X10X: Model Checking a New Programming Language with an "Old" Model Checker

Milos Gligoric
University of Illinois
Urbana, IL 61801, USA
gliga@illinois.edu

Peter C. Mehlitz
NASA Ames Research Center
Moffett Field, CA 94035, USA
peter.c.mehlitz@nasa.gov

Darko Marinov
University of Illinois
Urbana, IL 61801, USA
marinov@illinois.edu

***Abstract*—Parallel and distributed computing is becoming a norm with the advent of multi-core, networked, and cloud computing platforms. New programming languages are emerging for these platforms, e.g., the X10 language from IBM. While these languages explicitly support concurrent programming, they cannot eliminate all concurrency related bugs, which are usually hard to find and analyze. This requires specialized tools, including model checking, which needs to be language-aware to facilitate adoption.**

We address the following question: How to create a high quality systematic testing (or model checking) tool for a new language, such as X10, in a reasonable amount of time? Development from scratch is not an option since it can require the same effort as the language itself. Thus, our approach is to start with an existing tool and change it only as necessary. Specifically, we have a readily available model checking tool for Java, JPF, and X10 programs can be compiled to Java. Unfortunately, checking X10 programs with unmodified JPF and X10 runtime can miss some behaviors and scales poorly. This paper describes four sets of techniques that can be employed to make checking "new" language with an "old" model checker more practical: (1) modify the model checker, (2) modify the language runtime, (3) extend the language compiler, and (4) develop a new static analysis. We instantiated each technique to enable checking X10 programs with JPF. New techniques were evaluated on over 100 X10 example programs and provide a substantial speedup.

I. Introduction

The widespread availability of high-speed networks and multi-core processors has recently spawned development of several new general-purpose programming languages (or significant extensions of the existing languages) for parallel and distributed programming, e.g., to name a few from industry: Chapel from Cray [12], DryadLINQ from Microsoft [32], Fortress from Oracle/Sun [3], Pig from Yahoo! [15], TBB from Intel [1], and X10 from IBM [9], [26]. While these new languages make development of more reliable parallel code easier, such development still remains error-prone, with potential concurrency bugs including atomicity violations, dataraces, and deadlocks, or traditional semantic bugs in sequential portions of the code. A common way to check for such bugs is to perform systematic testing (or explicit-state model checking [10]) that explores code behavior for various interleavings of shared-memory accesses.

This paper focuses on the X10 programming language being developed at the IBM Research in collaboration with academic partners [9], [26]. X10 is a modern language designed for high-productivity and high-performance computing on high-end platforms [31], its name reflecting the goal of increasing both productivity and efficiency of parallel programming by an order of magnitude. X10 is a statically typed, object-oriented language that provides several abstractions for concurrency, distribution, and locality (as described later in the text). The current X10 version provides compilers from X10 to Java and C++, as well as runtime systems that enable execution of the compiled programs.

We address the following question: How can we create a high quality model checking tool for a new language, such as X10, in a reasonable amount of time? One option is to translate the input language into another language for an existing tool. For example, Java PathFinder (JPF) [21], [29] is a popular, open-source model checker for Java programs that showed promising results, e.g., for checking NASA software [24]. The very first version of JPF [16], [17] checked Java code by translating it into the Promela language and using the Spin model checker [18]. While such a translation can be developed relatively quickly, it needs to bridge a potentially large semantic gap between greatly different languages and may not support all the features of the input language.

Another option is to build a new model checking tool from scratch. For instance, the second version of JPF was developed (in Java) from scratch as a backtrackable Java Virtual Machine (JVM) that can directly explore programs compiled to Java bytecode [21], [29]. However, developing from scratch a high quality tool, which can explore a large subset of the input language and scales well to non-trivial code, can take a long time, e.g., JPF development has taken over 20 man-years, and the JPF core alone has over 100KLOC, with property extensions of about the same size.

Yet another option, possible when the input language already translates into the language used by a tool, is to directly use an existing model checking tool. Specifically, X10 can be compiled into Java, and JPF can check Java code. Can we simply use JPF to check X10 programs?

Unfortunately, no: checking X10 programs directly with the unmodified JPF and X10 runtime has two problems. First, it can miss some executions (and hence miss bugs). Second, it scales very poorly.

There has been a long line of research on how to adapt an existing model checker for a new language (or an extension of an existing language) [5], [8], [11], [13], [17], [19], [25], e.g., we have recently used JPF for checking a *subset* of programs written in the Scala programming language [22]. However, that work focused *only on the second problem*—making the checking faster—and provided *only one solution technique*—modifying the language runtime. In *contrast*, this paper analyzes *both problems*, provides *four sets of solution techniques*, and considers the *entire X10 language*.

This paper makes the following contributions.

Complete Exploration: We identify the problem that *model checking programs compiled from one language to another can miss some behaviors*. Specifically, running X10 programs compiled to Java on the unmodified JPF and X10 runtime is incomplete. The key reason is that the X10 runtime is designed for *fast execution* of (parallel) programs and uses sophisticated techniques such as work stealing [7]. However, such techniques *prevent exploration* of all possible interleavings of shared memory accesses. We also found that JPF did not support some Java constructs used by the X10 runtime. We describe changes to the X10 runtime and JPF that enable complete exploration of X10 programs on JPF.

Faster Exploration: After ensuring complete exploration, we focus on speeding up the exploration by developing techniques (related to partial-order reduction [10]) that prune some executions equivalent to others. As in related previous work [5], [8], [11], [13], [16], [17], [19], [22], [25], the goal is to explore executions induced by the semantics of the input language and not by the language runtime. Whereas the previous work achieved speedup only by modifying the language runtime, we recognized *four sets of techniques*: (1) changing the model checker, (2) customizing the runtime for execution on the model checker, (3) modifying the compiler, and (4) introducing a novel static analysis to further optimize the checking. We modified JPF, X10 runtime, X10-to-Java compiler and introduced a novel static analysis.

Implementation: We implemented our techniques as a JPF extension called *X10X* (from "X10 eXplorer"). We supported all the constructs from the X10 language and enabled their exploration on top of JPF, while avoiding non-determinism introduced by the X10 runtime system.

Evaluation: We evaluated X10X on a large set of X10 example programs. 65 of these programs include *all* the example programs from the X10 compiler distribution that focus on parallel language constructs. 31 programs we wrote ourselves to understand the X10 language constructs. 11 programs are larger programs from the X10 distribution, and 3 programs are our larger programs. The results show that our techniques provide over an order of magnitude speedup.

II. Example

In this section we introduce the programming model and key constructs of the X10 language[1] through a running example that is used in the following sections to illustrate our techniques for complete and faster exploration. Our example computes a histogram of a given array, i.e., counts how many elements of the array fall in various bins, where each bin is simply defined via a modulo operation. This example comes from the publicly available X10 distribution [30], but we modified the provided sequential version to explicitly illustrate X10 parallel constructs. Before we discuss the example in more detail, we provide a brief, high-level overview of X10. We finally discuss how our changes affect the exploration of the example on JPF.

A. X10 Background

X10 is an object-oriented language designed for programming multi-cores, GPUs, and clusters. X10 follows the partitioned global address space (PGAS) programming model [31], where computation entities share a single global memory space logically partitioned into *places*. Places can either be distributed over a number of machines or reside on one machine, which depends on the platform used for the execution of a program. Every object (or a part of the object, in case of a distributed object) resides during the entire program execution in place where it is created.

Sequential constructs of X10 are similar to those of Java with some extensions such as type inference, anonymous functions, and structs. Due to the X10's focus on high-performance computing, the main difference between X10 and Java is related to arrays. Namely, Java-like arrays are not available in X10 but instead a special set of classes is used to replace arrays, e.g., the class `Array` where all elements reside at one place, or the class `DistArray` where the elements are distributed over a number of places. These classes include many methods used for manipulation of array objects, e.g., for creating arrays with a given distribution, finding distribution of a given array, iterating over all indexes in an array (which need not be consecutive integers), etc.

Parallel constructs of X10 are quite different from Java. X10 has no Java-like threads, but its basic entity of parallel execution is an *activity*, which can be viewed as an anonymous, lightweight thread. Each activity executes at some place. Creating a new activity to execute a statement `S` (at the place of the current activity) in parallel with the other activities is done simply with `async S`. An activity can access a local object at the same place directly, but accessing a remote object at a different place requires explicitly specifying the remote place, e.g., with the `at (p) S` construct that executes the statement `S` at the place `p`. Other parallel constructs in X10 include collecting finish, `finish S`, which blocks the current activity until all activities

[1] Specifically X10 version 2.0.x.

```
1  public static def histogram(a: DistArray[Int], S: Int) {
2    val results = DistArray.make[Array[Int]](a.dist.places());
3
4    finish ateach ((p) in results) {
5      val t = new Array[Int](S);
6      for ((i) in (a.dist | p)) { async {
7        val bin = a(i) % S;
8        atomic t(bin)++;
9      } }
10     results(p) = t;
11   }
12
13   val sum = (a1: Array[Int], a2: Array[Int]) => {
14     val s = new Array[Int](a1.length);
15     for ((i) in a1) s(i) = (at(a1) a1(i)) + (at(a2) a2(i));
16     return s;
17   };
18   return results.reduce(sum, new Array[Int](S));
19 }
```

Figure 1. Distributed calculation of histogram

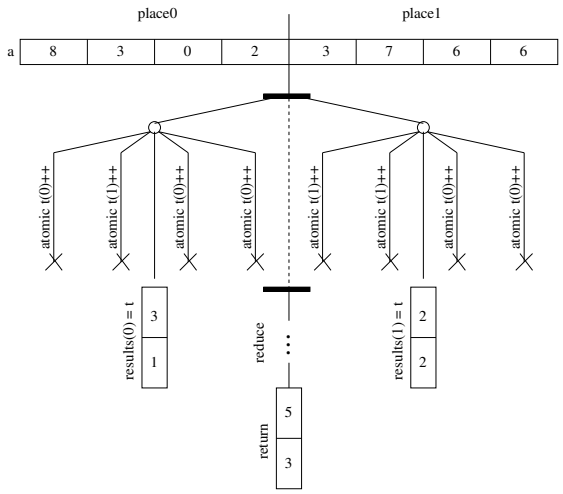

Figure 2. Example histogram execution with 2 places, array of length 8, and bin size 2

spawned directly or indirectly in `S` finish; atomic block, `atomic S`, which executes the statement `S` atomically; and clocks which represent a generalization of barriers.

B. Example Code and Execution

Figure 1 shows the code for our running example that computes histogram for a given distributed array `a` and the size of bins `S`. For instance, if `S` is 2, the code effectively computes how many array elements are even and how many are odd. The `histogram` method has three parts. First, it creates (line 2) an array `results` distributed at the same places where the elements of `a` are distributed. Second, it collects the partial results at each place (lines 4-11). Third, it combines these partial results (lines 13-18).

The second part starts with the `ateach` statement (line 4) that spawns a new activity at each place `p` where `results`/`a` reside. (Putting the loop variable `p`/`i` within `ateach`/`for` in parentheses means that the variable ranges over the array indexes rather than the array values.) Because of `finish` (line 4), the main activity blocks until all the activities created by `ateach` finish. Each such activity computes the partial result for a place `p` in a temporary array `t` that is created with `S` elements, by default all initialized to zero (line 5). For each index `i` of the (sub)array `a` at the place `p` (the operator '`|`' effectively projects the array distribution onto the given place), the code spawns an activity (at the place `p`) that computes the `bin` value for the array element `a(i)` and atomically increments the appropriate value in `t`. In the end, there can be as many parallel activities executing as there are array elements in `a`.[2]

The third part uses the library method `reduce` (from the class `DistArray`) to combine the partial results from the distributed array `results`.[3] The method applies the given reduction operator, in our example the function `sum`, that performs the pairwise addition of elements of arrays `a1` and `a2`. These arrays are from different places where the array `results` was distributed, and thus `sum` uses the `at` construct to access the potentially remote elements (line 15). Note that the `reduce` library method will perform the pairwise addition of all arrays in `results` until it computes only one array of size `S` that is returned as the result of `histogram`.

Figure 2 visualizes an example execution of `histogram` for an input array `a` with eight (random) elements distributed over two places and for `S` being two. Each vertical line represents execution of an activity. After the second part, the `results` distributed array holds the number of even/odd elements at the two places, and in the third part, these numbers are summed up into the final result.

The X10(-to-Java) compiler translates the input X10 code into the appropriate Java code suitable for execution with the X10 runtime. Of particular interest is the translation for activities (created explicitly with `async` or implicitly within `ateach`/`foreach` and even `at` constructs or within the library methods such as `reduce`). X10 translates them into lightweight tasks (using the Fork/Join framework available since Java 7) executed using a pool of worker threads. While this translation allows for *efficient execution* of X10 programs, it creates *problems for exploration* (in JPF or any other model checking tool for Java).

C. Exploration Issues

The first problem is that the use of the thread pool in the X10 runtime (or any other similar runtime) can hide some parallelism. Our example execution has eight activities for array elements, and consider the limit case where the runtime uses only one worker thread for these activities. Using only one thread effectively serializes the execution of

[2]X10 provides syntactic sugar for some idioms we made explicit for the sake of example, e.g., `for (...) async S` can be expressed as `foreach (...) S`.

[3]The second part is similar to the scatter phase in `reduce`, but we made it explicit for the sake of example.

all these activities. If the code had a bug such as omitting the keyword `atomic` (line 8), the bug would not be found because the runtime would execute all activities atomically. Hence, to enable JPF to explore all possible executions of X10 programs, we had to change the X10 runtime to create at least as many threads as there are activities during the execution. Section III presents the related changes in detail.

The second problem is that the use of the thread pool and other complexities of the X10 runtime make exploration of X10 programs in JPF much costlier than it needs to be. Namely, our goal is to check the X10 programs themselves, but JPF explores Java, including the non-determinism from the X10 runtime. For instance, our running example initially timed out after two hours on JPF. However, we can significantly reduce this time by customizing the X10 runtime for exploration (e.g., creating a bijective correspondence between activities and threads), changing JPF itself (e.g., how it handles starting and joining for Java threads), changing the X10 compiler (e.g., to translate `ateach` differently into Java), and performing some static analysis (e.g., to find that certain activities are *place-local* and cannot interact with activities at different places). With all our optimizations, JPF can explore this same running example in just 35 seconds. Section IV presents our optimizations in detail.

III. Complete Exploration

As illustrated, running the unmodified X10 runtime on the unmodified JPF model checker could miss some executions. This section discusses the changes to the X10 runtime and JPF that are necessary to enable complete exploration.

A. Changes to Runtime

The X10 runtime is designed for efficient execution of X10 programs and not for exploration. In particular, creating a (heavyweight) Java thread for each (lightweight) X10 activity would not be good for execution because Java (kernel) threads require significant execution time and memory overhead (e.g. for initialization and stack allocation). Instead, the X10 runtime uses a *pool* of Java threads that it creates only once and then assigns activities to be executed by these threads. When the same thread executes two or more activities, it executes them serially one after another, without interleaving the shared memory accesses from these activities. This happens on both JVM and (unmodified) JPF. While this provides one possible execution of X10 programs, it does not provide all possible executions. Even if we exhaustively tested an X10 program for one pool size and found no bugs, it could have bugs for a different (larger) pool size that has more interleavings among activities.

It is necessary to change the X10 runtime and/or JPF to be able to explore all interleavings. By default, JPF explores (all) interleavings of accesses from different Java threads. Specializing JPF for X10 activities (such that JPF explores interleavings from activities even when they are executed on the same thread) could be done but is not a general solution. Therefore, we opted to change the X10 runtime. Note that it suffices to have *at least* as many Java threads in the pool as the largest number of X10 activities that can execute in parallel. A changed runtime can dynamically track the number of activities and increase the pool size as necessary. However, to make the exploration faster, we modified the X10 runtime to create *exactly* as many Java threads as there are X10 activities and to assign each activity to exactly one thread. Section IV-C discusses this further.

B. Changes to Model Checker

JPF had several small deficiencies that prevented it from executing X10 programs with either the original or the modified X10 runtime. We changed JPF to remove these deficiencies, and while these changes are more engineering than research oriented, they still support our point that building a high-quality model checking tool from scratch takes very long time. Specifically, our changes to JPF can be split into two groups. First, we added so called *model methods* for several methods from the standard Java API or from the JVM internal classes such as `sun.misc.Unsafe`. JPF can interpret Java bytecodes, but it cannot execute methods that are "native", i.e., consists of machine code executed outside of the JVM. Such methods have to be modeled in JPF to ensure equivalent behavior and allow backtracking. JPF already has models for a large part of the standard Java library, but we had to support some more methods used by the X10 runtime. Second, we found and fixed four bugs in JPF. Our changes are already included in the publicly available JPF distribution [21].

IV. Faster Exploration

Our goal is to check X10 programs themselves, at the level of the X10 language semantics, and not for any particular X10 runtime. However, checking X10 programs compiled to Java (together with the X10 runtime) on JPF actually explores these programs at the level of Java, including the non-determinism from the X10 runtime. For example, the original X10 runtime maintains a queue of activities to be scheduled and a pool of worker threads, and the threads can "steal" these activities to execute [7]. Without any changes, JPF would explore various assignments in which these threads can execute activities, e.g., for two threads `t1` and `t2` and two activities `a1` and `a2`, one assignment is that `t1` executes `a1` and `t2` executes `a2`, and another assignment is that `t1` executes `a2` and `t2` executes `a1`. However, at the level of X10 these are equivalent executions.

We made a number of changes to explore only the executions required by the X10 semantics. These changes are at several levels of the X10 and JPF systems: a static analysis of X10 programs (Section IV-A), compiler from X10 to Java (Section IV-B), X10 runtime (Section IV-C), and JPF itself (Section IV-D).

A. Novel Static Analysis

We first describe how JPF uses a dynamic analysis of Java objects to speed up its exploration of Java programs. We then describe a new static analysis that we developed for X10 programs and how to use the results of this analysis to speed up exploration of X10 programs.

JPF performs a *dynamic* reachability analysis that infers which *Java objects are thread-local* [27], [29], i.e., although all Java objects reside on the system global heap, most objects are only referenced from within the threads that created them. JPF uses the information about thread-local objects to prune the exploration of Java programs (effectively performing a partial-order reduction [10] that reduces the state space without missing any behavior). When a program reads or writes a field of a shared object, JPF needs to schedule all runnable threads to explore potential dataraces. Such rescheduling is not required if the object is thread-local. This optimization provides a substantially faster exploration of Java programs in JPF [29].

We developed a *static* analysis that infers which *X10 activities are place-local*, i.e., which X10 activities access only (X10) objects allocated at the same place. The X10 programming model makes each object place-local, i.e., each object resides at one place, where it is created, during the entire program execution.[4] It is possible to have at one place a remote reference to an object that resides at another place, but no direct access to any (non-global) field of the remote object is allowed. (The X10 system guarantees this mostly statically through a type system that includes information about places where objects reside, with some dynamic checks and type casts for objects at unknown places.) The only way to access a (non-global) field of the remote object is indirect: to spawn an activity with `async (P)` or execute the `at (P)` statement/expression at the place `P` where the remote object resides. (The X10 compiler translates '`at (P)`' into (several) remote '`async (P)`' activities.) Thus, X10 makes communication among places explicit.

Our static analysis infers which of the activities (either explicitly written with `async` or desugared by the compiler from other statements/expressions) do not communicate with remote places. The current implementation performs a simple intraprocedural analysis[5] to find whether an activity body spawns another activity at a different place. We implemented our static analysis as a pass in the X10 compiler and added a boolean `isPlaceLocal` flag to the (Java) object representing the activity. Our modified X10-to-Java compiler forwards the analysis results to the X10 runtime and eventually to JPF by changing the activity creation to appropriately set the flag.

[4]More precisely, some objects can be distributed, such as the `DistArray` objects `a` and `results` in the example from Figure 1, but in that case each object *part* resides at the same place during the entire program execution.

[5]If an activity has any method call, we conservatively label it as not place-local.

To illustrate our analysis on an example, recall the code from Figure 1. The activities created in line 6 are place-local as they spawn no new remote activities and access only local objects. More precisely, the activities access only local parts of distributed objects, and this would hold even if `t(bin)` were replaced with `results(p)(bin)`.

We modified JPF to use the information about place-local activities to prune exploration. Effectively, our analysis enables us to generalize JPF's pruning for (one-)thread-local objects to pruning for set-of-thread-local objects. The X10 compiler maps X10 objects and activities into Java objects and threads, respectively. By default, accesses to non-thread-local Java objects would make JPF schedule *all* runnable threads. However, if a place-local activity accesses an object, it suffices to schedule only those threads/activities executing at the same place, i.e., the remote activities need not be scheduled. We modified JPF such that for accesses to non-thread-local objects, JPF uses the `isPlaceLocal` flag, the place of the object, and the place of the activity to decide whether or not to schedule a thread/activity.

B. Changes to Compiler

In addition to changing the X10 compiler to support our static analysis, we made another change for the `ateach` construct to enable faster exploration. The change for `ateach` shows the trade-off between fast *execution* and fast *exploration*. The X10 system developers built the existing X10 compiler and runtime to optimize execution speed, which in many cases results in slower exploration.

After we observed that exploration of our simple examples with the `ateach` construct was taking a significant amount of time (Section V), we inspected the compiled version of the programs and noticed that it creates many activities. The statement `ateach (p in D) S`, where `D` is a distribution of a number of points across places (e.g., distribution of the indexes of a distributed array over places as in our example in Figure 1), executes the statement `S` for every point in the distribution at the place of that point. For instance, in our running example (line 4), `ateach` is used to compute the local histogram at each place `p` of the array.

The X10 language report [26] actually discusses two ways to implement `ateach` and the effect that they have on efficient execution. One way creates more activities but less communication among them, and the other way creates fewer activities but more communication among them. While the report correctly argues that the former is preferred for execution (where communication is more costly than creating activities), the latter is preferred for exploration (as it creates fewer activities/threads and hence fewer scheduling choices and a smaller search space). We added a compiler option to use the latter, which provided a substantial improvement in exploration time (Section V).

C. Changes to Runtime

Similarly to changing the X10 compiler, we modified the X10 runtime to favor exploration over execution speed. We introduced a one-to-one mapping between X10 activities and Java threads (Section IV-C1). We also realized that exploration is sub-optimal for some X10 language constructs because the runtime creates many activities that are not required for the correctness but only for the performance of execution. We modified the implementation of these constructs, specifically `at` and *clocks*, to create fewer activities while preserving their semantics (Section IV-C2).

1) One Thread Per Activity: As discussed previously, the X10 runtime uses lightweight tasks and Java thread pools to execute X10 activities, which avoids the high cost of creating Java threads in JVM and allows executing several activities on the same thread. However, the relative cost of creating Java threads in JPF is moderate since JPF is an interpreter and uses its own thread representation with its own scheduler. The cost of reusing the same thread for several activities is actually higher, since JPF would either miss some executions if there are not enough threads (Section III-A), or it would systematically explore various assignments of activities to threads although those assignments are equivalent.

We modified the X10 runtime to create a new Java thread for each activity when executing on JPF. Each such thread executes only one activity and then gets garbage collected. This one-to-one mapping ensures that the exploration considers both *all executions* allowed by the X10 language semantics and also *only executions* due to the X10 language semantics and not due to the specific X10 runtime implementation. We kept our change to the X10 code as small as possible because our goal is to make the change easy to port when the X10 runtime evolves. For example, we retained the queue of activities that each thread maintains, but the queue always has size one. This change allows us to directly reuse the state comparison (symmetry reduction [10]) that JPF performs to prune the exploration when it encounters equivalent states.

2) Single Memory Space: We first explain how X10 programs that are translated to Java execute in a single memory space within one JVM (be it a regular JVM, such as OpenJDK `java`, or a specialized backtrackable JVM, such as JPF) even if they have several (logical) places. We then describe how we optimized the execution and exploration of two X10 language constructs for the case when they execute in a single JVM.

As of this writing, there is no intermediate language or a special virtual machine for X10, but the X10 system developers decided to compile X10 programs to existing languages, in particular C++ and Java. The Java runtime, which we used, supports only execution on one machine, more precisely on one JVM. While executing on one machine precludes speedup obtained from several machines, executing on one machine is well suited for exploration. In fact, the JPF core checks only non-distributed Java programs, so had the X10 Java runtime been distributed, we would have needed to modify the runtime or to use the JPF extensions for distributed computing [4], [6].

We leveraged the fact that X10 executes on one JVM (and thus all objects are directly accessible via references in the same memory space) to allow one activity/thread executing at one (logical) place to directly execute the code at another (logical) place. Note that the optimizations that we introduced for `at` and clocks are more general than just for exploration; in other words, these optimizations also make the execution faster in runtime that uses singe JVM.

At: The `at (p)` statement/expression executes a piece of code at a place `p`, (potentially) different from the current place where the current activity is executing. Unlike `async` that spawns a new parallel activity and immediately continues execution of the current activity, `at` is synchronous and blocks the current activity until the piece of code finishes its execution. For instance, our running example (line 15) uses `at` to access array elements that may reside at a different place. A general implementation of `at` spawns two new activities: one at the place `p` to execute the given statement/expressions and another at the current place to receive the result (or collect exceptions). However, creating new activities every time is expensive for both execution and exploration; for exploration, activities create not only additional threads that should be scheduled but also unnecessary interleavings of the current activity (until it gets blocked) and the newly spawned activities.

We developed two solutions in the runtime to optimize the activities that `at` spawns. Ideally, we would like to omit those activities completely. The first solution indeed executes the remote statement/expression with the current activity and does not create any new activity. Executing within one activity the code at different places is possible because everything is executed in the single memory space, on one JVM. However, due to the current design of the X10 runtime (and our goal to keep changes relatively small), it is possible to use this solution only if the remote statement does not spawn new activities itself. The second solution is to introduce a new activity, which we call a *friend activity*, to execute the remote statement, but to disable the activities that have friends from being scheduled. To control this scheduling, we added a new, friend-aware scheduler in JPF.

Clocks: The X10 library class `Clock` represents a generalization of a barrier. Any number of activities, which can be at different places, can be registered at the same clock. When an activity registered on some clock reaches a certain execution point (specified by the `next` statement), it blocks until all activities registered on the same clock reach their corresponding execution points, after which they all proceed. More details are available in the X10 report [26]. The key issue is that each clock, as any other X10 object,

resides at one place (where it is created), but all activities, using the method calls from the `Clock` class, update the state of that one object. These methods are small (e.g., incrementing a counter), so it is not necessary to spawn new activities to perform them. Instead, as for `at`, the same current activity can be used to execute the code at a different place (since the current activity would be blocked anyhow).

For the `at` language construct, all the changes were in the X10 runtime (i.e., in the Java code that implements the `at` construct), but for the `Clock` library class, we also want to change the X10 code for the library. However, in X10 we cannot simply use the same activity to execute the code at another place because it gives a compiler error (as it indeed should give). Thus, we slightly modified the X10 compiler to accept our annotations about the X10 code and to insert special method calls for the runtime, after passing all compiler checks.

D. Changes to Model Checker

We also modified JPF itself to enable faster exploration of X10 programs. We focused on the Java constructs that are heavily used by X10: thread operations (Section IV-D1) and atomic sections (Section IV-D2).

1) Thread Operations: Since our modified X10 runtime uses one Java thread per X10 activity, and since X10 programs can use a large number of activities, exploration of X10 programs can execute a large number of thread operations, including `start` that initiates thread execution and `join` that waits for a thread to complete execution. While JPF had implementations for these operations, our careful analysis showed that they created many unnecessary interleavings. In particular, `start` created a scheduling choice, while `join` was implemented as a synchronized method with a busy-wait loop. More generally, while JPF employs an on-the-fly partial-order reduction [10], [29], it is not very aggressive. As a result, programs with many runnable threads can end up not only with a large number of states but also encountering many transitions leading into matched states, which is very detrimental to exploration time, especially when these transitions are expensive (e.g., in X10 when processing large arrays). The `join` operation was an example for this since it had no thread-external state change in the loop (between the synchronization and the wait choice point).

We added to JPF an option to not treat `start` as a scheduling choice. Rather, when a new thread is started, the current thread can keep executing until it accesses some shared data (or sync) variable. This is a sound optimization that can be always used (in fact, `start` is a typical "left mover", which makes it safe to not break the transition [23]). Our optimization is already included in the publicly available JPF [21]. We also changed `join` to be a non-synchronized method that is native to JPF.

We also added a special support for the common `start/join` pattern where one new thread is started and then all the threads that are (recursively) started from the new thread (and its descendants) need to eventually join together. This pattern also provides support for the X10 construct `finish S` that blocks the current activity until all activities created (recursively) by the execution of the statement `S` finish.

Our optimizations have also inspired other changes in the current JPF version 6, such as detection of shared objects by precise thread access tracking instead of conservative reachability analysis.

2) Atomic Sections: Most changes described so far are sound because they either add more executions to be explored or prune from exploration some executions that are provably equivalent to others and thus cannot miss any behavior. The last change that we describe can miss some behavior but also can provide a substantial speedup for exploration. This change considers the `atomic` keyword from X10. For instance, our running example (line 8) atomically increments an array element.

While executing atomic sections, JPF may encounter accesses to some shared (non-thread-local) objects. As discussed previously (Section IV-A), for such accesses JPF by default schedules all runnable threads for execution. However, it is not necessary to schedule these threads if all shared accesses are properly protected by atomic sections, i.e., if there can be no concurrent access to a shared field outside of atomic sections while the shared field is accessed in an atomic section. In such cases, the current thread can simply continue its execution.

We added to JPF an optimization to not schedule threads for shared accesses in X10 atomic sections. These sections are easy to recognize as the X10 compiler translates them into special method calls in the X10 runtime. Moreover, JPF already has code for tracking which locks protect which objects. (That code worked only with the traditional Java locks present since Java 1.0, but we also added support for the `java.util.concurrent` [28] locks that are present since Java 5 and used in the X10 runtime.) While this optimization prunes exploration, it can miss some bugs when there are improperly protected shared accesses. However, in many such cases, JPF can issue a warning, effectively that certain accesses were not considered not shared and have not been explored, but they do appear to be shared.

V. Evaluation

We implemented our changes to JPF in an extension called *X10X* ("X10 eXplorer"). X10X also includes our changes to the X10 system (compiler and runtime). Our changes are controlled via command-line options such that one can run the original or changed versions.

We evaluated X10X on a large set of over 100 X10 example programs. We used such an extensive evaluation

program	Time (h:mm:ss)		Speedup	Memory (MB)		#Transitions		#States	
	Basic	Opt.	(x)	Basic	Opt.	Basic	Opt.	Basic	Opt.
Async/AsyncFieldAccess	1:23:57	0:00:25	201.48	444	167	100248	17	47032	18
Async/AsyncReturn	0:00:45	0:00:11	4.09	248	98	554	7	280	8
Async/AsyncTest2	>2:00:00	0:01:49	>66.06	-	260	-	131	-	83
Async/AsyncTest5	>2:00:00	0:06:50	>17.56	-	360	-	398	-	329
Async/ClockAsyncTest2	>2:00:00	0:00:11	>654.55	-	98	-	8	-	9
Async/AsyncNext	0:03:33	0:00:18	11.83	420	138	4376	11	1848	12
Async/AsyncTest3	1:37:46	0:00:25	234.64	428	163	48336	15	23247	16
Async/ClockAsyncTest	0:00:13	0:00:11	1.18	137	98	19	3	20	4
Finish/FinishTest1	0:00:46	0:00:11	4.18	247	99	559	7	274	8
Finish/FinishTest2	0:02:02	0:00:12	10.17	386	136	4224	27	1681	23
Future/Future0	0:08:20	0:00:11	45.45	402	108	18695	19	8577	19
Future/Future0a	0:08:20	0:00:11	45.45	404	107	18695	19	8577	19
Future/Future1	0:08:20	0:00:11	45.45	407	108	18695	19	8577	19
Future/Future1a	0:08:16	0:00:11	45.09	412	108	18695	19	8577	19
Future/FutureForce	>2:00:00	0:00:28	>257.14	-	225	-	159	-	119
Future/FutureForced	0:31:21	0:00:11	171.00	418	107	94302	23	42167	23
Future/FutureTest2	0:07:59	0:00:12	39.92	408	108	18695	19	8577	19
Future/FutureTest5	>2:00:00	0:42:41	>2.81	-	585	-	9184	-	5776
AtEach/AtEach	>2:00:00	0:00:30	>240.00	-	169	-	72	-	64
AtEach/AtEach2	>2:00:00	0:00:21	>342.86	-	166	-	42	-	34
AtEach/AtEachLoopOnArray	>2:00:00	0:04:35	>26.18	-	342	-	11274	-	2058
For/ForLoop	0:00:28	0:00:08	3.50	189	98	228	5	110	6
For/ForLoop2	0:00:27	0:00:08	3.38	183	98	228	5	110	6
For/ForLoop3	0:00:27	0:00:08	3.38	183	98	228	5	110	6
For/ForLoop4	0:00:27	0:00:08	3.38	183	98	228	5	110	6
For/ForLoopOnArray	0:00:26	0:00:08	3.25	190	98	228	5	110	6
Place/AtCheck	0:00:26	0:00:08	3.25	181	98	228	5	110	6
Place/AtCheck2	0:00:25	0:00:08	3.13	176	98	228	5	110	6
Place/AtThisIntoAtHere	0:00:27	0:00:08	3.38	175	98	228	5	110	6
Place/CheckThisTypeInCall	0:00:26	0:00:08	3.25	175	98	228	5	110	6
Place/FieldReceiverIsExpr	0:00:26	0:00:08	3.25	175	98	228	5	110	6
Place/FieldWrite	0:00:26	0:00:08	3.25	176	98	228	5	110	6
Place/FutureFieldAccessStruct	0:11:03	0:00:13	51.00	417	138	19173	11	8810	12
Place/FutureGlobalMethodInvoke	0:10:59	0:00:12	54.92	417	139	19173	11	8810	12
Place/FutureGlobal...Rev	>2:00:00	0:00:19	>378.95	-	164	-	17	-	18
Place/FutureGlobal...Static	0:11:51	0:00:13	54.69	405	138	19173	11	8810	12
Place/FutureGlobal...Struct	0:11:09	0:00:13	51.46	417	139	19173	11	8810	12
Place/FuturePropertyAccess	0:10:54	0:00:13	50.31	422	138	19173	11	8810	12
Place/FutureProperty...Rev	>2:00:00	0:00:19	>378.95	-	163	-	17	-	18
Place/FutureProperty...Static	0:11:16	0:00:12	56.33	401	139	19173	11	8810	12
Place/GlobalAccess	0:11:10	0:00:12	55.83	415	139	19173	11	8810	12
Place/PlaceCheckArray	>2:00:00	0:00:18	>400.00	-	166	-	17	-	18
Distribution/BlockDist	0:00:32	0:00:08	4.00	219	98	256	5	126	6
Distribution/BlockDist2	0:06:12	0:00:55	6.76	621	420	228	5	110	6
Distribution/BlockDist...Set	0:00:26	0:00:08	3.25	187	98	228	5	110	6
Distribution/ConstDist	0:31:00	0:00:09	206.67	419	99	42973	13	19489	14
Distribution/DistBounds1D	>2:00:00	0:00:26	>276.92	-	161	-	23	-	24
Distribution/DistBounds2D	>2:00:00	0:00:44	>163.64	-	243	-	75	-	75
Distribution/DistBounds3D	0:02:04	0:00:09	13.78	363	99	4910	9	2281	9
Distribution/DistributionTest	0:00:26	0:00:08	3.25	185	99	234	5	116	5
Distribution/DistributionTest1	0:00:26	0:00:08	3.25	183	98	234	5	116	5
Distribution/Restrict	0:00:27	0:00:09	3.00	182	98	228	5	110	6
Atomic/Atomic2	0:00:49	0:00:11	4.45	297	97	632	7	316	8
Atomic/AtomicMethodTest	0:00:37	0:00:09	4.11	271	99	981	20	481	17
Atomic/AtomicOrdered	0:02:25	0:00:16	9.06	395	174	1586	26	746	23
Atomic/AtomicReturn	0:00:25	0:00:08	3.13	176	99	234	5	116	5
Atomic/AtomicTest	0:00:36	0:00:09	4.00	239	98	981	20	481	17
Atomic/AwaitTest	1:54:45	0:00:10	688.50	452	108	439065	66	114735	49
Atomic/AwaitTest2	0:00:21	0:00:08	2.63	185	99	228	5	110	6
Atomic/ConditionalAtomicTest	>2:00:00	0:54:34	>2.20	-	436	-	193342	-	101117
Atomic/WhenReturn	0:00:26	0:00:08	3.25	186	99	228	5	110	6
Atomic/WhenReturnAll	0:00:26	0:00:08	3.25	183	100	228	5	110	6
At/AtFieldAccess	0:24:31	0:00:12	122.58	425	138	54395	13	24719	14
At/AtFieldWrite	0:04:07	0:00:11	22.45	391	97	7006	10	3309	11
At/AtNext	0:03:32	0:00:11	19.27	391	98	5959	10	2754	11

Figure 3. Exploration statistics for programs from the X10 compiler distribution

program	Time (h:mm:ss)		Speedup	Memory (MB)		#Transitions		#States	
	Basic	Opt.	(x)	Basic	Opt.	Basic	Opt.	Basic	Opt.
ArraySum	>2:00:00	0:00:49	>146.94	-	250	-	180	-	92
FSSimpleDist	0:01:21	0:00:11	7.36	309	98	1273	8	556	8
GCSpheres	0:00:27	0:00:08	3.38	182	98	25	3	18	4
Histogram	0:15:20	0:00:10	92.00	408	109	50328	36	18268	20
HistogramDist	>2:00:00	0:00:35	>205.71	-	268	-	188	-	147
KMeans	0:00:14	0:00:06	2.33	129	71	25	3	18	4
KMeansSPMD	0:00:14	0:00:06	2.33	132	71	53	3	34	4
MontyPi	>2:00:00	0:00:10	>720.00	-	141	-	21	-	22
NQueensDist	>2:00:00	0:00:18	>400.00	-	161	-	26	-	26
NQueensPar	1:40:40	0:00:18	335.56	468	167	251124	43	101883	26
SPOR	>2:00:00	0:01:44	>69.23	-	397	-	1618	-	987
StructSpheres	0:00:27	0:00:08	3.38	195	98	25	3	18	4
TutorialSum	0:00:51	0:00:08	6.38	385	98	1149	8	482	8
WhenBoundedBuffer	>2:00:00	0:00:21	>342.86	-	186	-	519	-	293

Figure 4. Exploration statistics for larger programs: 11 samples from the X10 distribution, and 3 our examples

set because one goal was to ensure that X10X can support a significant subset of the X10 language (in fact, X10X supports all parallel constructs from X10). The other goal was to measure how our techniques improve exploration. We compare the basic mode, where the X10 runtime and JPF are modified only to enable complete exploration (Section III) and the optimized mode with all our changes for faster exploration (Section IV). We tabulate the total exploration time, the required memory (as reported by JPF), and the state-space size (the number of transitions and states).

Our experiments use JPF version 5.0 with the default configuration setting (depth-first search, state matching based on isomorphism [20], etc.), including the memory limit of 1GB. All experiments were performed on an Intel Pentium 4 3.4GHz desktop, running Linux 2.6.28-18 and Sun's JVM 1.6.0_20. We set the time limit of 2 hours for each run.

Figure 3 shows the results for *all* 65 example programs from the X10 compiler distribution that focus on parallel language constructs. X10X provides a substantial speedup, with the average (geometric mean) of 19.1x. We also explored 31 simple examples that we wrote to understand language constructs. The *geometric* mean of the speedup provided by our techniques for our examples is 10.6x.

Figure 4 shows the results for 11 larger sample X10 programs from the X10 distribution, and 3 of our larger programs (`HistogramDist` which is our running example, `SPOR`, and `WhenBoundedBuffer`). X10X provides an average (geometric mean) speedup of 44.2x. Note that the relative speedup increases for programs that have larger exploration (in terms of time or state-space sizes).

While our goal was to evaluate X10X performance and not look for bugs, we still encountered a bug, specifically in the file `x10.tests/examples/Constructs/Async/-ClockAsyncTest.x10`, one of the examples from the X10 distribution. X10X reported a potential *deadlock* in `ClockAsyncTest`. We inspected the source file and confirmed the deadlock property violation, caused by a `finish` statement used in combination with a clock/barrier.

VI. Related Work

Our work studies how to obtain a high quality model checking tool for a new (parallel) language in a relatively short amount of time, which typically precludes building a tool from scratch. Previous work on this topic mostly followed two approaches.

One approach is to create a translator from the source language to the target language for which a model-checking tool is available. Several tools use this approach, e.g., Bandera [11] automatically extracts from the Java code a model that can be verified by some of the existing model checkers; Bogor [25] translates Java and some other languages to the Bogor Modeling Language; etomcrl [5] translates Erlang programs to μCLR; FeaVer [19] extracts a Promela model [18] from the C code, driven by user-specified annotations; and the first version of JPF [16], [17] translated Java code also into Promela that was checked using Spin [18]. The problems with the translation approach is that it may not support all features of the source language, and it may need to translate not only the program to be checked but also the libraries/runtime that it depends on. Our work on X10X leverages the existing X10-to-Java compiler that can translate the entire X10 language and produces Java code that can be checked, together with the X10 runtime written in Java, with JPF. For a complete and faster generation, however, we had to change the compiler, the runtime, and JPF.

Another approach to build a model checker for a new language is to modify its runtime. For example, McErlang [14] is a model checker for programs written in the Erlang programming language. McErlang modifies the concurrency system of the runtime library to enable verification. As another example, Basset [22] is a generic framework for verification of programs written in the Actor programming model [2]. Basset can support multiple actor languages but requires that their runtime be modified to connect to the Basset core. Our work also modifies the X10 runtime, but we additionally provide a new static analysis, modify the compiler, and change JPF.

VII. Conclusions

New parallel languages are emerging, and their adoption can be faster if they come with good development tools. Particularly important for parallel languages are tools for systematic testing. In this paper, we presented how to adapt an "old" model checker, such as JPF, to check a new language, such as X10. While we presented specific techniques for X10 and JPF, our overall approach generalizes to other languages and model checking tools. To model check a new language with an "old" model checker, one has to consider both problems of complete exploration and of faster exploration. Also, for the best possible results, one has to employ techniques that modify the model checker, change the language runtime, extend the compiler, and perform static analysis.

Acknowledgments

We would like to thank Steven Lauterburg and the attendees of the JPF Workshop at Google for their feedback on this work. This material is based upon work partially supported by the National Science Foundation under Grant Nos. CCF-0916893 and CCF-0746856, and by an IBM X10 Innovation Award. Milos Gligoric was supported by the Mission Critical Technologies, Inc. and NASA.

References

[1] Intel(R) threading building blocks, September 2009. http://www.threadingbuildingblocks.org/.

[2] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. The Fortress language specification, 2008. http://projectfortress.sun.com/.

[4] C. Artho and P.-L. Garoche. Accurate centralization for applying model checking on networked applications. In *ASE*, 2006.

[5] T. Arts and C. B. Earle. Development of a verified Erlang program for resource locking. In *FMICS*, 2001.

[6] E. Barlas and T. Bultan. NetStub: A framework for verification of distributed Java applications. In *ASE*, 2007.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46, 1999.

[8] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *AAMAS*, 12, 2006.

[9] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[11] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE*, 2000.

[12] Cray, Inc. Chapel language specification 0.795, April 2010. http://chapel.cray.com/.

[13] C. DeMartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *SPE*, 1999.

[14] L. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *ICFP*, 2007.

[15] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of Map-Reduce: The Pig experience. *VLDB Endow.*, 2, 2009. http://hadoop.apache.org/pig/.

[16] K. Havelund. Java PathFinder, a translator from Java to Promela. In *SPIN*, 1999.

[17] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *STTT*, 1999.

[18] G. Holzmann. The model checker SPIN. *TSE*, 1997.

[19] G. Holzmann and M. Smith. Software model checking - extracting verification models. In *FORTE*, 1999.

[20] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, 2001.

[21] JPF home page. http://babelfish.arc.nasa.gov/trac/jpf/.

[22] S. Lauterburg, M. Dotta, D. Marinov, and G. A. Agha. A framework for state-space exploration of Java-based actor programs. In *ASE*, 2009.

[23] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *CACM*, 18, 1975.

[24] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, 2008.

[25] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *FSE*, 2003.

[26] V. Saraswat. Report on the programming language X10, April 2010. http://x10.codehaus.org/.

[27] S. D. Stoller. Model-checking multi-threaded distributed Java programs. In *SPIN*, 2000.

[28] M. Ujma and N. Shafiei. JPF-Concurrent: An extension of Java PathFinder for java.util.concurrent. In *Java PathFinder Workshop*, 2011.

[29] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Springer ASE-J*, 10, 2003.

[30] X10 home page. http://x10.codehaus.org/.

[31] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *PASCO*, 2007.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.