Computer Science Technical Report

UPC-SPIN: A Framework for the Model Checking of UPC Programs[1]

Ali Ebnenasir

Michigan Technological University
Computer Science Technical Report
CS-TR-11-03
July 2011

Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu

UPC-SPIN: A Framework for the Model Checking of UPC Programs[†]

Ali Ebnenasir

July 2011

Abstract

This paper presents a method supported by a software framework for the model checking of Unified Parallel C (UPC) programs. The proposed framework includes a front-end compiler that generates finite models of UPC programs in the modeling language of the SPIN model checker. The model generation is based on a set of sound abstraction rules that transform the UPC synchronization primitives to semantically-equivalent code snippets in SPIN's modeling language. The back-end includes SPIN that verifies the generated model. If the model checking succeeds, then the UPC program is correct with respect to properties of interest such as data race-freedom and/or deadlock-freedom. Otherwise, the back-end provides feedback as *sequences of UPC instructions that lead to a data race or a deadlock from initial states*, called *counterexamples*. Using the UPC-SPIN framework, we have detected design flaws in several real-world UPC applications, including a program simulating heat flow in metal rods, parallel bubble sort, parallel data collection, and an integer permutation program. More importantly, for the first time (to the best of our knowledge), we have mechanically verified data race-freedom and deadlock-freedom in a UPC implementation of the Conjugate Gradient (CG) kernel of the NAS Parallel Benchmarks (NPB). We believe that UPC-SPIN provides a valuable tool for developers towards increasing their confidence in the computational results generated by UPC applications.

[*]This work was sponsored by the NSF grant CCF-0950678.
[†]This work was sponsored by the NSF grant CCF-0950678.

1 Introduction

The dependability of High Performance Computing (HPC) software is of paramount importance as researchers and engineers use HPC in critical domains of application (e.g., weather simulations, bio-electromagnetic modeling of human body, etc.) where design flaws may mislead scientists' observations. As such, we need to increase the confidence of developers in the accuracy of computational results. One way to achieve this goal is to devise techniques and tools that facilitate the detection and correction of concurrency failures[1] such as data races, deadlocks and livelocks. Due to the inherent non-determinism of HPC applications, software testing methods often fail to uncover concurrency failures as it is practically expensive (if not impossible) to check all possible interleavings of threads of execution. An alternative method is *model checking* [2–4] where we generate *finite models* of programs that represent a specific behavioral aspect (e.g., inter-thread synchronization functionalities), and exhaustively verify all interleavings of the finite model with respect to a property of interest (e.g., data race/deadlock-freedom). If the model is correct, then the program executions are also correct. Otherwise, the failure of the model may be a spurious error. In such cases, the model can be further refined and model checked again (see Figure 1). This paper presents a novel method (see Figure 1) and a framework (see Figure 2) for the model checking of the Partitioned Global Address Space (PGAS) applications developed in Unified Parallel C (UPC).

While many HPC applications are developed using the Message Passing Interface (MPI) [5], there are important science and engineering problems that can be solved more efficiently in a shared memory model in part because the pattern of data access by independent threads of execution is irregular (e.g., the weighted matching problem [6–8]). As such, while there are tools for the model checking of MPI applications [9–13], we would like to enable the model checking of PGAS applications. The PGAS memory model aims to simplify programming and to increase applications' performance by exploiting data locality in a shared address space.

This paper presents a method (see Figure 1) supported by a framework, called UPC-SPIN (see Figure 2), for the model checking of UPC applications using the SPIN model checker [3], thereby facilitating/automating the debugging of concurrency failures. UPC is a variant of the C programming language that supports the Single Program Multiple Data (SPMD) computation model with the PGAS memory model. UPC has been publicly available for many years and so many HPC users have experience with it. For example, Berkeley's UPC implementation runs on the widest variety of platforms of any PGAS language and so the results of the proposed approach will have the greatest possible audience. The proposed method (see Figure 1) requires programmers to manually specify abstraction rules as a Look-Up Table. Such abstraction rules are property-dependent in that for the same program and different properties, we may need to specify different abstraction rules. The abstraction rules specify how relevant UPC constructs are captured in the modeling language of SPIN. After creating a LUT, UPC-SPIN automatically extracts finite models from the source code and model checks the models with respect to properties of interest. The abstraction LUTs should be kept synchronized with any changes made in the source code. Our experience shows that after creating the first version of an LUT, keeping it synchronized with the source code has a very low overhead. Overall, the proposed method significantly facilitates the debugging of UPC programs as UPC-SPIN has revealed errors in programs believed to be correct (see Section 4 and [14]).

The proposed framework includes two components (see Figure 2): a front-end compiler, and a back-end model checker. The front-end, called UPC Model Extractor (UPC-ModEx), extends the ModEx model extractor of ANSI C programs [15–17] in order to support the UPC grammar. UPC-ModEx takes a UPC program along with a set of abstraction rules and automatically generates a Promela model (see Figure 2).[2] Promela [18] is the modeling language of SPIN, which is an extension of C with additional keywords and abstract data types for modeling concurrent computing systems. We expect that the commonalities of UPC and Promela will simplify the transformation of UPC programs to Promela models and will decrease

[1] In the context of dependable systems [1], *faults* are events that cause a system to reach an *error* state from where system executions may deviate from its specification; i.e., a *failure* may occur.

[2] An executable copy of UPC-ModEx is available at `http://asd.cs.mtu.edu/projects/pgasver/index_files/upc-modex.html`.

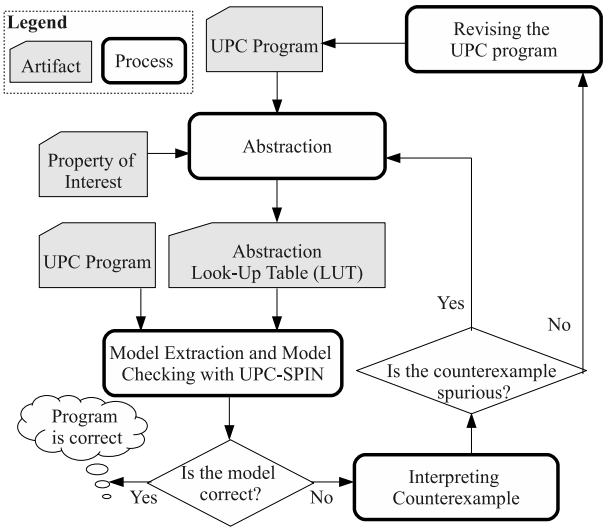

Figure 1: A method for the model checking of UPC programs.

the loss of semantics in such transformations. We present a set of built-in abstraction rules for the most commonly-used UPC synchronization primitives. After generating a finite model in Promela, developers specify properties of interest (e.g., data race-freedom) in terms of either simple assertions or more sophisticated Linear Temporal Logic [19] expressions. SPIN verifies whether or not all executions of the model from its initial states satisfy the specified properties. If the model fails to meet the properties, then a sequence of the UPC program instructions that lead to the failure from the initial states is generated for the programmer (see Figure 2).

We have used UPC-SPIN to detect and correct concurrency failures in small instances (i.e., programs with a few threads) of real-world UPC programs including parallel bubble sort, heat flow in metal rods, integer permutation and parallel data collection. More importantly, for the first time (to the best of our knowledge), we have generated a finite model of a UPC implementation of the Conjugate Gradient (CG) kernel of the NAS Parallel Benchmarks (NPB) [20]. We have model checked small instances of the extracted model of CG for data race-freedom and deadlock-freedom, thereby demonstrating its correctness. While the success of model checking means a model is correct, model checking can only be applied to small/moderate size models (see Section 8) due to the state space explosion problem. This may seem like a significant limitation considering the common belief amongst developers that some defects manifest themselves only when an application is scaled up (in terms of the number of processes and domain size of the input variables/parameters). Nonetheless, there is ample experimental evidence [21, 22] that most concurrency failures also exist in small instances of concurrent applications. A model checker that exhaustively verifies all possible interleavings can detect such failures in small instances of HPC applications. Thus, we believe that model checking can significantly increase the confidence of researchers in the dependability of HPC applications and the scientific conclusions that they draw out of computational results.

Organization. Section 2 provides a background on UPC, model extraction and model checking. Section 3 discusses how the UPC-ModEx front-end extends ModEx to support the UPC constructs. Subsequently, Section 4 illustrates how we model check the Promela models of UPC programs with SPIN. Section 5 presents the model checking of an UPC implementation of the CG kernel of the NPB benchmarks [20]. Section 6 discusses the verification of a parallel bubble sort, and a parallel data collection application is model checked in Section 7. Section 8 presents the time costs of model checking for our case studies. Finally, Section 9 makes concluding remarks and discusses future work.

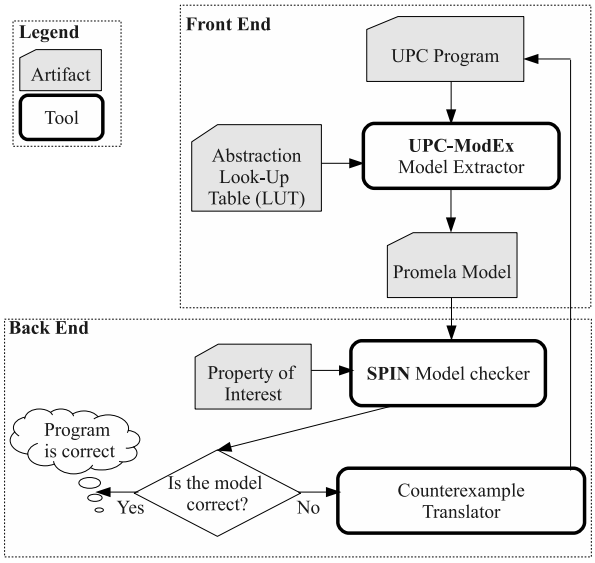

Figure 2: An overview of the UPC-SPIN framework.

2 Preliminaries

This section provides the basic concepts of UPC [23, 24] (Subsection 2.1), finite models of UPC programs (Subsection 2.2), concurrency failures and properties of interest (Subsection 2.3) and an overview of model checking using SPIN [3] (Subsection 2.4). Subsection 2.5 briefly discusses the internal working of the ANSI C Model Extractor (ModEx).

2.1 UPC: An Overview

UPC extends ANSI C by a SPMD model of computation where the same piece of code (e.g., Figure 3) is replicated in distinct threads of execution to process different data streams. The memory model of UPC (i.e., PGAS) divides its address space into shared and private parts. The shared area of the memory space is partitioned into THREADS sections, where THREADS is a system constant representing the total number of threads. Each thread has a private memory space and is also uniquely associated with a shared section, called its *affinity*; e.g., `A[MYTHREAD]` in Line 10 of Figure 3, where `MYTHREAD` denotes thread's own thread number. To support parallel programming, UPC augments C with a set of synchronization primitives, a work-sharing iteration statement `upc_forall` and a set of collective operations. Figure 3 demonstrates an integer permutation application that takes an array of distinct integers (see array `A` in Line 2 of Figure 3) and randomly generates a permutation of `A` without creating any duplicate/missing values. Shared data structures are explicitly declared with a `shared` type modifier. A shared array of `THREADS` locks (of type `upc_lock_t`) is declared in Line 3. Each thread initializes `A[MYTHREAD]` (Line 10) and randomly chooses an array element (Line 14) to swap with the contents of `A[MYTHREAD]`.

2.2 Finite Models of UPC Programs

Let p be a UPC program with a fixed number of threads, denoted $N > 1$. A model of p is a non-deterministic finite state machine denoted by a triple $\langle V_p, \delta_p, I_p \rangle$ representing the inter-thread synchronization functionalities of p, called the *synchronization skeleton* of p. V_p represents a finite set of synchronization variables with finite domains. A *synchronization variable* is a shared variable (e.g., locks) between multiple threads used for synchronizing access to shared resources/variables. A *control variable* (e.g., program counter) captures the execution control of a thread. A *state* is a unique valuation of synchronization and control variables. An ordered pair of states (s_0, s_1) denotes a transition. A *thread* contains a set of transitions, and δ_p denotes the union of the set of transitions of threads of p. I_p represents a set of initial states. The state space of p,

denoted $\mathcal{S}_p$, is equal to the set of all states of p. A *state predicate* is a subset of $\mathcal{S}_p$; i.e., defines a function from $\mathcal{S}_p$ to {true, false}. A state predicate $\mathcal{X}$ is true (i.e., holds) in a state s iff (if and only if) $s \in \mathcal{X}$. A *computation* (i.e., *synchronization trace*) of p is a *maximal* sequence $\sigma = \langle s_0, s_1, \cdots \rangle$ of states s_i, where $s_0 \in I_p$ and each transition (s_i, s_{i+1}) is in δ_p, $i \geq 0$. That is, either σ is infinite, or if σ is a finite sequence $\langle s_0, s_1, \cdots, s_f \rangle$, then no thread is enabled for execution at s_f.

2.3 Concurrency Failures and Properties of Interest

Intuitively, a *safety* property stipulates that nothing bad ever happens in any computation. Data race-freedom and deadlock-freedom are instances of safety properties. A *data race* occurs when multiple threads access shared data simultaneously, and at least one of those accesses is a write [25]. A block of statements accessing shared data is called a *critical section* of the code; e.g., Lines 19-21 and 29-31 in Figure 3 where threads perform the swapping. A data race could occur when two or more threads are in their critical sections. However, the upc_lock statements in Lines 17-18 and 27-28 ensure that each thread gets exclusive access to its critical section so no data races occur. The section of the code where a thread tries to enter its critical section is called its *trying section* (e.g., Lines 17-18 and 27-28). A program is *deadlocked* when no thread can *make progress* in entering its critical section. Deadlocks occur often due to circular-wait scenarios when a set of threads $T_1, \cdots, T_k$ wait for one another in a circular fashion (e.g., T_1 waits for T_2, T_2 waits for T_3 and so on until T_k which waits for T_1). Formally, a deadlock state has no outgoing transitions. The two if-statements in Lines 16 and 26 of Figure 3 impose a total order on the way lock variables are acquired in order to break circular waits.

In the UPC program of Figure 3, a safety property stipulates that *it is always the case* that at most one thread is in its critical section in any computation. In model checkers, such properties are formally specified using the *always* operator in Temporal Logic (TL) [19], denoted $\Box$. The example UPC code of Figure 3 ensures that the safety property $\Box\neg(CS_i \wedge CS_j)$ is met by acquiring locks ($0 \leq i, j <$ THREADS), where CS_i is a state predicate representing that thread i is in its critical section.

A *progress* property states that it is *always* the case that if a predicate P becomes true, then another predicate Q will *eventually* hold. We denote such progress properties by $\mathcal{P} \rightsquigarrow \mathcal{Q}$ (read it as 'P leads to Q') [19]. For example, in the example UPC program of Figure 3, we specify progress for each thread i ($0 \leq i <$ THREADS) as $TS_i \rightsquigarrow CS_i$; i.e., it is always the case that if thread i is in its trying section (represented by the predicate TS_i), then it will eventually enter its critical section (i.e., CS_i holds).

2.4 Model Checking, SPIN and Promela

Explicit-state model checkers (e.g., SPIN [3]) create models as finite-state machines represented as directed graphs in memory, where each node captures a unique state of the model and each arc represents a state transition. Symbolic model checkers create models as Binary Decision Diagrams (BDDs) (e.g., SMV [4]) and are mostly used for hardware verification. If model checking succeeds, then the model is correct. Otherwise, model checkers provide scenarios as to how an error is reached from initial states, called *counterexamples*. SPIN is a explicit-state model checker with a C-like modeling language, called Promela [18]. The syntax of Promela is based on the C programming language. A Promela model comprises (1) a set of variables, (2) a set of (concurrent) processes modeled by a predefined type, called *proctype*, and (3) a set of asynchronous and synchronous channels for inter-process communications. The semantics of Promela is based on an operational model that defines how the actions of processes are interleaved. An action (a.k.a *guarded command*) is of the form $grd \rightarrow stmt$, where the guard grd is an expression in terms of program variables and the statement $stmt$ updates program variables. When the guard grd holds (i.e., the action is *enabled*), the statement $stmt$ can be executed, which accordingly updates some variables. Actions can be atomic or non-atomic, where an atomic action (denoted by the atomic {} blocks in Promela) ensures that the guard evaluation and the execution of the statement is uninterrupted.

2.5 ModEx: Model Extractor of ANSI C Programs

Since in Section 3 we extend the front-end compiler of the ANSI C Model Extractor (ModEx) [15–17] to support the UPC grammar, this section presents an overview of ModEx, which is a software tool for

```
1  // Declaring shared global variables
2  shared int A[THREADS];
3  upc_lock_t *shared lk[THREADS];
4
5 // Declaring thread local variables
6  int i, s, temp;
7
8  // The body of each thread starts
9  // Initialize the array A with distinct integers
10 A[MYTHREAD] = MYTHREAD;
11 upc_barrier;
12 for (i = MYTHREAD; i < MYTHREAD + 1; i++) {
13 // Randomly generate a swap index
14 s = (int)lrand48() % (THREADS);
15
16 if (s<i) {
17  upc_lock(lk[i]);  // Acquire locks
18  upc_lock(lk[s]);
19 temp = A[i];  // Swap
20 A[i] = A[s];
21 A[s] = temp;
22 upc_unlock(lk[s]);  // Release locks
23 upc_unlock(lk[i]);
24    }
25
26 if (i<s) {
27  upc_lock(lk[s]);  // Acquire locks
28  upc_lock(lk[i]);
29 temp = A[i];  // Swap
30 A[i] = A[s];
31 A[s]] = temp;
32 upc_unlock(lk[i]);  // Release locks
33 upc_unlock(lk[s]);
34    }
35 }  // For loop
```

Figure 3: Excerpts of the integer permutation program in UPC.

extracting finite models from ANSI C programs.

ModEx generates finite models of C programs in three phases, namely *parsing, interpretation using abstraction rules* and *optimization for verification*. In the **parsing phase**, ModEx generates an uninterpreted parse tree of the input source code that captures the control flow structure of the source code and the type and scope of each data object. All basic linguistic constructs of C (e.g., declarations, assignments, conditions, function calls, control statements) are collected in the parse tree and remain uninterpreted. The parse tree also keeps some information useful for representing the results of model checking back to the level of source code (e.g., association between the lines of source code and the lines of code in the model). The essence of the **interpretation phase** is based on a *tabled-abstraction* method that pairs each parse tree construct with an interpretation in the target language. ModEx can perform such interpretation based on either a default set of abstraction rules or programmer-defined abstraction rules. Different types of abstractions can be applied to the nodes of the parse tree including *local slicing* and *predicate abstraction*. In local slicing, data objects that are irrelevant to the property of interest (e.g., local variables that have no impact on inter-thread synchronizations) are sliced away. Any operation (e.g., assignments, functional calls) performed on or dependent upon irrelevant data objects are sliced away and replaced with a null operation in the model. In predicate abstraction, if there are variables in the source code whose domains carry more information than necessary for model checking, then they can be abstracted as Boolean variables in the model. For example, consider a variable $0 \leq \mathsf{temp} \leq 100$ that stores the temperature of a boiler tank (in Celsius), and the program should turn off a burner if the temperature is 95 degrees or above. For verifying whether the burner is off when $\mathsf{temp} \geq 95$, a Boolean variable can capture the value of a predicate representing whether or not temp is below 95. In the **optimization** phase, ModEx uses a set of rewrite rules to simplify some generated statements in Promela and to eliminate statements that have no impact on verification. For example, the

guarded command $false \rightarrow x = 0$ in Promela can be omitted without any impact on the result of model checking because the guard is always *false* and the action is never enabled/executed.

3 UPC Model Extractor (UPC-ModEx)

This section discusses how we extend ModEx to support the parsing (Section 3.1) and the interpretation (Section 3.2) of UPC constructs in UPC-ModEx. Since our focus is on the most commonly-used UPC constructs, we skip UPC collectives (except for upc_forall, upc_wait and upc_notify) in this paper. Section 3.3 discusses how we abstract read/write accesses to shared date, and Section 3.4 demonstrates model extraction in the context of the integer permutation program in Figure 3.

3.1 Parsing UPC Constructs

The ANSI C ModEx lacks support for the UPC extension of C including type qualifiers, unary expressions, iteration statements, synchronization statements and UPC collectives. This section discusses how we have augmented ModEx to parse UPC constructs in the UPC-ModEx front-end.

Type qualifiers. UPC includes three type qualifiers, namely shared, strict and relaxed. The shared type qualifier is used to declare data objects in the shared address space. We augment the grammar using the following rules in the BNF form [26]:

- type_qual: CONST | VOLATILE | shared_type_qual | reference_type_qual
- shared_type_qual: "shared" | "shared" '[' opt_const_expr ']' | "shared" '[*]'

The reference type qualifiers strict and relaxed are used to declare variables that are accessed based on the strict or relaxed memory consistency model.

- reference_type_qual: "relaxed" | "strict"

Unary expressions. UPC includes three unary expressions, namely upc_localsizeof, upc_elemsizeof and upc_blocksizeof. upc_localsizeof applies to shared objects/types and returns the size of the local portion of its operand in bytes. The returned value is the same value for all threads. upc_blocksizeof applies to shared objects/types and returns the block size of its operand. upc_elemsizeof applies to shared objects/types and returns the size of the leftmost type that is not in an array. The revised and new grammar rules in BNF are as follows:

- unary_expr: sizeof_expr | upc_localsizeof_expr | upc_blocksizeof_expr | upc_elemsizeof_expr
- upc_localsizeof_expr: "upc_localsizeof" '(' type_name ')' | "upc_localsizeof" unary_expr
- upc_elemsizeof_expr: "upc_elemsizeof" '(' type_name ')' | "upc_elemsizeof" unary_expr
- upc_blocksizeof_expr: "upc_blocksizeof" '(' type_name ')' | "upc_blocksizeof" unary_expr

Iteration statements. In addition to regular iteration statements of C, UPC has a work-sharing iteration statement, denoted upc_forall. The upc_forall statement enables programmers to distribute independent iterations of a for-loop across distinct threads. The grammar of upc_forall in BNF is as follows:

- forall_stemnt: "upc_forall" '(' opt_expr ';' opt_expr ';' opt_expr ';' opt_expr ')' stemnt | "upc_forall" '(' opt_expr ';'opt_expr ';' opt_expr ';' affinity_expr ')' stemnt
- affinity_expr: "continue" | opt_expr

The affinity expression affinity_expr determines which thread executes which iteration of the loop depending on the affinity of the data objects referred in affinity_expr. If affinity_expr is an integer expression *expr*, then each thread executes the body of the loop when MYTHREAD is equal to (*expr mod* THREADS). If affinity_expr is continue or not specified, then each thread executes every iteration of the loop body.

Synchronization statements. The most commonly used synchronization statements in UPC include upc_barrier, upc_wait and upc_notify statements. Moreover, UPC has a new type upc_lock_t that enables programmers to declare lock variables for synchronizing access to shared resources/data. The two functions

upc_lock() and upc_unlock() are used to acquire and release shared variables of type upc_lock_t. The grammar of the new synchronization statements is as follows:

- upc_barrier_stemnt: "upc_barrier" opt_expr ';'
- upc_wait_stemnt: "upc_wait" opt_expr ';'
- upc_notify_stemnt: "upc_notify" opt_expr ';'

We extend the lexical analyzer and the parser of ModEx to support the compilation of UPC-specific constructs discussed above.

3.2 Interpreting UPC Constructs Using Abstraction

This section presents a set of abstraction rules that we have developed for model extraction from UPC programs. After defining the abstractions, we use the ModEx commands [15–17] for the specification of such rules. After parsing a UPC program, UPC-ModEx generates an abstract parse tree whose nodes are UPC constructs in an uninterpreted form. To interpret UPC constructs, we use sound rules that generate Promela code corresponding to the nodes of the parse tree. Each rule is of the form:

left-hand side right-hand side

The left-hand side is a UPC statement and the right-hand side could be either a piece of Promela code that should be generated corresponding to the left-hand side or an abstraction command that specifies how the left-hand side should be treated in the model (see Table 1 for some example commands). For example, the *skip* command generates a null operation in the Promela model corresponding to the left-hand side, the *hide* command conceals the left-hand side in the model; i.e., nothing is generated, and the *keep* command preserves the left-hand side in the Promela model. Some abstraction commands enable string matching and replacement, such as the *Substitute* command. For example, the command

"Substitute MYTHREAD _pid"

replaces any occurrence of MYTHREAD in the UPC code with _pid in the model, where _pid captures a unique identifier for each proctype in Promela. There is also an *Import* command that includes the data objects name from the UPC source code inside the Promela model with the global scope or the scope of a specific proctype in Promela.

Table 1: Sample Abstraction Commands

Command	Meaning
skip	Replace with a null operation
hide	Conceal in the model
keep	Preserve in the model
Substitute P_1 P_2	Substitute any occurrence of P_1 with P_2
Import name scope	Include name with a scope of scope

We present the following abstraction rules for model generation from UPC programs (see [14] for more rules):

Rule 1: upc_lock() The upc_lock(upc_lock_t *lk) function locks a *shared* variable of type upc_lock_t. If the lock is already acquired by some thread, the calling thread waits for the lock to be released. Otherwise, the calling thread acquires the lock *lk atomically. The corresponding Promela code is as follows:

```
1 bool lk;  // Lock variable
2 atomic{ !lk -> lk=true; }
```

Line 2 represents an atomic guarded command in Promela that sets the lock variable lk to true (i.e., acquires lk) if lk is available. Otherwise, the atomic guarded command is blocked.

Rule 2: upc_unlock() The upc_unlock(upc_lock_t *lk) is translated to an assignment lk = false in Promela.

Assignments are executed atomically in Promela.

Rule 3: upc_lock_attempt(upc_lock_t *lk) The upc_lock_attempt(upc_lock_t *lk) tries to get the lock lk if available; otherwise, it will not wait on it. The following Promela code represents a translation of upc_lock_attempt(). The atomic guarded command assigns *true* to lk if it is $false$; otherwise, the atomic statement exits. A Promela 'if statement' checks the guard of each action starting with :: and non-deterministically selects one of them that is true. If there are no true guards, then the entire 'if statement' is blocked.

```
1 atomic{  if  :: (lk == false) -> lk= true;
2              :: else->skip;    fi;   }
```

In round-based computations, all threads include an alternating sequence of upc_notify and upc_wait statements. A *synchronization phase* (i.e., round) starts with the execution of a upc_wait statements and ends with the start of the next upc_wait. By executing a upc_notify statement a thread informs other threads that it is about to reach the next synchronization point, and when it reaches the upc_wait statements it waits until all threads have reached their upc_notify statement in the current phase. Next, we illustrate the abstraction rules corresponding to upc_notify and upc_wait.

Rule 4: upc_notify We use two global integer variables barr and proc to implement the semantics of upc_notify in Promela. Initially, the value of barr is equal to THREADS. To demonstrate that it has reached a notify statement, each thread *atomically* decrements the value of barr and sets the flag proc to zero. Notice that barr and proc are updated atomically because they are shared variables in the model and a non-atomic update may cause data races.

```
1 atomic{ barr = barr -1;  proc=0;}
```

Rule 5: upc_wait Once reached a upc_wait statement, a thread waits until the value of barr becomes zero; i.e., all threads have reached their notify statement in the current synchronization phase. The value of proc is set to 1 indicating that some thread has observed that barr has become zero. Afterwards, each thread increments barr and waits until all threads increment barr or some thread has witnessed that barr has become equal to THREADS in the current phase (i.e., proc has been set to 0).

```
1 (barr == 0) || (proc == 1) -> proc = 1;
2  barr = barr + 1;
3 (barr == THREADS) || (proc == 0) -> proc = 0;
```

Rule 6: upc_barrier The upc_barrier is in fact the union of a pair of upc_notify and upc_wait statements. Separate use of upc_notify and upc_wait implements the split-phase barrier synchronization. Split-phase barrier can reduce the busy-waiting overhead of barrier synchronizations by allowing each thread to perform some local computations between the time it reaches a notify statement and the time it reaches a wait statement.

Rule 7: upc_fence The upc_fence statement enforces a strict consistency model ensuring that all shared memory references issued before upc_fence are complete before any references after upc_fence are issued. Since currently UPC-ModEx generates models only for strict memory references, at any point of execution we are guaranteed that all references to shared variables in the Promela model have been completed before subsequent accesses take place. Thus, UPC-ModEx generates a null statement corresponding to upc_fence.

3.3 Abstracting Shared Data Accesses

In the model checking of concurrent programs for data race-freedom, the objective is to check whether or not multiple threads have simultaneous access to shared data where at least one thread performs a write operation. Thus, the contents of shared variables are irrelevant to verification; rather it is the type of access to shared data that should be captured in a model. For this reason, corresponding to each shared variable x, we consider two bits in the Promela model; one represents whether a read operation is being performed on x and the other captures the fact that x is being written. Accordingly, if a shared array is used in the UPC program, its corresponding model will include two bit-arrays. For example, corresponding to the array A in Figure 3, we consider the following bit arrays:

```
1 bit read_A[THREADS];
2 bit write_A[THREADS];
```

The bit read_A$[i]$ (for $0 \leq i \leq$ THREADS-1) is 1 if and only if a read operation is performed on A$[i]$. Likewise, write_A$[i]$ is 1 if and only if A$[i]$ is written. Thus, corresponding to any read (respectively, write) operation on A$[i]$ in the UPC code, we set read_A$[i]$ (respectively, write_A$[i]$) to 1 in the Promela model.

3.4 Example: Promela Model of Integer Permutation Program

This section illustrates how UPC-ModEx generates a Promela model corresponding to the Permute program of Figure 3. For model extraction, UPC-ModEx needs two input files: the input UPC program and a text file that contains the abstraction LUT. Figure 4 illustrates the abstraction Look-Up Table (LUT) for the program in Figure 3:

```
1  %F Locks.c
2  %X -L main.lut
3  %L
4  Import                i          main
5  Import                s          main
6  Substitute            MYTHREAD  _pid
7  A[MYTHREAD]=MYTHREAD  hide
8  upc_barrier    atomic barr = barr -1;  proc=0;
9                 (barr == 0) || (proc == 1) -> proc = 1;
10                barr = barr + 1  ;
11                (barr == THREADS)||(proc == 0)-> proc = 0;
12 s=((int )lrand48()...    if
13                            :: true -> s = 0;
14                            :: true -> s = 1;
15                            :: true -> s = 2;
16                            :: true -> s = 3;
17                          fi
18 upc_lock(lk[i])         atomic{ !lk[i] -> lk[i] = true }
19 upc_lock(lk[s])         atomic{ !lk[s] -> lk[s] = true }
20 upc_unlock(lk[i])       lk[i] = false
21 upc_unlock(lk[s])       lk[s] = false
22 t=A[i]                  read_A[i]=1;
23                         read_A[i]=0;
24 A[i]=A[s]               read_A[s]=1;
25                         read_A[s]=0;
26                         write_A[i]=1;
27                         write_A[i]=0;
28 A[s]=t                  write_A[s]=1;
29                         write_A[s]=0;
```

Figure 4: The abstraction file for the program in Figure 3, where THREADS $= 4$.

While the commands used in this file are taken from ModEx, the abstraction rules that specify how a model is generated from the UPC program are our contribution. The first line in Figure 4 (i.e., command %F) specifies the name of the source file from which we want to extract a model. Line 2 (i.e., command %X) expresses that UPC-ModEx should extract a model of the main function using the subsequent abstraction rules. Line 3 (i.e., command %L) denotes the start of the look-up table that is used for model extraction. Lines 4 and 5 define that the variables i and s should be included as local variables in the proctype that is generated corresponding to the main function of the source code. Since the contents of array A is irrelevant to the verification of data race/deadlock-freedom, we hide the statement A[MYTHREAD] =MYTHREAD in the model. We apply Rule 6 (presented in Section 3.2) for the abstraction of upc_barrier (Lines 8-11 in Figure 4). Line 14 of Figure 3 (i.e., s = (int)lrand48() % (THREADS)) assigns a randomly–selected integer (between 0 and THREADS-1) to variable s. To capture the semantics of this assignment, we generate a non-deterministic if-statement in the Promela model (Lines 12-17 in Figure 4). Since the guard of every action in this if-statement is true, one action is randomly selected for execution by SPIN at verification time,

thereby assigning a random value to s. The value of the variable s determines the array cell with which the value of $A[i]$ should be swapped by thread i. Lines 18-21 include the rules for the abstraction of upc_lock() and upc_unlock() functions. Lines 22-29 illustrate the rules used to abstract read/write accesses to shared data (as explained in Section 3.3). For example, the assignment $A[i] = A[s]$ in UPC is translated to four assignments demonstrating how $A[s]$ is read and $A[i]$ is written.

Taking the program in Figure 3 and the abstraction file of Figure 4, UPC-ModEx generates the Promela model in Figure 5. Lines 1-5 have been added manually. Line 1 defines a macro that captures the system constant THREADS; in this case 4 threads. Lines 2-6 declare global shared variables that are accessed by all proctypes in the model. The prefix active in Line 8 means that a set of processes are declared that are *active* (i.e., running) in the initial state of the model. The array suffix [THREADS] specifies the number of instances of the main proctype that are created by SPIN. The body of the main proctype is a translation of the UPC program based on the abstraction rules defined in Figure 4. Lines 11-14 implement upc_barrier. The for-loop in Line 12 of Figure 3 is transformed to a do-od loop in Lines 17-59 in Figure 5. This do-od loop terminates in Line 59 if the condition ($i <$_pid$+1$) evaluates to false. Each proctype randomly assigns a value between 0 and THREADS-1 to variable s (Lines 19-23) and then performs the swapping in either one of the if-statements in Lines 24 or 40. Line 56 increments the loop counter. Lines 60-63 implement another upc_barrier. The automatically-generated line numbers that are written as comments in the model associate the instructions in the source code with the statements in the model.

4 Model Checking with SPIN

In order to verify a model with respect to a property, we first have to specify the property in terms of the data flow or the control flow of the model (or both). For example, to verify the model of Figure 5 for data race-freedom, we first determine the conditions under which a shared datum is read and written by multiple threads at the same time. Using the abstractions defined for shared data accesses in Section 3.3, we define the following macros for the Promela model of Figure 5:

```
1 #define race_0  (read_A[0] && write_A[0])
```

The macro race_0 defines conditions under which the array cell A[0] is read and written at the same time; i.e., there is a race condition on A[0]. Likewise, we define macros representing race conditions for other cells of array A. To express data race-freedom in SPIN, we specify the temporal logic expression $\Box$!race_0 meaning that *it is always the case that the condition race_0 is false*. The data race-freedom in this case is guaranteed by the use of upc_lock statements in Lines 17-18 and 27-28 in Figure 3, which are translated as Lines 25-26 and 41-42 in Figure 5. Nonetheless, the use of locks often causes deadlocks in concurrent programs due to circular waiting of threads for shared locks. To verify deadlock-freedom, we must make sure that each thread eventually terminates; i.e., eventually reaches Line 64 of Figure 5. To specify this property, we define the following macros:

```
1 #define fin_0  (main[0]@P)
```

The macro fin_0 is defined in terms of the control flow of the first instance of the main proctype, denoted main[0], which means that thread 0 is at the label P (inserted in Line 64). Thus, if thread 0 eventually reaches its last statement (which is a null operation in Promela denoted by skip), then it is definitely not deadlocked. Such a property is specified as the temporal logic expression $\Diamond$ fin_0, where $\Diamond$ denotes the eventuality operator in temporal logic. SPIN successfully verifies the integer permutation program with respect to data race-freedom and deadlock-freedom properties.

4.1 Example: Heat Flow

The Heat Flow (HF) program includes THREADS>1 threads and a shared array t of size THREADS$\times regLen$, where $regLen > 1$ is the length of a region vector accessible to each thread. That is, each thread i ($0 \leq i \leq$ THREADS-1) has read/write access to array cells $t[i * regLen]$ up to $t[((i + 1) * regLen) - 1]$. The shared

```
#define THREADS 4
int barr = THREADS;
int proc = 0;
bool lk[THREADS];
bit read_A[THREADS];
bit write_A[THREADS];

active [THREADS] proctype main() {
 int i, s;

atomic{ barr = barr -1;  proc=0; } /* line  39 */
(barr == 0) || (proc == 1) -> proc = 1;
barr = barr + 1;
(barr == THREADS) || (proc == 0) -> proc = 0;
               /* line  44 */
 i=_pid;          /* line  50 */
L_0: do
  ::  (i<((_pid+1)))  -> {        /* line  50 */
    if :: true -> s = 0;
       :: true -> s = 1;
       :: true -> s = 2;
       :: true -> s = 3;
    fi;         /* line  53 */
    if ::  (s<i) -> {         /* line  54 */
      atomic{ !lk[i] -> lk[i] = 1 }; /* line  59 */
      atomic{ !lk[s] -> lk[s] = 1 }; /* line  60 */
      read_A[i]=1;         /* line  62 */
      read_A[i]=0;
      read_A[s]=1;         /* line  63 */
      read_A[s]=0;
      write_A[i]=1;
      write_A[i]=0;
      write_A[s]=1;          /* line  64 */
      write_A[s]=0;
      lk[i] = 0;         /* line  66 */
      lk[s] = 0;         /* line  67 */
      }
      :: else;         /* line  67 */
    fi;
    if ::  (s>i) -> {         /* line  67 */
      atomic{ !lk[s] -> lk[s] = 1 }; /* line  70 */
      atomic{ !lk[i] -> lk[i] = 1 }; /* line  71 */
      read_A[i]=1;         /* line  73 */
      read_A[i]=0;
      read_A[s]=1;         /* line  74 */
      read_A[s]=0;
      write_A[i]=1;
      write_A[i]=0;
      write_A[s]=1;         /* line  75 */
      write_A[s]=0;
      lk[i] = 0;         /* line  77 */
      lk[s] = 0;         /* line  78 */
      }
      :: else;         /* line  78 */
    fi;
    i = i + 1;         /* line  78 */
    }
    :: else -> break        /* line  78 */
    od;
    atomic{ barr = barr -1;  proc=0;} /* line  83 */
    (barr == 0) || (proc == 1) -> proc = 1;
    barr = barr + 1  ;
    (barr == THREADS) || (proc == 0) -> proc = 0;
      P: skip;  }
```

Figure 5: The Promela model generated for the program in Figure 3.

array t captures the transfer of heat in a metal rod and the HF program models the heat flow in the rod. We present an excerpt of the UPC code of HF as follows:

```
1  shared double t[regLen*THREADS];
2  double tmp;
3
4  . . .  //  Perform some local computations
5  upc_barrier;
6  base = MYTHREAD*regLen;
7  for (j =0; j < regLen+1; j++) {
8   if (MYTHREAD == 0) { tmp[0] = t[0]; }
9   else {
10     tmp[0] = (t[base-1] + t[base] + t[base+1])/3.0;
11     e = fabs(t[base]-tmp[0]); }
12  for (i=base+1; i<base+regLen-1; ++i) {
13    tmp = (t[i-1] + t[i] + t[i+1]) / 3.0;
14    etmp = fabs(t[i]-tmp[1]);
15    t[i-1] =  tmp[0];
16    }
17  if (MYTHREAD < THREADS-1) {
18    tmp = (t[base+regLen-2] +
19                  t[base+regLen-1] +
20                          t[base+regLen]) / 3.0;
21    etmp = fabs(t[base+regLen-1]-tmp[1]);
22    t[base+regLen-1] = tmp[1];
23    }
24  upc_barrier;
25  t[base+regLen-2] = tmp[0];
26 }
```

Figure 6: Excerpt of the Heat Flow (HF) program in UPC.

Each thread performs some local computations and then all threads synchronize with the upc_barrier in Line 5. The base of the region of each thread is computed by MYTHREAD$*$ regLen in Line 6. Each thread continuously executes the code in Lines 7 to 26. In Lines 8-11, the local value of tmp[0] is initialized. Then, in Lines 12 to 16, each thread i, where $0 \leq i \leq$ THREADS-1, first computes the heat intensity of the cells $t[base]$ to $t[base + regLen - 3]$ in its own region. Subsequently, every thread, except the last one, updates the heat intensity of $t[base+regLen-1]$ (see Lines 17-23). Before updating $t[base+regLen-2]$ in Line 25, all threads synchronize using upc_barrier. Then they compute an expression that is assigned to $t[base+regLen-2]$. Our objective is to verify whether or not there are any data races in HF. The significance of this example is that the access to shared data is changed dynamically as each thread updates the value of heat flow. Moreover, despite the small number of lines of code in this example, it is difficult to *manually* identify where the data races may occur.

Abstraction Look-Up Table (LUT) for HF. We present the abstraction LUT of the HF program below. Lines 1-7 of the table include the local data and simple mapping rules. The rest of the abstraction table includes 11 entries located in Lines 8, 10, 14, 21, 24, 31, 33, 35, 38, 40 and 42. Each entry includes a left-hand side and a right-hand side defined based on the rules presented in Sections 3.2 and 3.3. Hence, we omit the explanation of the abstraction rules of the HF program. Notice that the arrays read_t and write_t have been declared for the abstraction of data accesses to the shared array t (as explained in Section 3.3).

```
1  %F fwup.c
2  %X -L main.lut
3  %L
4  Import   i   main
5  Import   j   main
6  Import   base   main
7  Substitute   MYTHREAD   _pid
8  tmp[0]=t[0]    read_t[0]=1;
9                 read_t[0]=0;
10 upc_barrier    atomic barr = barr -1;  proc=0;
11                (barr == 0) || (proc == 1) -> proc = 1;
```

```
12                barr = barr + 1  ;
13                (barr==THREADS)||(proc==0) -> proc = 0;
14 tmp[0]=(((t[(base-1)]+t[base])+t[(base+1)])/3)
15                read_t[base-1] = 1;
16                read_t[base-1] = 0;
17                read_t[base] = 1;
18                read_t[base] = 0;
19                read_t[base+1] = 1;
20                read_t[base+1] = 0;
21 e=fabs((t[base]-tmp[0]))    read_t[base]=1;
22                             read_t[base]=0;
23
24 tmp[1]=(((t[(i-1)]+t[i])+
25                t[(i+1)])/3) read_t[i-1] = 1;
26                             read_t[i-1] = 0;
27                             read_t[i] = 1;
28                             read_t[i] = 0;
29                             read_t[i+1] = 1;
30                             read_t[i+1] = 0;
31 etmp=fabs((t[i]-tmp[1]))    read_t[base]=1;
32                             read_t[base]=0;
33 t[(i-1)]=tmp[0]             write_t[i-1] = 1;
34                             write_t[i-1] = 0;
35 etmp=fabs((t[((base+regLen)-1)]-tmp[1]))
36                read_t[((base+regLen)-1)]=1;
37                read_t[((base+regLen)-1)]=0;
38 t[((base+regLen)-1)]=tmp[1] write_t[((base+regLen)-1)]=1;
39                             write_t[((base+regLen)-1)]=0;
40 t[((base+regLen)-2)]=tmp[0] write_t[base+regLen-2]=1;
41                             write_t[base+regLen-2]=0;
42 tmp[1]=(((t[((base+regLen)-2)]+ t[((base+regLen)-1)])
43 +t[(base+regLen)])/3) read_t[((base+regLen)-2)] = 1;
44                       read_t[((base+regLen)-2)] = 0;
45                       read_t[((base+regLen)-1)] = 1;
46                       read_t[((base+regLen)-1)] = 0;
47                       read_t[(base+regLen)] = 1;
48                       read_t[(base+regLen)] = 0;
```

The Promela model of HF. UPC-ModEx generates the following Promela model for the HF program using its abstraction LUT. This is an instance with 3 threads and region size of 3 for each thread. The SPIN model checker creates THREADS instances of the main proctype as declared in Line 8. We omit the explanation of the Promela model of HF as it has been generated with the rules defined in Sections 3.2 and 3.3.

```
1 #define regLen 3
2 #define THREADS 3
3 int barr = THREADS;
4 int proc = 0;
5 bit read_t[9];
6 bit write_t[9];
7
8 active [THREADS] proctype main() {
9 int base ;        /* mapped */
10 int i ;        /* mapped */
11 int j ;        /* mapped */
12 atomic{ barr = barr -1;  proc=0; }   /* line  50 */
13 (barr == 0) || (proc == 1) -> proc = 1;
14 barr = barr + 1 ;
15 (barr == THREADS) || (proc == 0) -> proc = 0;
16 base = _pid * regLen;  /*  line  55 */
17 j=0;          /* line  60 */
18L_0: do
19    :: (j<(regLen +1)) -> { /* line  60 */
20       if :: (_pid==0) /* line  60 */
```

```
21              read_t[0]=1; read_t[0]=0; /* line  64 */
22           :: else;         /* line  64 */
23        read_t[base-1] = 1; read_t[base-1] = 0;
24        read_t[base] = 1; read_t[base] = 0;
25        read_t[base+1] = 1; read_t[base+1] = 0;
26        /* line  68 */
27        read_t[base]=1; read_t[base]=0; /* line  69 */
28        fi;
29        i = base +1;  /* line  71 */
30L_1: do
31       :: (i < base+regLen-1) -> {
32          read_t[i-1] = 1;  read_t[i-1] = 0;
33          read_t[i] = 1; read_t[i] = 0;
34          read_t[i+1] = 1; read_t[i+1] = 0;
35          /* line  73 */
36          read_t[base]=1; read_t[base]=0; /* line  74 */
37          write_t[i-1] = 1; write_t[i-1] = 0; /* line  76 */
38          i = i +1;  /* line  76 */
39          }
40       :: else  -> break           /* line  76 */
41       od;
42       if :: (_pid<(THREADS-1))  /* line  76 */
43          read_t[((base+regLen)-2)] = 1; /* line  81 */
44          read_t[((base+regLen)-2)] = 0;
45          read_t[((base+regLen)-1)] = 1;
46          read_t[((base+regLen)-1)] = 0;
47          read_t[(base+regLen)] = 1;
48          read_t[(base+regLen)] = 0;
49          read_t[((base+regLen)-1)]=1; /* line  83 */
50          read_t[((base+regLen)-1)]=0;
51          write_t[((base+regLen)-1)]=1; /* line  87 */
52          write_t[((base+regLen)-1)]=0;
53          :: else;         /* line  87 */
54       fi;
55       atomic{barr = barr -1; proc=0;} /* line  89 */
56       (barr == 0) || (proc == 1) -> proc = 1;
57       barr = barr + 1 ;
58       (barr == THREADS) || (proc == 0) -> proc = 0;
59       write_t[base+regLen-2]=1;
60       write_t[base+regLen-2]=0; /* line  91 */
61       j = j +1 ; /* line  91 */
62      }
63     :: else -> break  /* line  91 */
64  od;
65}
```

Model checking for data race-freedom. We specify data race conditions as the following macros that must always be false:

```
1 #define race_0 (read_t[0] && write_t[0])
```

To verify data race-freedom, we use SPIN to check whether or not the following property is correct: *it is always the case that there are no simultaneous reads and writes on array cell* $t[i]$. This parameterized property is formally expressed as $\Box\neg$ race_i. We verify the model of the HF program for data race-freedom on all memory cells of array t; i.e., $0 \leq i \leq$ THREADS$*$regLen-1. For an instance of the model where THREADS$= 3$ and regLen$= 3$, SPIN finds data races on the boundary array cells. That is, there are data races on $t[2], t[3], t[5]$ and $t[6]$. For instance, in the source code of HF (see Figure 6), a data race occurs on $t[2]$ when thread 0 is writing $t[2]$ in Line 22 and thread 1 is reading $t[2]$ in Line 10. Another data race occurs when thread i is writing its t[base] in Line 15 of Figure 6 in the first iteration of the for-loop of Line 12, and thread $i-1$ is reading t[base+regLen] in Line 20, for $0 \leq i \leq$ THREADS-1. This example illustrates how model checking could simplify the detection of data races. These data races can be corrected by using lock variables in appropriate places in the code.

5 Case Study: Model Checking the Conjugate Gradient Kernel of NPB

This section presents a case study on verifying the data race-freedom and deadlock-freedom of a UPC implementation of the CG kernel of the NPB benchmark (taken from [20]). The NPB benchmarks provide a set of core example applications, called *kernels*, that are used to evaluate the performance of highly parallel supercomputers. The kernels of NPB test different types of applications in terms of their patterns of data access and inter-thread communication. Since the PGAS model is appropriate for solving problems that have irregular data accesses, we select the CG kernel of NPB that implements the conjugate gradient method by the inverse power method (which has an irregular data access pattern). Another interesting feature of CG is the use of a two-dimensional array in the affinity of each thread and the way array cells are accessed. To the best of our knowledge, this section presents the first attempt at mechanical verification of CG. Figure 7 demonstrates the inter-thread synchronization functionalities of CG.

```
1  struct cg_reduce_s { double v[2][NUM_PROC_COLS]; };
2  typedef struct cg_reduce_s cg_reduce_t;
3  shared cg_reduce_t sh_reduce[THREADS];
4
5  int main(int argc, char **argv) {
6     // declaration of local variable
7     // Initialization
8     upc_barrier;
9    // Perform one (untimed) iteration to
10   // initialize code and data page tables
11   /* The call to the conjugate gradient routine */
12    conj_grad( . . . );
13    reduce_sum_2( . . . );
14   // post-processing
15   upc_barrier;
16   /* Main Iteration for inverse power method */
17   for (it = 1; it <= NITER; it++)  {
18        conj_grad( . . . );
19        reduce_sum_2( . . . );
20       // post-processing
21         }
22   upc_barrier;
23   // Thread 0 prints the results
24 }
25
26 void reduce_sum_2( double rs_a, double rs_b ) {
27  int rs_i;
28  int rs_o;
29
30  upc_barrier;
31  rs_o = (nr_row * NUM_PROC_COLS);
32  for( rs_i=rs_o; rs_i<(rs_o+NUM_PROC_COLS); rs_i++ ){
33     sh_reduce[rs_i].v[0][MYTHREAD-rs_o] = rs_a;
34     sh_reduce[rs_i].v[1][MYTHREAD-rs_o] = rs_b;    }
35  upc_barrier;
36 }
```

Figure 7: Inter-thread synchronization functionalities of the Conjugate Gradient (CG) kernel of the NPB benchmarks.

The first three lines in Figure 7 define a data structure that is used to store the results of computations in a collective reduce fashion. The cg_reduce_s structure captures a two dimensional vector, and the shared array sh_reduce defines a two dimensional vector in the affinity of each thread. After performing some local initializations, all threads synchronize using upc_barrier in Line 8 of Figure 7. Then an untimed iteration of the inverse power method is executed (Lines 9-10) before all threads synchronize again. The reduce_sum_2 routine distributes the results in the shared address space. The for-loop in Line 17 implements the inverse

power method. Afterwards all threads synchronize in Line 22, and then Thread 0 prints out the results. The main difficulty is in the way we abstract the pattern of data accesses in reduce_sum_2.

Abstraction. To capture the way write operations are performed, we consider the following abstract data structures in the Promela model corresponding to sh_reduce:

```
1 typedef bitValStruc { bit b[NUM_PROC_COLS] };
2 typedef bitStruc { bitValStruc v[2] };
3 bitStruc write_sh_reduce[THREADS];
```

Lines 1-2 above define a two dimensional bit array (of type bitStruc) in Promela and Line 3 declares a bit array of bitStruc with size THREADS. Next, we abstract the for-loop of reduce_sum_2 in Promela as follows:

```
1 rso = (_pid / NUM_PROC_COLS) * NUM_PROC_COLS;
2 rsi = rso;
3 do
4   :: (rsi < rso + NUM_PROC_COLS) -> {
5       write_sh_reduce[rsi].v[0].b[_pid-rso] = 1;
6       write_sh_reduce[rsi].v[0].b[_pid-rso] = 0;
7       write_sh_reduce[rsi].v[1].b[_pid-rso] = 1;
8       write_sh_reduce[rsi].v[1].b[_pid-rso] = 0;
9       rsi = rsi + 1; }
10  :: !(rsi < _pid + NUM_PROC_COLS) -> break;
11 od;
```

Observe that the for-loop in the reduce_sum_2 function in Figure 7 has been captured by a do-od loop in the Promela model of CG and any write operation has been modeled by setting and resetting the corresponding bit in the array write_sh_reduce.

Model checking CG for data race-freedom. To model check CG for data race-freedom, we have to ensure no two threads simultaneously write on the same memory cell. In other words, we should verify that before setting a bit in the array write_sh_reduce, that bit is not already set by another thread. We insert an *assert* statement (Lines 3 and 7 below) that verifies data race-freedom. While model checking, SPIN verifies that the assertions hold. In the case of CG for 2, 3 and 4 threads, we found no data races.

```
1  do
2   :: (rsi < rso + NUM_PROC_COLS) -> {
3       assert(write_sh_reduce[rsi].v[0].b[_pid-rso] != 1);
4       write_sh_reduce[rsi].v[0].b[_pid-rso] = 1;
5       write_sh_reduce[rsi].v[0].b[_pid-rso] = 0;
6
7       assert(write_sh_reduce[rsi].v[1].b[_pid-rso] != 1);
8       write_sh_reduce[rsi].v[1].b[_pid-rso] = 1;
9       write_sh_reduce[rsi].v[1].b[_pid-rso] = 0;
10      rsi = rsi + 1; }
11  :: !(rsi < _pid + NUM_PROC_COLS) -> break;
12 od;
```

Model checking CG for deadlock-freedom. To make sure that each thread eventually terminates, we insert a label "fin:" for a null operation *skip* as the last operation in the body of each thread (similar to the label P in Line 64 of Figure 5). Then we define the following macro:

```
1 #define fin_0  (main[0]@fin)
```

We use similar macros to specify a progress property as $\Diamond$ fin_i for each thread i stating that each thread will eventually terminate; i.e., will not deadlock. SPIN verified that each thread terminates for instances of the CG kernel with 2, 3 and 4 threads.

6 Case Study: Parallel Bubble Sort

This section presents the model extraction and model checking of a parallel bubble sort algorithm implemented in UPC. The significance of this example is two-fold. First, we illustrate how we make abstractions of conditional if-statements for model checking. Second, the nature of critical section is *dynamic* in this case; i.e., depending on the *run-time* value of the array indices in each thread, the place of the critical section of the code changes. Line 1 below declares a shared array for the unsorted array of integers, and Line 2 declares two lock variables for each memory cell. The affinity of each thread includes N integers, which is partitioned into two sections (i.e., sectionSize= $N/2$). That is, each thread has two sections. curlock and oldlock are two lock variables that keep track of the array cell that is being relocated for sorting. The variables mc, firsttime and done are used as Boolean flags. Each thread starts the bubble sort at $N*$ MYTHREAD and scans the entire array to the end. The scanning repeats until the entire array is sorted. The local variables i and l in each thread hold the indices of the array cells that are compared with each other and swapped if necessary (see Lines 43-48). The variable j is a counter that keeps track of the array elements in the current section. Once the elements of a section have been all visited, the next section is locked (see Lines 30-36 and Lines 49-52). The way variables i and l are incremented ensures that threads are synchronized in a pair-wise fashion and they do not pass each other during sorting (Lines 40-42 and 55-57). Observe that since the values of i and l are determined at run-time, the critical section of each thread moves to different places in the array.

```
1  shared int * shared array;
2  shared upc_lock_t * shared locks[THREADS*2];
3
4  int main(int argc, char *argv) {
5  /* local variables per thread */
6  int curlock, N, oldlock, sectionSize, temp;
7  int mc, firsttime, done;
8  int i, j, l; // Loop counters
9  // Initialization and memory allocation
10 curlock = MYTHREAD*2;
11 /* initialize array */
12 upc_barrier;
13 /* Thread 0 prints out the unsorted array */
14 /* start of sorting  */
15 i = MYTHREAD; // start of this thread's section
16 j = N*MYTHREAD; mc = 0;
17 firsttime = 1; done = 0;
18 upc_lock(locks[curlock]);
19 do {
20  /* if back where I started, see if I should terminate */
21   if (j == N*MYTHREAD) {
22      if (firsttime) firsttime = 0;
23      else {
24           if (mc == 0) { done = 1;
25            upc_unlock(locks[curlock]); }
26            mc = 0;
27           }
28    }
29   if (!done) {
30       if (j%sectionSize == sectionSize-1) {
31           /* if at the end of a section... */
32            oldlock = curlock;
33            curlock = (curlock+1)%(THREADS*2);
34           /* obtain lock of next section */
35            upc_lock(locks[curlock]);
36        }
37     /* check to see if last element in current section *
38      * is greater than the first element of the next   *
39      * section, if so, swap.                           */
40       l = i + THREADS;
41       if (l >= N*THREADS)
42       l = l%(N*THREADS)+1;
```

```
43       if (array[i] > array[l]) {
44           temp = array[i];
45           array[i] = array[l];
46           array[l] = temp;
47           mc = 1;
48          }
49       if (j%sectionSize == sectionSize-1) {
50       /* release lock of current section */
51           upc_unlock(locks[oldlock]);
52        }
53     /* increment counter, wrap to beginning of array *
54      * when at the last element.                     */
55           i += THREADS;
56       if (i >= N*THREADS) i = i%(N*THREADS)+1;
57       if (i == N*THREADS-1) i = 0;
58       j = (j+1)%(N*THREADS);
59       if (j == N*THREADS-1) j = 0;
60       }
61    } while (!done);
62   /* synchronize before Thread 0 prints sorted array */
63 upc_barrier;
64  /* Thread 0 prints out the sorted array */
65 }
```

Abstraction and model extraction. Since most of the abstraction rules are similar to the ones used for previous examples, we omit the abstraction LUT and present the extracted Promela model. The below Promela code is for an instance of the parallel bubble sort where $N = 6$ and THREADS$= 3$. The only new abstraction we have in this example is the abstraction corresponding to the conditions of the if-statements (e.g., the if-statements in Line 43 of the source code of the parallel bubble sort) that either the then part or the else part or both include a critical section of the code where shared data is read/written. In the case of bubble sort, when the entire array is sorted, the if-statements in Line 43 will never become true anymore. That is, the variable mc will never be set to 1; i.e., remains 0 for at least two consecutive iterations of the do-while loop. Thus, the condition of the if-statements in Line 24 will eventually become true; i.e., done will eventually be set to 1. That is, the algorithm will eventually terminate. Now, to model the if-statements in Line 43, we consider a down counter variable cnt in the Promela model. As long as cnt is non-zero the modeled condition (in Line 63 below) evaluates to true. Once cnt becomes zero, it captures the scenario in the UPC code where the array is sorted and no more swapping is performed in Lines 43-48 of the UPC code.

```
1  #define N 6 /* number of elements in each thread's region */
2  #define THREADS 3
3  #define sectionSize 3 /* is N/2 */
4  int barr = THREADS;
5  int proc = 0;
6  bit read_a[18]; /* (N * THREADS) */
7  bit write_a[18]; /* (N * THREADS) */
8  bool locks[6]; /* (2 * THREADS) */
9
10 active [THREADS] proctype main() {
11  int i; /* mapped */
12  int j; /* mapped */
13  int l; /* mapped */
14  int mc; /* mapped */
15  int firsttime; /* mapped */
16  int done; /* mapped */
17  int curlock; /* mapped */
18  int oldlock; /* mapped */
19  int cnt = 1;
20  curlock=_pid*2; /* line  50 */
21  i=_pid; /* line  83 */
22  j=(N*_pid); /* line  84 */
23  mc=0; /* line  84 */
```

```
firsttime=1; /* line  85 */
done=0; /* line  85 */
/* line  86 */
atomic{!locks[curlock] -> locks[curlock] = 1};
atomic{ barr = barr -1;  proc=0;} /* line  94 */
(barr == 0)||(proc == 1)-> proc = 1;
barr = barr + 1;
(barr == THREADS)||(proc == 0)->proc = 0;
L_0: { /* do-while */
     if :: (j==(N*_pid))-> /* line 100 */
        if :: firsttime -> /* line 100 */
              firsttime=0; /* line 103 */
           :: else -> {   /* line 103 */
              if :: (mc==0)->{ /* line 103 */
                  done=1; /* line 106 */
                  locks[curlock] = 0; /* line 107 */
                  }
                 :: else -> skip; /* line 107 */
              fi;
              mc=0; /* line 109 */
              }
        fi;
       :: else -> skip;
     fi;
     if :: (!done) -> {
          if :: ((j%sectionSize)==(sectionSize-1)) -> {
             oldlock = curlock; /* line 116 */
             curlock=((curlock+1)%(THREADS*2));
              /*line 117*/
             atomic{ !locks[curlock] -> locks[curlock] = 1};
             /* line 120 */ }
          :: else -> skip; /* line 120 */
           fi;
          l= i+THREADS; /* line 126 */
         if
           :: (l >= (N*THREADS)) -> /* line 126 */
               l=((l%(N*THREADS))+1); /* line 128 */
           :: else -> skip; /* line 128 */
         fi;
         if :: (cnt > 0) -> {
               read_a[i]=1; read_a[i]=0; /* line 131 */
               read_a[l]=1; read_a[l]=0;
               write_a[i]=1; write_a[i]=0; /* line 132 */
               write_a[l]=1; write_a[l]=0; /* line 133 */
               mc=1; /* line 134 */
               cnt = cnt -1;
               }
            :: else-> skip;
         fi;
         if   /* line 134 */
            ::((j%sectionSize)==(sectionSize-1)) ->
               locks[oldlock] = 0; /* line 139 */
            :: else-> skip; /* line 139 */
         fi;
         i = i + THREADS; /* line 144 */
         if :: (i >= (N*THREADS)) -> /* line 144 */
               i=((i%(N*THREADS))+1); /* line 146 */
            :: else-> skip; /* line 146 */
         fi;
         if :: (i==((N*THREADS)-1)) -> /* line 146 */
                i=0; /* line 147 */
            :: else-> skip; /* line 147 */
         fi;
         j=((j+1)%(N*THREADS)); /* line 148 */
         if :: (j==((N*THREADS)-1)) -> /* line 148 */
```

```
89                 j=0; /* line 149 */
90             :: else-> skip; /* line 149 */
91          fi;
92       }
93      :: else-> skip;
94      fi;
95   if :: (!done) -> goto L_0;
96      :: else -> skip;
97   fi;
98 atomic{ barr = barr -1;  proc=0;}
99 (barr == 0)||(proc == 1)-> proc = 1;
100 barr = barr + 1;
101 (barr == THREADS)||(proc == 0) -> proc = 0;
102   /*line 154 */
103  R: skip;
104 }
```

Property specification and model checking. We have model checked both data race-freedom and deadlock-freedom of the model. The property specification is similar to previous examples, where for data race-freedom we declare bit arrays for read and write operations, and for deadlock-freedom of each thread, we use a label with a null operation as the last statement (see the label R in Line 103 below). We have verified the bubble sort for models with 3, 4 and 5 threads, and the region size of 6. The maximum model checking time is 77.8 seconds for data race-freedom of the instance with 5 threads.

```
1 #define race (read_t[0] && write_t[0])
2 #define fin0 (main[0]@R)
3 #define fin1 (main[1]@R)
4 #define fin2 (main[2]@R)
```

7 Case Study: Parallel Data Collection

The Parallel Data Collection (PDC) program gets samples for ITERATION times for a histogram with SIZE bars. The function foo() in the below UPC code generates the samples based on a non-uniform probability distribution function, and the array hist contains the histogram. After constructing the histogram, its data is scaled and averaged in an array representing a discrete form of the signal. Our objective is to ensure that no data races occur on the shared array hist. The following UPC code is a sliced version of the original UPC code of PDC where any data objects (and their associated instructions) that are irrelevant to data race-freedom of hist have been abstracted away.

```
1 shared [*] int hist[SIZE];
2 int main () {
3  int trial = 1;
4  int i, s;
5  do {
6   // Initialize and compute the histogram.
7    for(i = 0; i<ITERATIONS; ++i) {
8      if (MYTHREAD == (i%THREADS)) {
9          s = foo(SIZE);
10         ++hist[s];            }
11    }
12   upc_barrier;
13   //  compute error
14   ++trial;
15  } while (error > EPSILON);
16}
```

Abstraction and model extraction. UPC-ModEx generates the following model based on the existing abstraction rules. This is an instance of the model where SIZE= 15 and THREADS= 3. The bit arrays read_hist and write_hist (Lines 5-6 below) model read and write accesses to the shared array hist. The c_code statement blocks (see Lines 13, 17, 45 and 53 below) specify that the statements in the block are to be treated as

ANSI C statements. Likewise, c_exper (see Lines 20-21) evaluates the expression in the c_exper block as a C expression.

```
1 #define SIZE 15
2 #define THREADS 3
3 int barr = THREADS;
4 int proc = 0;
5 bit read_hist[15];
6 bit write_hist[15];
7
8 active [THREADS] proctype main() {
9  int cnt = 3;
10 int i; /* mapped */
11 int s; /* mapped */
12 int trial; /* mapped */
13 c_code { trial=1; }; /* line 111 */
14L_1:
15 do /* do-while */
16   :: /* for-loop: */
17    c_code { i=0; }; /* line 121 */
18   L_2:
19     do
20     :: c_expr { (i<2) }; /* line 121 */
21        if :: c_expr{(_pid==(i%THREADS))};
22               /* line 121 */
23            if :: true -> s = 0;
24               :: true -> s = 1;
25               :: true -> s = 2;
26               :: true -> s = 3;
27               :: true -> s = 4;
28               :: true -> s = 5;
29               :: true -> s = 6;
30               :: true -> s = 7;
31               :: true -> s = 8;
32               :: true -> s = 9;
33               :: true -> s = 10;
34               :: true -> s = 11;
35               :: true -> s = 12;
36               :: true -> s = 13;
37               :: true -> s = 14;
38             fi;  /* line 124 */
39             read_hist[s] = 1;
40             read_hist[s] = 0;
41             write_hist[s] = 1;
42             write_hist[s] = 0; /* line 124 */
43          :: else; /* line 124 */
44        fi;
45        c_code { ++i; }; /* line 124 */
46      :: else  -> break
47      od;
48      atomic{ barr = barr -1;  proc=0;}
49      (barr == 0)||(proc == 1)->proc = 1;
50      barr = barr + 1;
51      (barr == THREADS)||(proc == 0)
52                        ->proc = 0; /* line 135 */
53      c_code { ++trial; }; /* line 135 */
54     if :: (cnt >= 0) -> cnt = cnt -1;
55        :: else -> break;
56     fi
57 od;
58}
```

Model checking. The model checking of PDC reveals a data race on the hist array as multiple threads could simultaneously execute Line 10 in the UPC code (Lines 39 to 42 in the Promela model of PDC) and cause a race condition.

8 Experimental Results

In order to give a measure of the time cost of model checking, this section presents our experimental results for the model checking of several real-world UPC programs including the integer permutation program of Figure 3, the Heat Flow program of Figure 6 and the CG program of Figure 7. The platform of model checking is an HP Tablet PC with an Intel Dual Core Processor T5600 (1.83 GHz) and 1GB memory available for model checking.

Model checking of integer permutation. We have model checked the integer permutation program of Figure 3 for data race-freedom and deadlock-freedom, where $3 \leq \mathsf{THREADS} \leq 5$. Verifying data race-freedom took 0.047 and 5.11 seconds respectively for models with 3 and 4 threads. The model checking of thread termination needs more time as we spent 0.125 and 12.4 seconds for instances with 3 and 4 threads. Since the algorithm for the model checking of progress properties is more complex, more time is spent for the verification of progress properties. SPIN explored 2.7 million states in the model checking of integer permutation for 4 threads. For $\mathsf{THREADS} = 5$, model checking of both data race-freedom and deadlock-freedom took much longer and eventually failed due to insufficient memory.

Model checking of heat flow. For 3 threads and region size 3, the model checking took 0.8 seconds for cases where the model satisfied data race-freedom, called the *success cases*, and 0.171 Sec. for cases where the model violates the data race-freedom, called the *failure cases* (e.g., accessing the array cells that are on the boundary of two neighboring regions). Once a failure is found, SPIN stops and provides a counterexample, thereby avoiding the exploration of the unexplored reachable states. For this reason, the verification of failure cases takes less time. For 4 threads and region size 3, model checking took 20.5 seconds for success cases (i.e., for array cells where no data races occur) and 0.171 for failure cases. For 5 threads, SPIN failed to give any results due to memory constraints.

To investigate the impact of the region size on the time complexity of model checking, we increased the region size for the case of $\mathsf{THREADS} = 3$. For 3 threads and region size 4, verification took 2.14 Sec for success cases, and for 3 threads and region size 5, SPIN verified the model in 5 Sec. The time increase in failure cases is negligible. Observe that, increasing the number of threads raises the model checking time exponentially, whereas the region size causes linear increases. This is because increasing the number of threads exponentially increases the number of reachable states due to the combinatorial nature of all possible thread interleavings.

In terms of space complexity, SPIN explored 17.1 million states for the model checking of an instance of the heat flow model with 3 threads and the region size of 11. Nonetheless, for an instance with 4 threads and region size 3, SPIN explored 6.7 million states. The verification of an instance with 5 threads and the region size 3 failed due to insufficient memory.

Model checking of CG. For 2 threads, the time spent for the model checking of assertion violation is negligible. For deadlock-freedom of each thread, SPIN needed 0.015 seconds. The model checking of assertion violations for 3 and 4 threads respectively required 0.031 and 3.17 seconds and SPIN spent 0.4 and 17.5 for the verification of deadlock-freedom for 3 and 4 threads respectively. The model checking of the CG model with 5 threads, was inconclusive due to memory constraints. To verify an instance with 4 threads, SPIN explored almost 3 million states.

9 Conclusions and Future Work

We presented a method (see Figure 1) and a framework, called UPC-SPIN (see Figure 2), for the model checking of UPC programs. The proposed method requires programmers to create a tabled abstraction file that specifies how different UPC constructs should be modeled in the Promela [18], which is the modeling language of the SPIN model checker [3]. Such abstractions are property-dependent in that the abstraction rules depend on the property of interest (e.g., data race-freedom and deadlock-freedom) against which the program should be model checked. If either the program or the property of interest changes, then a new set of abstraction rules should be specified by developers. We presented a set of built-in rules that enable the abstraction of UPC synchronization primitives in Promela. The UPC-SPIN framework includes a front-end compiler, called the UPC Model Extractor (UPC-ModEx), that generates finite models of UPC programs,

and a back-end that uses SPIN to verify models of UPC programs for properties of interest. Using UPC-SPIN, we have verified several real-world UPC programs including parallel bubble sort, heat flow in metal rods, integer permutation and parallel data collection (see [14] for details). Our verification attempts have both mechanically verified the correctness of programs and have also revealed several concurrency failures (i.e., data races and deadlocks/livelocks). For instance, we have detected data races in a program that models heat flow in metal rods (see Section 4.1). The significance of the heat flow example is in the dynamic nature of data races that depend on the run-time read/write accesses of different threads to the memory cells of a shared array. More importantly, we have generated a finite model of a UPC implementation of the Conjugate Gradient (CG) kernel of the NAS Parallel Benchmarks (NPB) [20], and have mechanically demonstrated its correctness for data race-freedom and deadlock-freedom. We have illustrated that even though we verify models of UPC programs with a few threads, it is difficult to *manually* detect the concurrency failures that are detected by the UPC-SPIN framework. Moreover, since SPIN exhaustively checks all reachable states of a model, such failures certainly exists in model instances with larger numbers of threads.

There are several extension to this work. First, we would like to devise abstractions rules for all UPC collectives and implement them in UPC-ModEx, which is the model extractor of UPC-SPIN. Second, to scale up the time/space efficiency of model checking, we are currently working on integrating a swarm platform for model checking [27] in the UPC-SPIN framework so that we can exploit the processing power of computer clusters for the model checking of UPC applications. Third, we plan to design algorithmic methods that can provide suggestions for developers as to how a concurrency failure should be fixed without introducing new design flaws. Last but not least, we believe that a similar approach can be taken to facilitate the model checking of other PGAS languages.

Acknowledgement. The author would like to thank Professor Steve Seidel for his insightful comments about the semantics of UPC constructs and for providing some of the example UPC programs. We also thank Mr. Aly Farahat and Mr. Mohammad Amin Alipour for studying existing model extraction techniques.

References

[1] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[2] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[3] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[4] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[5] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI : portable parallel programming with the message-passing interface*. MIT Press, 1994.

[6] Doruk Bozdag, Assefaw Hadish Gebremedhin, Fredrik Manne, Erik G. Boman, and Ümit V. Çatalyürek. A framework for scalable greedy coloring on distributed-memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.

[7] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A self-stabilizing approximation algorithm for the maximum matching problem. In *10th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 94–108, 2008.

[8] A. Thorsen, P. Merkey, and F. Manne. Maximum weighted matching using the partitioned global address space model. In *HPC '09: Proceedings of the High Performance Computing and Simulation Symposium*, 2009.

[9] Stephen F. Siegel. Verifying parallel programs with MPI-Spin. In *14th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 13–14, 2007.

[10] Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Scheduling considerations for building dynamic verification tools for mpi. In *Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging*, pages 3:1–3:6, 2008.

[11] Anh Vo, Sarvani Vakkalanka, Jason Williams, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. Sound and efficient dynamic verification of mpi programs with probe non-determinism. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 271–281, 2009.

[12] Sarvani S. Vakkalanka, Anh Vo, Ganesh Gopalakrishnan, and Robert M. Kirby. Precise dynamic analysis for slack elasticity: Adding buffering without adding bugs. In *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting (EuroMPI)*, pages 152–159, 2010.

[13] Stephen F. Siegel and Timothy K. Zirkel. Automatic formal verification of mpi-based parallel programs. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 309–310, 2011.

[14] Ali Ebnenasir. UPC-SPIN: A Framework for the Model Checking of UPC Programs. Technical Report CS-TR-11-03, Computer Science Department, Michigan Technological University, Houghton, Michigan, July 2011.

[15] Gerard J. Holzmann and Margaret H. Smith. A practical method for verifying event-driven software. In *International Conference on Software Engineering (ICSE)*, pages 597–607, 1999.

[16] Gerard J. Holzmann. Logic verification of ANSI-C code with SPIN. In *7th International Workshop on SPIN Model Checking and Software Verification*, pages 131–147, 2000.

[17] Gerard J. Holzmann and M.H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.

[18] G. J. Holzmann. Promela language reference. `http://spinroot.com/spin/Man/promela.html`.

[19] E.A. Emerson. *Handbook of Theoretical Computer Science: Chapter 16, Temporal and Modal Logic*. Elsevier Science Publishers B. V., 1990.

[20] UPC NAS Parallel Benchmarks. George Washington University, High-Performance Computing Laboratory. `http://threads.hpcl.gwu.edu/sites/npb-upc`.

[21] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. *SIGARCH Computer Architecture News*, 36(1):329–339, 2008.

[22] Stephen F. Siegel and Louis F. Rossi. Analyzing blobflow: A case study using model checking to verify parallel scientific software. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 274–282, 2008.

[23] UPC Consortium. UPC language specifications v1.2, 2005. Lawrence Berkeley National Lab Tech Report LBNL-59208 (`http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf`).

[24] Tarek A. El-Ghazawi and Lauren Smith. UPC: Unified Parallel C. In *The ACM/IEEE Conference on High Performance Networking and Computing*, page 27, 2006.

[25] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[26] J.W. Backus. The syntax and semantics of the proposed international algebraic language of the zurich ACM-GAMM conference. In *Proceedings of International Conference on Information Processing – UNESCO*, pages 125–132, 1959.

[27] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, 2008.