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

Ali Ebnenasir
Department of Computer Science
Michigan Technological University
Houghton MI 49931, USA
aebnenas@mtu.edu

Abstract
This paper presents a software framework for the model checking of the inter-thread synchronization functionalities 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 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.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Program Verification; D.2.5 [Software Engineering]: Testing and Debugging

General Terms High Performance Computing, Verification

Keywords PGAS, UPC, Model Checking

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\(^1\) 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). This paper presents a novel framework (see Figure 1) for model extraction and 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)\(^9\), 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\(^3\,17,\,23\)). As such, while there are tools for the model checking of MPI applications\(^20\,22\,25\), we would like to enable the model checking of PGAS applications. The PGAS memory model aims at simplifying programming and increasing performance by exploiting data locality in a shared address space.

This paper presents a framework, called UPC-SPIN (see Figure 1), for model extraction and model checking of UPC applications using the SPIN model checker\(^11\), 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. The proposed framework (see Figure 1) requires programmers to manually specify abstraction rules for model extraction in a Look-Up Table (LUT). Such abstraction rules are property-dependent in that for the same program and different properties/requirements (e.g., data race-freedom, deadlock-freedom), we may need to specify different abstraction rules. The abstraction rules specify how relevant UPC constructs are captured in the modeling language of the SPIN model checker\(^11\). After creating a LUT, UPC-SPIN automatically extracts a finite model

\(^{1}\) 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.

\(^{2}\) This work was sponsored by the NSF grant CCF-0950678.
from the source code and model checks the model 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 relatively low overhead.

The proposed framework includes two components (see Figure 1): 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 [12–14] in order to support the UPC grammar. UPC-ModEx takes a UPC program along with a set of abstraction rules (specified as a LUT) and automatically generates a Promela model (Figure 1). Promela [10] 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 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 temporal logic [8] expressions. SPIN verifies whether all executions of the model from its initial states satisfy the specified properties. If the model fails to meet the properties, then UPC-SPIN generates a sequence of program instructions that could lead to the failure from the initial state (Figure 1).

![Figure 1. An overview of the UPC-SPIN framework.](image)

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 2D space (Lines 10-11 of Figure 2, 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 2 demonstrates an integer permutation application that takes an array of distinct integers (see array A in Line 2 of Figure 2) 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*) is declared in Line 3. Each thread initializes A [MYTHREAD] (Lines 10-11) 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 \( p \). A model of \( p \) is a non-deterministic finite state transition system denoted by a triple \( (V_p, \delta_p, I_p) \) 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 \( \delta_p \) denotes the union of the set of transitions of threads of \( p \). We use actions (a.k.a guarded commands) to represent sets of program transitions. An action is of the form \( grd \rightarrow stmt \), where the guard \( grd \) is an expression in terms of model variables and the statement \( stmt \) updates model...
variables. When the guard $\text{grd}$ holds (i.e., the action is enabled), the statement $\text{stmt}$ can be executed, which accordingly updates some variables. Each action captures a set of transitions of a specific thread. $I_p$ represents a set of initial states. The state space of $p$, denoted $S_p$, is equal to the set of all states of $p$. A state predicate is a subset of $S_p$; i.e., defines a function from $S_p$ to $\{\text{true, false}\}$. A state predicate $X$ is true (i.e., holds) in a state $s$ if (and only if) $s \in X$. A computation (i.e., synchronization trace) of $p$ is a maximal sequence $\sigma = (s_0, s_1, \cdots)$ of states $s_i$, where $s_0 \in I_p$ and each transition $(s_i, s_{i+1})$ belongs to an action of some thread; i.e., $(s_i, s_{i+1}) \in \delta_i$ for $i \geq 0$. That is, either $\sigma$ is infinite, or if $\sigma$ is a finite sequence $(s_0, s_1, \cdots, s_f)$, then no thread is enabled at $s_f$, where an enabled thread has at least one enabled action.

2.3 Model Checking, SPIN and Promela

Explicit-state model checkers (e.g., SPIN [11]) 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 [18]) 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 model 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 communication. The semantics of Promela is based on an operational model that defines how the actions of processes are interleaved. Actions can be atomic or non-atomic, where an atomic action (denoted by atomic {}) blocks in Promela ensures that the guard evaluation and the execution of the statement is uninterrupted.

2.4 Concurrency Failures and Properties of Interest

To verify a model using model checkers, developers have to specify safety and liveness 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 [19]. A block of statements accessing shared data is called a critical section of the code, denoted $CS_i$, for thread $i \leq 0 \leq i < \text{THREADS}$; e.g., Lines 19-21 and 29-31 in Figure 2 where threads perform the swapping. A data race could occur when two or more threads are in their critical sections. However, the $\text{upc_unlock}$ 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, denoted $TS_i$, for thread $i$ (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 2 impose a total order on the way lock variables are acquired in order to break circular waits.

In the UPC program of Figure 2, a safety property stipulates that it is always the case that no two threads have access to the same array cell. In SPIN, such properties are formally specified using the always operator in Linear Temporal Logic (LTL) [8], denoted $\Box$. The example UPC code of Figure 2 ensures that the safety property $\Box \left( (\text{main}[i] : s = j) \lor (\text{main}[j] : s = i) \right) \Rightarrow \neg (CS_i \land \neg CS_j)$ is met by acquiring locks ($0 \leq i, j < \text{THREADS}$), where $CS_i$ is a state predicate representing that thread $i$ is in its critical section (i.e., Lines 19-21 or Lines 29-31) and ‘main[i]:s’ denotes the value of the local variable $s$ in thread $i$ created from the proctype ‘main’ in Figure 4.

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 $P \Rightarrow Q$ (read it as ‘$P$ leads to $Q$’) [8]. For example, in the example UPC program of Figure 2, we specify progress for each thread $i$ ($0 \leq i < \text{THREADS}$) as $TS_i \Rightarrow CS_i$; 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).

2.5 ModEx: Model Extractor of ANSI C Programs

Since in Section 3 we extend the front-end compiler of the ANSI C Model Extractor (ModEx) [12–14] to support the UPC grammar, this section presents an overview of ModEx, which is a software tool for 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 modeling 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, function 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 include more information than necessary for model checking, then they can be abstracted as Boolean variables in the model. For example, consider a variable \( 0 \leq \text{temp} \leq 100 \) 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 \( \text{temp} \geq 95 \), a Boolean variable can capture the value of a predicate representing whether or not \( \text{temp} \) 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 \( \text{false} \rightarrow x = 0 \) in Promela can be omitted without any impact on the result of model checking because the guard is always \( \text{false} \) 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. Section 3.3 discusses how we abstract read/write accesses to shared data, and Section 3.4 demonstrates model extraction in the context of the integer permutation program in Figure 2.

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. Due to space constraints, we omit the extension for unary expressions (see [6] for details). The extension for UPC collectives is outside the scope of this paper.

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 [2]:

- \( \text{type} = \text{CONST} \mid \text{VOLATILE} \mid \text{shared}, \text{type} = \text{QUAL} \mid \text{reference}, \text{type} = \text{QUAL} \)
- \( \text{shared}, \text{type} = \text{QUAL} : \text{"shared"} \mid \text{"shared"} \left[ \text{opt_const, expr} \right] \)
- \( \text{"shared"} \left[ * \right] \)

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

- \( \text{reference, type} = \text{QUAL} : \text{"strict"} \mid \text{"strict"} \)

We note that, in this paper, we focus on model checking in the strict consistency model.

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

- \( \text{forall, stmt} : \text{"upc forall"} \left( \text{opt_expr} \mid \text{opt_expr} \mid \text{opt_expr} \right) \)
- \( \text{affinity, expr} : \text{"continue"} \mid \text{opt_expr} \)

The affinity expression \( \text{affinity, expr} \) determines which thread executes which iteration of the loop depending on the affinity of the data objects referred in \( \text{affinity, expr} \). If \( \text{affinity, expr} \) is an integer expression \( \text{expr} \), then each thread executes the body of the loop when \( \text{MYTHREAD} \) is equal to \( \text{expr} \mod \text{THREADS} \). If \( \text{affinity, expr} \) 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 \( \text{upc barrier}, \text{upc wait} \) and \( \text{upc notify} \) statements. Moreover, UPC has a new type \( \text{upc lock} \) that enables programmers to declare lock variables for synchronizing access to shared resources/data. The two functions \( \text{upc lock}() \) and \( \text{upc unlock}() \) are used to acquire and release shared variables of type \( \text{upc lock} \). The grammar of the synchronization statements is as follows:

- \( \text{upc barrier, stmt} : \text{"upc barrier"} \left( \text{opt_expr} \right) \)
- \( \text{upc wait, stmt} : \text{"upc wait"} \left( \text{opt_expr} \right) \)
- \( \text{upc notify, stmt} : \text{"upc notify"} \left( \text{opt_expr} \right) \)

We extend 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. We use ModEx commands [12–14] for the specification of such rules. Each rule is of the form:

\[
\begin{array}{c|c}
\text{left-hand side} & \text{right-hand side} \\
\hline
\text{skip} & \text{Replace with a null operation} \\
\text{hide} & \text{Conceal in the model} \\
\text{keep} & \text{Preserve in the model} \\
\text{Substitute} P_1 \text{~} P_2 & \text{Substitute any occurrence of } P_1 \text{ with } P_2 \\
\text{Import} \text{name scope} & \text{Include name with a scope of } \text{'scope'}
\end{array}
\]

We present the following abstraction rules for model generation from UPC programs (see [6] for more rules):

**Rule 1: upc lock()** The \( \text{upc lock(UPC lock, *lk)} \) function locks a \text{shared} variable of type \( \text{upc lock} \). 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 \( \text{lk} \) atomically. The corresponding Promela code is as follows:

\[
\begin{array}{l}
\text{bool lk; // Global lock variable} \\
\text{atomic{ }
\begin{array}{l}
\text{lk += lk;}
\text{lk++;}
\text{lk++;
\end{array}
\}
\]

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

**Rule 2: upc unlock()** The \( \text{upc unlock(UPC lock, *lk)} \) is translated to an assignment \( \text{lk} = \text{false} \) in Promela. Assignments are executed atomically in Promela.
Rule 3: 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.

Rule 4: 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).

Rule 5: (Split-Phase) 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 6: upc forall  To model the work-sharing iteration statement upc forall in Promela, we first explain how regular for-loops in C are modeled by ModEx. Then, we describe how we extract Promela models from upc forall statements. Consider a C for-loop "for (init; cond; cntr update) stmtBlk;" where init denotes the initialization of the loop counter, the cond represents the termination condition, cntr update updates the loop counter and stmtBlk is the statement block in the loop body. The following Promela statements model such a C for-loop.

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 and the way it is accessed (i.e., via pointers or by name) are irrelevant to verification; rather it is the type of read/write operation on the 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 2, we consider the following bit arrays in its Promela model:

3.4 Example: Promela Model of Integer Permutation

For model extraction, UPC-ModEx needs two input files: the input UPC program and a text file that contains the abstraction LUT. Figure 3 illustrates the LUT for the program in Figure 2:

Figure 3. The abstraction file for the program in Figure 2, 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 contributions. The first line in Figure 3 (i.e., command \%F) specifies the name of the source file from which we want to extract a model. Line 2 (i.e., command \%L) 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[i] = i in the model, where i is set to MYTHREAD. We apply Rule 5 (presented in Section 3.2) for the abstraction of
upc_barrier (Lines 8-11 in Figure 3). Line 14 of Figure 2 (i.e., s = (int)rand48() % (THREADS)) assigns a randomly-selected integer (between 0 and THREADS−1) to variable s. The semantics of Line 14 is captured by a ‘select(v : L.H)’ statement in Promela (e.g., Line 12 in Figure 3), where a random number between L and H (inclusive) is assigned to variable v. The value of the variable v determines the array cell with which the value of A[s] should be swapped by thread i. Lines 13-16 include the rules for the abstraction of upc_lock() and upc_unlock() functions. Lines 17-24 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 2 and the abstraction file of Figure 3, UPC-ModEx generates the Promela model in Figure 4. Lines 1-6 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 suffix [THREADS] specifies the number of instances of the main proctype that are created by SPIN. Lines 11-14 implement upc_barrier. Each proctype randomly assigns a value between 0 and THREADS−1 to variable s (Line 17) and then performs the swapping in either one of the if-statements in Lines 18 or 34. The automatically-generated line numbers that are written as comments associate the instructions in the UPC 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 4 for lack of simultaneous read and write operations, we first determine the conditions under which a shared datum is read and written by multiple threads at the same time. (Section 5 illustrates a case where we verify freedom from simultaneous writes.) Using the abstractions defined for shared data accesses in Section 3.3, we define the following macro for the Promela model of Figure 4:

```c
#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;
    i = _pid; /* line 42 */
    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 */
    select(s: 0 .. THREADS-1);
    if :: (s<i) -> { /* line 54 */
      atomic{ lk[i] = 1; } /* line 59 */
      atomic{ lk[s] = 1; } /* line 60 */
    read_A[s]=1; /* line 62 */
    read_A[i]=0;
    read_A[s]=1; /* line 63 */
    write_A[i]=1;
    write_A[s]=1; /* line 64 */
    write_A[i]=0;
    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[i] = 1; } /* line 70 */
    atomic{ lk[s] = 1; } /* line 71 */
    read_A[i]=1; /* line 73 */
    read_A[s]=1; /* line 74 */
    read_A[i]=0;
    write_A[i]=1;
    write_A[s]=0;
    write_A[i]=0;
    write_A[s]=1; /* line 75 */
    write_A[i]=0;
    lk[i] = 0; /* line 77 */
    lk[s] = 0; /* line 78 */
  } /* line 79 */
  fi;
P: skip; }
```

Figure 4. The Promela model generated for the program in Figure 2.

4.1 Example: Heat Flow

The Heat Flow (HF) program includes THREADS>1 threads and a shared array t of size THREADS×regLen, where regLen > 1 is the length of a region vector accessible to each thread. That is, each thread i (0 ≤ i ≤ THREADS−1) has read/write access to array cells t[i × regLen] up to t[(i + 1) × regLen] − 1. The shared array t captures the transfer of heat in a metal rod and the HF program models the heat flow in the rod. Figure 5 presents an excerpt of the UPC code of HF.

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 ≤ i ≤ 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-
shared double t[regLen*THREADS];
double tmp[2];
double e, etmp;
...
/* Perform some local computations
upc_barrier;
base = MYTHREAD*regLen;
for (j = 0; j < regLen+1; j++) {
  if (MYTHREAD == 0) { tmp[0] = t[0]; }
  else {
    tmp[0] = t[base-1] + t[base] + t[base+1])/3.0;
    e = fabs(t[base] - tmp[0]);
  }
  for (i = base+1; i < base+regLen-1; i++) {
    tmp[i] = (t[i-1] + t[i] + t[i+1]) / 3.0;
    etmp = fabs(t[i] - tmp[i]);
    t[i] = tmp[i];
  }
  if (MYTHREAD < THREADS-1) {
    tmp[base+regLen-1] = 1;
    etmp = fabs(tmp[base+regLen-1] - tmp[1]);
    read_t[base+regLen-1] = 1;
  }
  for (i = base+1; i < base+regLen-1; i++) {
    t[i] = tmp[i];
    etmp = fabs(t[i] - tmp[i]);
    read_t[i] = 1;
  }
  upc_barrier;
  t[base+regLen-2] = tmp[0];
}

Figure 5. Excerpt of the Heat Flow (HF) program in UPC.

23). Before updating \( \text{base} + \text{regLen} - 2 \) in Line 25, all threads synchronize using upc_barrier. Our objective is to verify whether or not there are any simultaneous read-write operations in HF. Since no thread writes in another thread’s region, no simultaneous writes occur. 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).

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.

```cpp
#define regLen 3
#define THREADS 3
int barr = THREADS;
int proc = 0;
int write_t[regLen * THREADS];
bit read_t[regLen * THREADS];
active [THREADS] proctype main() {
  int base; /* mapped */
  int i; /* mapped */
  int j; /* mapped */
  atomic{
    barr = barr - 1; proc=0; /* line 50 */
    if (barr == 0) || (proc == 1) proc = 1;
    barr = barr + 1;
    barr = THREADS || (proc == 0) proc = 0;
    base = _pid * regLen; /* line 55 */
    i = base + 1; /* line 71 */
  }
  if (:: (:: (_pid==0) /* line 60 */
    read_t[0]=1; read_t[0]=0; /* line 64 */
    : : else; /* line 64 */
    read_t[base]=1; read_t[base]=0;
    read_t[base+1]=1; read_t[base+1]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
    read_t[base+regLen-1]=1;
    read_t[base+regLen-2]=1;
    read_t[base+regLen-1]=0;
    read_t[base+regLen-2]=0;
  if (:: (:: (_pid==0) /* line 60 */
    e = fabs(t[base]-tmp[0]));
  if (:: (:: (_pid==0) /* line 60 */
    base = _pid * regLen; /* line 55 */
    if (:: (:: (_pid==0) /* line 60 */
    barr = barr - 1; proc=0; /* line 50 */
    if (barr == 0) || (proc == 1) proc = 1;
    barr = barr + 1;
    barr = THREADS || (proc == 0) proc = 0;
    base = _pid * regLen; /* line 55 */
    i = base + 1; /* line 71 */
```
We verify the reduce operation in Line 2 and sum operation in Line 51 of Figure 5. Another data race occurs when thread 0 is writing its array cell in Line 17 and thread 1 could be reading the same cell in Line 18. Specifically, this counterexample demonstrates that Thread 0 could be reading array cell [base+regLen-2] in Line 17 and Thread 1 could be reading array cell [base+regLen-1] in Line 18. 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 [24]). 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 6 demonstrates the inter-thread synchronization functionalities of CG.

![Figure 6](image-url)

The first three lines in Figure 6 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 6. 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`:

```
typedef bitValStruc { bit b[NUM_PROC_COLS] };
bitStruc write_sh_reduce[THREADS];
```

Lines 1-2 above define a two dimensional bit array (of type `bitValStruc`) 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:
# States
12.4
4
2,681,440

# States
8,081,872
2.14

# Threads
8
7
reduce[rsi].v[0].b[
Time (Sec.)
9
3
3
67,296
# States
4
Time (Sec.)
166,740
37.5
1,838,224
10
3
reduce
0.125
4
960,248
9.53
0.4
pid-rso]] != 1)
1,299,858
reduce
17.5
33,189
4
440,852
25.1
6
20.5
reduce[rsi].v[1].b[...

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 simultaneous writes.

1
:: (rsi < rso + NUM_PROC_COLS) ->
2
assert(write_sh_reduce[rsi].v[0].b[_pid-rso] = 1);
3
write_sh_reduce[rsi].v[0].b[_pid-rso] = 1;
4
write_sh_reduce[rsi].v[0].b[_pid-rso] = 0;
5
:: (rsi < rso + NUM_PROC_COLS) ->
6
assert(write_sh_reduce[rsi].v[1].b[_pid-rso] = 1);
7. Conclusions and Future Work

We presented a framework, called UPC-SPIN (see Figure 1), for the model checking of UPC programs. The proposed framework requires programmers to create a tabled abstraction file that specifies how different UPC constructs should be modeled in Promela [10], which is the modeling language of the SPIN model checker [11]. 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 [6] 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). More importantly, we have generated a finite model of a UPC implementation of the Conjugate Gradient (CG) kernel of the NAS Parallel Benchmarks (NPB) [24], 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 exist 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 [15] 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 investigate the model checking of UPC programs in the relaxed memory consistency model. Last but not least, we believe that a similar approach can be taken to facilitate the model checking of other PGAS languages.

Acknowledgments

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.

References