JPF Verification of Habanero Java Programs*

Peter Anderson
Brigham Young University
Provo, Utah
anderson.peter@byu.edu

Brandon Chase
Brigham Young University
Provo, Utah
chs_brndn@yahoo.com

Eric Mercer
Brigham Young University
Provo, Utah
egm@cs.byu.edu

ABSTRACT

Habanero Java (HJ), a mid-level concurrent language, provides several correctness guarantees if its programs are data race free such as: deadlock freedom, determinism, serialization, etc. An HJ program execution can only demonstrate data race freedom for one scheduling path, but the correctness property only holds if it is data race free for all paths. Verifying an HJ program with a tool such as Java Pathfinder (JPF) for complete data race freedom is expensive in both time and memory because of the numerous JPF state explorations. This paper provides a small, stand-alone, alternative, HJ verification runtime (VR) that is more suited for verification in JPF. Additionally, this paper presents an alternative JPF scheduler that will explore only relevant HJ related scheduling paths in the VR. Finally, this paper shows state expansion results in JPF using HJ benchmarks, VR, and the scheduler.The results indicate a reduction in statespace using VR along with the scheduler.

1. INTRODUCTION

The Habanero effort at RICE University is intended to address the multicore challenge [4]. It provides a powerful set of task parallel programming constructs that are added as simple extensions to existing languages such as Java or C/C++. Unique to the Habanero approach is its emphasis on usability and safety in the parallel constructs and supporting runtime systems. Habanero guarantees properties such as deadlock freedom or determinism over subsets of its language. Such an approach may be able to open the promised performance of multicore architectures to programmers who are not experts in concurrency.

Habanero-Java (HJ) is the implementation of the Habanero paradigm in the Java language complete with compiler and runtime support [2]. HJ adds to Java new keywords and constructs for task parallel programming: `async` to create tasks, `finish` to join tasks, `isolated` to give mutual exclusion between tasks, and a `phaser` construct with the `next` keyword for software pipelines and data-flow computation. Habanero, as mentioned previously, guarantees safety properties over subsets of its language. As an example for HJ, programs that only employ the `async` and `finish` keywords are guaranteed to be deadlock-free, serializable, and deterministic if and only if the program is data-race free. The goal of the work in this paper is to verify that Habanero Java programs are data-race free.

Safety guarantees in HJ are predicated on data-race free execution. The current HJ distribution provides a tool to detect data-race as a program is executed in the HJ runtime system; thus, the tool only validates a single schedule of the program execution and is only able to tell the developer if a data-race occurred in that execution and not that the program is data-race free [10, 9]. Further, the tool requires both compiler and runtime support to annotate the source HJ program and track extra data structures.

This paper presents a new verification runtime (VR) system for HJ that is suitable to use in Java Pathfinder (JPF) to prove an HJ program is data-race free in all legal schedules. The VR system supports `async`, `finish`, `isolated`, and future data-driven forms of `async` and is able to run standalone as a new runtime system for HJ. The approach in this paper differs from recent efforts for X10 in that it creates a new HJ runtime for JPF rather than modify an existing HJ runtime, modify JPF to work with that modified runtime, and then modify the HJ compiler to better meet JPF's unique needs [3]. Further, the VR system is able to exploit nuances in JPF's virtual machine, such as low overhead in thread creation, to implement a runtime system that is small enough to manually inspect for correctness as it only comprises 992 locations in 12 different classes.

This paper further presents a new scheduler factory for JPF that is optimized for HJ program semantics. In particular, the new scheduler factory removes choice generators from schedules from the normal JPF factory that are not interesting. These uninteresting points include thread termination, entering a monitor, thread notification, etc. The new scheduler factory yields a significant reduction in states when compared to the default scheduler. To summarize, the main contributions in this paper are:

- a new runtime system for HJ small enough to understand through manual inspection and use in JPF;
- a new scheduler factory for JPF; and
- results from several benchmark examples showing performance over the HJ the default scheduler.

*Research funded by Google Summer of Code 2013 and NSF grant CCF-1302524

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

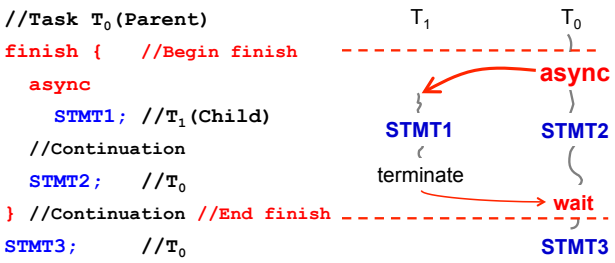

Figure 1: An example with async and finish.

2. HABANERO JAVA OVERVIEW

The overview in this paper comes from published work describing HJ including the various figures and text [2]. Only constructs supported in the VR system are described. The current HJ implementation supports most of Java 5 and Java 5/6/7 libraries are accessible in HJ programs. The HJ compiler generates Java class files that run in a standard JVM. The HJ runtime system provides the support for the added constructs (i.e., the system implements, in Java, the HJ semantics).

async: `async` is a construct for creating a new asynchronous activity. The statement `async` ⟨*stmt*⟩ causes the parent activity to create a new child activity to execute ⟨*stmt*⟩ (logically) in parallel with the parent activity. ⟨*stmt*⟩ can read or write any data in the heap and can read (but not write) any local variable belonging to the parent activity's lexical scope. The task created by `async` scheduled at the point it is declared in the program. See Figure 1.

finish: `finish` is a generalized join operation. The statement "**finish** ⟨*stmt*⟩" causes the parent task to execute ⟨*stmt*⟩ and then wait until all `async` tasks created within ⟨*stmt*⟩ have completed, including transitively spawned tasks. Each dynamic instance T_A of an `async` task has a unique *Immediately Enclosing Finish* (IEF) instance F of a `finish` statement during program execution, where F is the innermost `finish` containing T_A . There is an implicit `finish` scope surrounding the body of `main()` so program execution will only end after all `async` tasks have completed.

As an example, the `finish` statement in Figure 1 is used by task T_0 to ensure that child task T_1 has completed executing `STMT1` before T_0 executes `STMT3`. If T_1 created a child `async` task, T_2 (a "grandchild" of T_0), T_0 will wait for both T_1 and T_2 to complete in the `finish` scope before executing `STMT3`.

future: HJ includes support for `async` tasks with return values in the form of `futures`. The statement, "`final future<T> f = async<T> Expr;`" creates a new child task to evaluate `Expr` that is ready to execute immediately. In this case, `f` contains a "`future` handle" to the newly created task and the operation `f.get()` can be performed to obtain the result of the `future` task. If the `future` task has not completed as yet, the task performing the `f.get()` operation blocks until the result of `Expr` becomes available.

isolated: `isolated` ⟨*stmt1*⟩, guarantees that each instance of ⟨*stmt1*⟩ is performed in mutual exclusion with all other potentially parallel *interfering* instances of `isolated` statements ⟨*stmt2*⟩. Two instances of `isolated` statements are said to interfere with each other if both access the same shared location, such that at least one of the accesses is a write.

```
public class Foo {
    public static void main(String[] argv) {
        async {
            System.out.println("Hello World");}}}

public class Foo {
    public static void main(String[] paramArrayOfString) {
        place threadPool = Runtime.here();
        Activity childTask = new FooAsync();
        threadPool.runAsync(childTask);}}

class FooAsync extends Activity {
    // HJ task code
    public void runHjTask() {
        System.out.println("Hello World");}}
```

Figure 2: HJ Compiled "Hello World" program.

3. VERIFICATION RUNTIME

Support for `async`, `finish`, `isolated`, and `future` is handled by 4 main classes in VR, `Activity`, `Place`, `Future`, and `DataRaceDetector`.

`Activity`, inherits from Java's `Thread` class, and is used to represent each HJ task. Each `Activity` contains a reference to its parent and children tasks, a reference to a `Future` object, and methods to support the finish keyword. When an `Activity` is created the compiler inserts the computation associated with the `Activity` into its `runHjTask` method. Figure 2 shows how this works for a simple HJ program.

In HJ, all tasks created within a finish statement (including nested async statements) must complete before the parent task may proceed beyond the scope of the finish statement. In VR, the references in the `Activity` class are used to maintain this hierarchy. When an `Activity` is created via `async` keyword inside of a finish statement then it is given a reference to the `Activity` that created it and vice versa. If the child task terminates it passes its children to its parent performing an operation called task adoption. When the parent activity reaches the end of a finish statement it performs a join on all of its children including those received via adoption.

Activities can also be created using `final future<T> f = async<T> Expr;`. Whenever a `future` keyword is encountered VR creates an `Activity` that returns a `Future` upon termination. VR uses `java.util.Concurrent.CountDownLatch` to handle the synchronization between the task that creates the future (producer) and the task that requests it (consumer). Figure 3 contains methods that show how communication between consumers and the producer is handled. The producer creates and obtains the lock upon construction. When the producer has completed all of its computation and is ready to terminate it calls `setFuture` releasing the lock. This allows consumers access to the data upon all subsequent `get` calls.

The `isolated` keyword is handled outside of the `Activity` class. VR uses a single global lock, contained within the `Place` class, to enforce mutual exclusion between isolated statements. In the HJ language there are two different forms of isolated statements `isolated` ⟨*stmt1*⟩ and `isolated(Obj 1, Obj 2, ..., Obj n)` ⟨*stmt1*⟩ where the second form refines isolation to cover only the specified objects whilst the first form isolates all objects contained within its

```
//Consumer
public java.lang.Object get() {
        //Exception handling omitted
        lock.await();
        return item;}
//Producer
    public void setFuture(java.lang.Object object) {
        item = object;
        lock.countDown();}
```

Figure 3: The operation of Futures within VR.

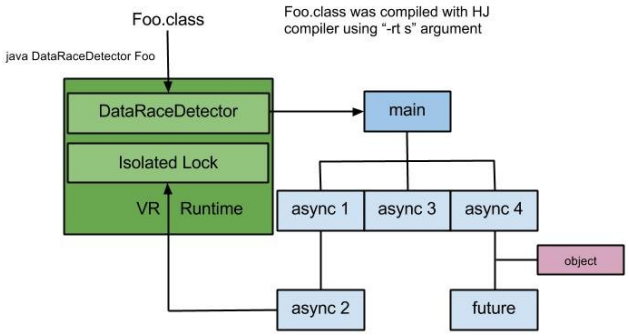

Figure 4: The VR execution and architecture.

scope. However, in VR all isolated statements are treated as the first form. This provides the same answer in terms of correctness, but is potentially less efficient than the semantics provide.

In summary, the architecture of VR is for the most part distributed amongst 4 classes. The user calls `DataRaceDetector`, the driver of the runtime, which in turn runs the HJ program the user specified. Every task is given its own thread, and each thread has variables specific to what its properties are, whether a future or an async task. Every thread also has a link to the thread that created it, and threads it created, which is used for finish statements. The isolated lock is global, and the thread that holds the lock is responsible for releasing it when it is finished. Figure 4 shows an abstract overview of the code sample in Figure 5.

VR operates independent of JPF, but is obviously best suited within JPF because of the potentially large number of threads it creates. In all VR is composed of 12 Java classes and 992 lines of code. This small design is inherently much simpler than HJ which is composed of more that 100 Java classes and >10,000 lines of code. The small design also simplifies the interaction with JPF. The repository for VR can be found at `https://bitbucket.org/bchase524/jpf-hj`.

```
public class Foo {
    public static void main(String args[]) {
        finish {
            async { // async 1
                async isolated <stmt> //async 2}
            async ; // async 3
            async { // async 4
                final future<Object> object =
                async<Object>{return object};
                object.get(); }}}}
```

Figure 5: A simple code sample.

4. JPF AND VR

JPF and VR both use Java reflection. When the user verifies an HJ program, the user runs JPF with VR as the target, and the HJ program as the input to the target. JPF uses reflection to open VR with the HJ program as an argument. VR in turn uses reflection to create a new thread to execute the HJ program and gives any remaining arguments as arguments to the HJ program. Reflection is intentional because VR has to be flexible to run an HJ program specified by the user and still provide functionality to HJ semantics. However, any property violations that JPF finds are within the HJ program and not within VR or JPF.

VR heavily relies on the soundness of JPF, and JPF's race detector. JPF's soundness is based on its partial order reduction and global search object ID. JPF's partial order reduction is critical: JPF must produce and examine essential interleaves for each thread and stop at locations where a data race occurs for checking. JPF does this by creating `ChoiceGenerators`, copying the current state of the machine, at certain parts of thread execution. JPF then systematically explores the state space, executing one thread at a time, which is called state expansion. The global search object ID is necessary for checks to ensure objects used by two threads are the same, even at different choice generators. JPF's `PreciseRaceDetector` from the default distribution is important because it checks for and reports data races that JPF has located.

VR was made with the intention to be used for verification, specifically by JPF. It was built in the simplest way possible to still produce correct results. We did not include any specific optimizations. To increase efficiency, an optimized scheduler can be integrated that is more suited with VR for scheduling, state expansion, and executing different interleavings than the default JPF scheduler factory.

The default JPF scheduler factory produces choice generators at several key locations in the partial order reduction, such as thread creation, thread termination, shared field access, monitor access, etc. However, not all state expansions from choice generators are interesting or relevant to VR. The optimized scheduler will only create choice generators for thread creation, shared array element access, and shared field access. Figure 6 shows psuedocode for the optimized scheduler. The optimized scheduler thus creates a smaller state space for JPF to verify than the default scheduler and removes the unnecessary choice generators that the default scheduler gives: thread suspense, thread resume, thread sleep, thread interrupt, thread yield, thread notify, thread terminate, shared object expose, wait, monitor enter, monitor exit, sync method enter, and sync method exit. Even after removing all these choice generators, the optimized scheduler still offers the same data race freedom guarentees that the default scheduler does. The optimized scheduler is given in the same repository as VR.

An informal proof for the optimized schedule factory's soundness and correctness follows. Suppose an HJ program data races on Thread A and B. In order for the two threads to data race, the threads will need to be alive at one point of execution. Figure 7 shows a simplified execution process for JPF using the optimized scheduler to find a data race with `ChoiceGenerators` with creation of new threads and shared field accesses.

```
createCG (string type, Thread[] threadsAvailable) {
    if (type.equals("THREAD_START"))
      createDefaultCG(type,threadsAvailable);
    else if (type.equals("SHARED_ARRAY_ACCESS"))
        createDefaultCG(type,threadsAvailable);
    else if (type.equals("SHARED_FIELD_ACCESS"))
        createDefaultCG(type,threadsAvailable);
    else
        noChoiceGenerator();}
```

Figure 6: Psuedocode for the optimized scheduler.

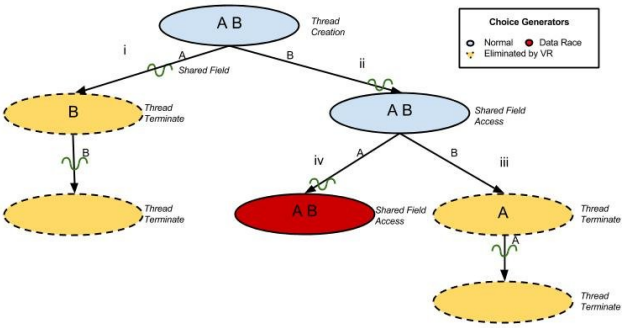

Figure 7: A JPF search illustrating the scheduler.

First, there is a `ChoiceGenerator` that has these two threads before the shared field is accessed (such as the creation of thread B). Thread A is executed first (i). The shared field is accessed by A, and JPF stores this access using global search object IDs. However, on this execution, JPF is unaware that Thread B accesses the shared field, and continues running. Thread A terminates, and JPF switches to a different thread, though a new `ChoiceGenerator` is not created. Thread B executes, and runs till B accesses the shared field. JPF stores this access using the global search object ID and finds that A also accesses it. However, since Thread A no longer is alive, a `ChoiceGenerator` is not created, and B continues to run till it terminates.

At the end of the execution, JPF back tracks to the last `ChoiceGenerator` (ii) and picks to execute Thread B. JPF executes B until B accesses the shared field. The global search object ID once again determines that Thread A also accesses the field. However, since Thread A is alive, a `ChoiceGenerator` is created. Thread B continues (iii) execution until termination, and Thread A then executes. As before (i), Thread A executes till it reaches the shared field, but finds that Thread B is no longer alive, so no `ChoiceGenerator` is created. Thread A terminates, and JPF backtracks to the last `ChoiceGenerator` and executes a different thread (iv). Thread A is selected and executed. When Thread A reaches the shared field access, JPF checks its global search object ID, finds that Thread B is still alive, and inserts another `ChoiceGenerator`.

At this point, JPF's `PreciseRaceDetector` searches at the `ChoiceGenerator` (and has been doing at all `ChoiceGenerators` thus far) for two threads that have a shared field access on the same field. If there are, which Thread A and B are, then `PreciseRaceDetector` makes an additional check for at least one being a write in order to report a data race. Since Thread A and B do, JPF with the optimized scheduler reports a data race.

Table 1: Characterization of Examples.

Test	File	Keywords	LOC	Tasks
Test 1	Example 1	A	18	12
Test 2	Example 1	A	18	102
Test 3	Example 2	A,Ft	28	3
Test 4	Example 3	A,Ft	27	3
Test 5	Example 4	A,Fn,I	19	11
Test 6	Example 5	A	15	45

Table 2: Scheduler performance in JPF.

Example	Optimized		Default	
	States	Time	States	Time
Test 1	25	00:00:00	N/A	Time Out
Test 2	205	00:00:01	N/A	Time Out
Test 3	205	00:00:01	6133	00:00:07
Test 4	718	00:00:01	9742	00:00:07
Test 5	1254396	00:07:59	N/A	Time Out
Test 6	93	00:00:01	N/A	Time Out

5. RESULTS

The HJ Distribution provides 10 small example files that make use of basic HJ constructs. A few of these files have been chosen to illustrate the performance of VR with and without scheduling optimizations. Table 1 characterizes the examples. Random initialization of arrays was removed and HJ barriers were replaced with equivalent async/finish statements on a few of the examples. Keywords are represented as async (A), finish (Fn), future (Ft), and isolated (I). Lines of Code (LOC) are also shown in the figure.

Test 1 and Test 2 are modified version of example 1, the difference of the two tests being the size of the array. The test is placing async within a for loop: summing an entry in two different arrays, and storing them into a third array. Test 3 splits an array in half. One future is charged with summing the first half of an array. The second future does the second half. Finally, the two answers are summed together resulting in the total of the array. Test 4 is similar to 3, but the answers are put into a static variable, and the futures are used to make sure the static variables are "safe" to read. Test 5 sums an array, but places the for loop within finish. Each entry is given an async isolated statement to sum the array. Test 6 focuses on asyncs within multiple for loops. All tests were limited to an hour of execution time. Tests that exceeded this limit were considered a "time out".

The performance of VR with and without the specialized scheduler is shown in Table 2. This data shows an average ten-fold reduction in state-space between the scheduled and unscheduled runtimes. In our scheduled runtime we restrict JPF to capture states on three occasions: 1) when threads are created 2) when JPF detects a shared-field access and 3) when JPF detects a shared-array access. The level of fine-tuning is relatively easy with VR compared with modern runtime libraries like HJ.

6. RELATED WORK

There is an existing extension for JPF for the X10 Language [3, 11]. Habanero is closely related to X10 in many of its constructs. In the extension, JPF operates directly on

the actual X10 runtime system. To accomplish the integration, JPF is modified, the X10 runtime is modified, the X10 compiler is extended, and a new static analysis is presented to help control state explosion. The extension represents a significant effort that affects all aspects of the X10 framework to enable JPF verification.

There is a formal model for the Chapel language with an accompanying model checker that employs symbolic execution [12]. The formal model is an intermediate representation (IR) suitable for concurrent constructs. The approach compiles Chapel programs into the IR and the model checker then verifies the IR for deadlock and data-race freedom. Creating a compiler and model checker is a significant undertaking beyond the approach in this paper. More critically, the verification tool models the runtime including the number of available worker threads to service tasks; thus, the verification results are dependent on the number of worker threads in the configuration rather than the semantics of the Chapel language.

Another approach to verifying concurrent languages is to leverage the production level language runtime system itself [5, 8, 6, 7]. These approaches typically require instrumentation of the source program, wrappers to intercept calls into the runtime, and a way to control runtime behavior. Although they are typically able to generate states faster than JPF, verification results are dependent on the employed runtime correctly implementing the language semantics.

Recent work proves the problem of state-reachability to be decidable and EXPSPACE hard for finite-valued programs in languages such as X10/Habanero [1]. The result is limited to a subset of the powerful task constructs in such languages and justifies a model checking effort. The computability and complexity of the more advanced constructs such as phasers is yet to be determined.

7. CONCLUSIONS AND FUTURE WORK

The paper presents the VR system which is an HJ runtime system for verification. The runtime supports `async`, `finish`, `isolated`, and future data-driven forms of `async` in the HJ language and is small enough to manual inspect for correctness. It runs standalone as an alternative HJ runtime system and in the JPF model checker without any modification to the runtime or JPF. When run in the JPF model checker, JPF is able to prove an HJ program data-race free. Such a proof ensures HJ safety guarantees such as deadlock freedom, deterministic execution, and the program is able to serialize. The paper further compares the VR system to HJ runtime systems. It also shows the performance gains in JPF between the default scheduler and the optimized scheduler.

Future work is to extend the VR system to support the `forall`, `forasync`, data-driven programming with `await`, `next`, `phaser` constructs, and exceptions which are also part of the HJ semantics. There is also a need to formalize the behavior of the new optimized default scheduler factory in JPF and prove that is sound and complete for proving data-race freedom in HJ programs. The scheduler will need to be further extended for the new HJ constructs, though the default schedule would still be sound and complete for data-race freedom. Finally, future work needs to address scalability in large HJ programs as the number of schedules to consider becomes rather prohibitive in these programs.

8. REFERENCES

[1] A. Bouajjani and M. Emmi. Analysis of recursively parallel programs. *SIGPLAN Not.*, 47(1):203–214, Jan. 2012.

[2] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[3] M. Gligoric, P. C. Mehlitz, and D. Marinov. X10X: Model checking a new programming language with an "old" model checker. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *ICST*, pages 11–20. IEEE, 2012.

[4] Habanero. Habanero multicore software research project. `https://wiki.rice.edu/confluence/display/HABANERO/Habanero+Multicore+Software+Research+Project`.

[5] S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In *Proceedings of the 20th international conference on Computer Aided Verification*, CAV '08, pages 66–79, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] A. Vo, S. Aananthakrishnan, G. Gopalakrishnan, B. de Supinski, M. Schulz, and G. Bronevetsky. A scalable and distributed dynamic formal verifier for MPI programs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–10, 2010.

[7] A. Vo, G. Gopalakrishnan, R. Kirby, B. De Supinski, M. Schulz, and G. Bronevetsky. Large scale verification of MPI programs using lamport clocks with lazy update. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 330–339, 2011.

[8] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical MPI programs. *SIGPLAN Not.*, 44(4):261–270, Feb. 2009.

[9] E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar. Permission regions for race-free parallelism. In *Proceedings of the Second international conference on Runtime verification*, RV'11, pages 94–109, Berlin, Heidelberg, 2012. Springer-Verlag.

[10] E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar. Practical permissions for race-free parallelism. In *Proceedings of the 26th European conference on Object-Oriented Programming*, ECOOP'12, pages 614–639, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] X10. X10: Performance and productivity at scale. `http://x10-lang.org/`.

[12] T. Zirkel, S. Siegel, and T. McClory. Automated verification of Chapel programs using model checking and symbolic execution. In G. Brat, N. Rungta, and A. Venet, editors, *NASA Formal Methods*, volume 7871 of *Lecture Notes in Computer Science*, pages 198–212. Springer Berlin Heidelberg, 2013.