detect problems. We next describe the sample programs. The code and experimental results are available at `http://vsl.cis.udel.edu/cvt`.

adder. This program adds numbers in an array multiple times. The simplest version of the code, *adderSpec*, uses nested `for` loops. The inner loop sums the numbers, while the outer loop controls the number of times the addition is performed. Another correct version, *adderPar*, uses a `forall` statement for the inner loop to distribute the addition among multiple tasks. A sync variable is used to store the sum in order to prevent data races. An erroneous version, *adderNoSync*, neglects to make the sum a sync variable. If a `forall` statement



(a) Verification of *adderPar* with _CVT_MAX_WORKERS=2.

(b) Checking functional equivalence of *adderSpec* and *adderPar* with _CVT_MAX_WORKERS=3.

(c) Verification of *prodCons*.

(d) Verification of *cycleCF*.

Fig. 6. CVT performance verifying and comparing correct versions of programs. In all graphs, y-axis is total time in seconds.

is used for the outer loop, the output becomes nondeterministic even though synchronization is used correctly. This occurs in *adderND*.

locks. The *locks* program is a classic deadlock example. A `cobegin` is used to spawn two processes. Each process runs a loop that tries to acquire and release two locks, but the processes acquire the locks in different orders.

```
config const N : int = 100;
var a: [0..N-1] sync int;
proc main() {
  forall i in 0..N-1 {
    var t : int;
    a[(i+1)%N] = i;
    writeln("Wrote ",i);
    t = a[i];
    writeln("Read ",t);
  }
}
```

Fig. 8. The *cycle* program.

cycle. The *cycle* program in Figure 8 performs reads and writes on a circular array of sync variables. Depending on how the loop iterations are partitioned among tasks, it may deadlock. In particular, *cycle* will deadlock if it is executed using one task. The version *cycleCF* avoids this problem by using a `coforall` in place of the `forall`.

Experiment	max workers	states	transitions	max procs	time (s)
adderPar vs *adderNoSync*	20	14495	14496	10	6.260
adderSpec vs *adderNB*	2	7675	7819	12	5.027
locks	N/A	5562	5742	5	2.712
cycle	20	1067	1066	4	1.224
prodCons vs *prodConsNoSync*	N/A	3011	3010	6	2.779

Fig. 7. Results of verifications and comparisons which detect errors. Column "max workers" gives the values of `_CVT_MAX_WORKERS` for that run. The "max procs" column gives the maximum number of active processes at one time during the experiment.

prodCons. The *prodCons* program implements a producer-consumer pattern. The producer adds items from an input array to a circular buffer of sync variables. The consumer reads items from the buffer. The erroneous version, *prodConsNoSync*, neglects to use `sync` variables in the buffer.

5 Conclusion and future work

We have described a new model for verification of Chapel programs. The parallel constructs and spawning of threads in arbitrary scopes in Chapel map naturally to this model. Using model checking with symbolic execution, we have demonstrated the feasibility of automatic verification and defect-detection for non-trivial Chapel programs.

CVT is a prototype tool which works only on a small subset of the full Chapel language. We would like to extend CVT to cover a larger portion of Chapel. In particular, more complex datatypes and arbitrary domains are interesting directions for future work. We would also like to improve scalability. This may be done by developing improved partial order reduction techniques to prune the search tree.

References

1. Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.W., Ryu, S., Steele, Jr., G.L., Tobin-Hochstadt, S.: The Fortress language specification, version 1.0. `http://labs.oracle.com/projects/plrg/Publications/fortress.1.0.pdf` (March 2008)
2. Barrett, C., Tinelli, C.: CVC3. In Damm, W., Hermanns, H., eds.: CAV 2007. Volume 4590 of LNCS., Springer (2007) 298–302
3. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. OOPSLA '05, New York, NY, USA, ACM (2005) 519–538
4. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Softw. Eng. **2** (May 1976) 215–222

5. Cray, Inc.: The Chapel parallel programming language. http://chapel.cray.com/ (2012)
6. Demartini, C., Iosif, R., Sisto, R.: dSPIN: A dynamic extension of SPIN. In: Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking, London, UK, Springer-Verlag (1999) 261–276
7. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8) (August 1975) 453–457
8. Flatt, M., PLT: The Racket reference, version 5.3.1. http://docs.racket-lang.org/reference/ retrieved Nov. 19, 2012.
9. Godefroid, P.: 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)
10. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
11. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In Garavel, H., Hatcliff, J., eds.: TACAS 2003. Volume 2619 of LNCS., Springer (2003) 553–568
12. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7) (1976) 385–394
13. Message Passing Interface Forum: MPI: A message-passing interface standard, version 3.0. http://www.mpi-forum.org/docs/docs.html (September 2012)
14. OpenMP Architecture Review Board: OpenMP application program interface, version 3.1. http://www.openmp.org/mp-documents/OpenMP3.1.pdf (July 2011)
15. Parr, T.: ANTLR Parser Generator. http://www.antlr.org
16. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM TOSEM **17**(2) (2008) Article 10, 1–34
17. Siegel, S.F., Zirkel, T.K.: TASS: The Toolkit for Accurate Scientific Software. Mathematics in Computer Science **5**(4) (2011) 395–426
18. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. Concurrency - Practice and Experience **10**(11-13) (1998) 825–836