Parallel Assertions for Debugging Parallel Programs

Daniel Schwartz-Narbonne, Feng Liu, Tarun Pondicherry, David August, Sharad Malik
Princeton University
{dstwo,fengliu,tpondich,august,sharad}@princeton.edu

***Abstract*—A parallel program must execute correctly even in the presence of unpredictable thread interleavings. This interleaving makes it hard to write correct parallel programs, and also makes it hard to find bugs in incorrect parallel programs. A range of tools have been developed to help debug parallel programs, ranging from atomicity-violation and data-race detectors to model-checkers and theorem provers. One technique that has been successful for debugging sequential programs, but less effective for parallel programs, is running the program using assertion predicates provided by the developer.**

These assertions allow programmers to specify and check their assumptions. In a multi-threaded program, the programmer's assumptions include both the current state, and any actions (e.g. access to shared memory) that other, parallel executing threads might take. We introduce *parallel assertions* which allow programmers to express these assumptions for parallel programs using simple and intuitive syntax and semantics. We present a proof-of-concept implementation, and demonstrate its value by testing a number of benchmark programs using parallel assertions.

I. Introduction

Debugging parallel programs is hard. A parallel program has unpredictable thread interleavings, which creates new classes of bugs. Standard sequential debugging tools are ineffective against these concurrency bugs. There are a number of tools that have been developed to help programmers debug parallel programs. Ideally, these tools should provide maximal debugging power with minimal programmer effort. In practice, there is a trade-off between the specificity (minimization of false results) and the programmer effort required to use it.

At the low specificity, low effort end of the scale are tools which check for general properties that tend to be associated with concurrency bugs, such as atomicity violations and data races. These tools are good at detecting some types of bugs, but do not utilize the programmer's knowledge about their code. Implicitly specified properties may not correspond to real bugs. Not all bugs are data races or atomicity violations: a program can be race free, but still calculate incorrect values or corrupt data. Not all data-races are bugs: in fact, research suggests that only 2–10% of data races are harmful [1]. The advantage of an implicit specification is that the programmer does not have to decide exactly what properties should be checked; the disadvantage is that the programmer may not know exactly what properties are being checked. Even within the verification community, there is a lack of consensus about the meaning of "data-race freedom" and "atomicity": both terms have multiple inconsistent definitions. For example, "atomicity" can be used to refer to transactions with or without indivisibility guarantees. The term "data-race" is similarly ill-defined [2].

At the opposite end of the spectrum are formal verification tools which provide high specificity at the cost of high effort. These tools are powerful, but complex. They require considerable skill to use correctly, limiting their utility for ordinary programmers. For example, model checkers allow programmers to specify properties on the concurrent execution using temporal logic. Theorem provers require verification conditions and often require interaction from the engineer. The computational complexity of these tools limit their use either to small programs, or to abstractions of large programs.

Running programs using user provided assertion predicates falls in the middle of the spectrum, providing medium specificity in exchange for medium effort. A programmer writing a piece of code has a set of assumptions about how the program will execute. If these assumptions can be expressed formally as assertions, they can be checked dynamically as the program runs. Assertions are widely used in sequential programming because they offer simple, intuitive predicates that can be easily expressed and understood. Despite, or perhaps because of, their simplicity, they have proven to be effective debugging tools. Kudrjavets et al. [3] demonstrate that the use of assertions is directly correlated to a decrease in bugs in real world sequential programs.

However, standard assertions are not suitable for parallel programming. In a single-threaded program, the only action that can occur during an assertion check is the assertion check itself. An assertion in a single-threaded program can therefore be expressed as a predicate on the program state at that point. In a multi-threaded program, the programmer's assumptions include both the current state, and any actions (e.g. access to shared memory) that other threads might take. For example, a programmer might assume that a given structure is not read while it is being modified. This property is difficult to express as a predicate on the system memory state at a single point in time. Debugging parallel programs requires specification and checking of both state and actions through time. Programmers need a new form of assertions for parallel programs, which provides a simple, understandable set of predicates that allows testing of complex interleaved programs. This paper introduces such an assertion mechanism, and a proof-of-concept prototype implementation. To distinguish between the two types of assertions, we name the existing form *sequential assertions* and our new form *parallel assertions*.

Tools are only useful if they are used; tools that are simple and intuitive for programmers to understand and write

978-1-4577-0118-4/11/$26.00 ©2011 IEEE

assertions for are more likely to be used. Our design provides a small, simple language that allows programmers to express a large range of important properties. We used these predicates to annotate a number of parallel programs. Our experience suggests that the small set of predicates we propose is a powerful tool for debugging real-world programs.

Paper Contributions:

- We introduce *parallel assertions*, a new mechanism for expressing correctness criteria in parallel code.
- We provide a formal semantics for these assertions.
- We present a proof-of-concept implementation.
- We evaluate the effectiveness of parallel assertions by annotating a number of benchmark programs with parallel assertions, and checking them using our implementation.

Paper Organization: We demonstrate the limitations of sequential assertions in Sec. II. Sec. III introduces the Parallel Assertion Statement, and Sec. IV describes its semantics both formally and with examples. Sec. V describes a proof-of-concept implementation. We tested the effectiveness of Parallel Assertions on a number of programs, and report the results in Sec. VI. Sec. VII discusses related work, and Sec. VIII provides some concluding remarks.

II. Limitations of the Sequential Assertion Statement in Parallel Programming

Sequential assertions are ineffective at debugging parallel programs for two reasons. They are not evaluated at the right time, and they do not check the right properties.

A. When is the assertion checked

After a sequential assertion has been checked, code that follows it should be able to rely on the fact that the assertion condition is true. However, sequential assertions are evaluated at one point in time, but the property they guarantee is used at a different point in time. Since sequential assertions are not atomic with the code they aim to protect, other threads can be scheduled between the check and the use. Thus, sequential assertions have no way of guaranteeing that the property they check will continue to hold. Each of these examples shows two threads with the position of the program statement indicating when it is executed.

1) Interference Problem: For example, consider a programmer who wishes to ensure that a shared pointer is non-null before using it. It is always possible for another thread to modify the variable between the check and its use.

```
Thread A                                Thread B
assert(buffer != NULL)
                                        buffer = NULL;
copy_to_buffer(str,buffer)
ERROR: buffer is null
```

Example I-a. Interference Problem

B. What properties can be checked

Sequential assertions are predicates over the *memory state* of a system. However, the state of a multi-threaded system includes both the current memory state and the actions that other threads are taking. Parallel code makes assumptions about what events are possible while the code is executing. These assumptions might be as simple as that a statement executes atomically (Example II-a), or as complex as the ordering requirements in a lock-free data structure (Example IV-a). Example III-a shows how an error can occur even when the other thread does not modify shared state. In each of these cases, there is no way to specify such events with a standard sequential assertion.

1) Statement Atomicity Problem: A programmer writing parallel code has an expectation that certain statements will execute atomically. However, even a simple increment instruction may actually be executed as a series of loads and stores, with a potential for interference and incorrect execution. The case below is particularly difficult to debug because each thread sees a memory state that started at x = 0, and ends at x = 1, which is correct from its point of view. The error here occurs due to conflicting accesses — no one thread could add a sequential assertion that the new value should be x = 2 if both threads have executed their statements.

```
                Initially x = 0;
Thread A                                Thread B
x++;                                    x++;
                COMPILES TO:
load r1,x                               load r2,x
inc r1                                  inc r2
store r1,x                              store r2,x
ERROR: x = 1 (expected x = 2)
```

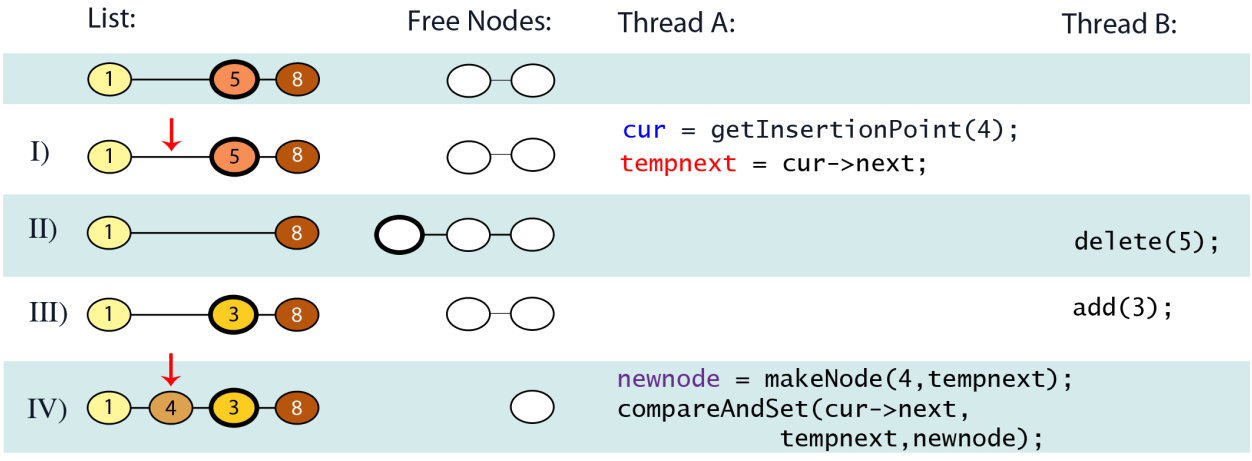
Example II-a. Statement Atomicity Problem

2) Thread Safety Problem: Programmers may add locks in an attempt to make accesses atomic. However, locks only work if every thread in the program always follows the correct locking discipline. Locking violations can even occur in cases where the offending thread only reads the variable. For example, thread B violates the lock and reads inconsistent state from the buffer, but this is not reflected by any direct change to global state by thread B. This makes it difficult for any assertion running on thread A to detect the error.

```
Thread A                    Thread B
lock(buf.lock);             //Cheating thread didn't
buf.val = foo;              //acquire lock
                            tempSize = buf.size;
                            tempVal = buffval;
buf.size = foosize;
unlock(buf.lock);
ERROR: Thread B read inconsistent state
```

Example III-a. Thread Safety Problem

3) ABA Problem: Locks are a fundamental bottleneck for parallel programs because they force the conservative serialization of potentially parallel operations. Instead, modern *non-blocking* algorithms [4] observe an initial state, speculatively prepare an update, and then only apply that update if the initial state still holds, using atomic instructions like `compareAndSet`. The programmer's intuition is that if an object has the same value at the end of the computation as at the beginning, no conflicting updates have occurred, and it is safe to commit the changes. Unfortunately, `compareAndSet`



Example IV-a. The ABA Problem

cannot distinguish between objects that have remained unmodified since last read, and ones that have been modified *twice*, leading to a subtle bug called the *ABA Problem* [4].

For example, consider a non-blocking algorithm that maintains a sorted list, shown in Example IV-a. Thread A wants to add the value 4 to this list. First it finds the insertion point, which is the node with value 1, and records that the next node is the node with value 5 (highlighted with a heavy border). Ideally, it would then use `compareAndSet` to atomically insert a new node with value 4, maintaining the sorted list property.

Unfortunately, thread B can delete the node with value 5, and then legally reinsert it with the value 3. Thread A still sees that the insertion point has a next pointer which points to the node with the heavy border, and so makes the insertion, violating the sorted list property.

An assertion to catch this error needs to check that the `compareAndSet` succeeded only if the list was in the correct unmodified state at the precise point where the `compareAndSet` occurred. Since sequential assertions are not atomic with respect to the code they protect, they can miss this bug.

III. The Parallel Assertion Statement

Sequential assertions are widely used because they offer simple, intuitive predicates that can be easily expressed and understood. Our design philosophy is to minimally extend sequential assertions, adding simple new constructs to solve the two limitations of sequential assertions in the parallel context. Other, more complicated assertions are possible. However, our experience annotating parallel programs suggests that the extensions we propose can cover a large range of properties of interest for parallel programs.

A parallel assertion has two parts that address the two limitations of sequential assertions in debugging parallel programs:

1) A parallel assertion defines a **scope** throughout which the assertion must hold. The parallel assertion scope solves the first problem with sequential assertions — they are checked at the wrong time — by unifying the assertion and the code it protects. The semantics of the parallel assertion scope are defined in Sec. IV-D.
2) It extends the **assertion condition** by providing variables that express the read and write actions by threads, lifting the second limitation of sequential assertions, i.e. what they check. These variables and their definitions are defined in section IV-E.

Syntactically, a parallel assertion looks much like a standard sequential assertion, except that instead of an assertion being located at a particular point in code, an assertion is specified with a scope associated with a `thru { ...}` statement. The basic form of the parallel assertion is language agnostic; parallel assertions can be specified in any multithreaded language with syntactic scopes. This paper simplifies the presentation of the syntax by focusing on C language examples and implementation.

In C code the assertion condition `passert_expr` is a standard side-effect free Boolean C expression, which may include the special variables defined in section IV-E.

```
thru {
  stmt;
  ... ;
  stmt;
} passert (passert_expr)
```

Figure 1. Parallel assertion scope and condition

IV. Semantics of Parallel Assertions

An assertion semantics should have several features. First and foremost, any bugs detected by an assertion should be real bugs in a real program execution. Secondly, assertion semantics which are simple, clear, and close to the semantics of the program they protect, are easier for programmers to reason about. Thirdly, assertions should only modify the execution of the program under test in limited ways. It should be possible to remove assertions from production builds without affecting the correctness of the program. A programmer should ideally

be able to use a slow debug build to test their program, and then ship a faster production build.

A. Observed Program Execution Timeline

A parallel assertion evaluation is the result of a real program execution. A parallel program consists of a set of threads, each of which generates a set of events, such as reads and writes. A program execution is an ordered interleaving of events caused by threads. A *Timeline* is an observed total ordering of these events for a particular execution of a parallel program. We ensure such a total order by placing several requirements on any implementation of parallel assertions. In many cases, these requirements will be guaranteed by the underlying hardware; in cases where they are not, a parallel assertion checking implementation must ensure they hold throughout the implementation. It is possible that this implementation may preclude certain errors from being detected by this framework, e.g. those related to errors in memory consistency assumptions.

Requirement 1: Events appear to occur *atomically*. For some actions, like writes to single words, this will be guaranteed by the underlying hardware; in cases where it is not, a parallel assertion implementation must ensure atomicity at the software level, perhaps by using locks.

Requirement 2: All threads must observe the *same consistent ordering* of events, i.e. there is a single total order. If thread 1 observes event A as occurring before event B, then every thread in the timeline must observe event A before event B. This property is inherently true on systems with sufficiently strict memory models; on other systems, a parallel assertion implementation may need to add appropriate memory fences to ensure a consistent ordering of events across multiple threads.

Requirement 3: There can be no *intra*-thread instruction reordering *across* a thru scope boundary. Scope begin/end must have acquire/release semantics — i.e. have implicit memory fences at both the hardware and programming language level.

These requirements restrict intra-thread optimizations, and may change the timing of a program, potentially changing its output. This "probing effect" is inherent to a debugging system — the act of observing a system changes its potential output. Even in a sequential program, an assertion must access the variables it references, forcing the compiler to avoid optimizations to those variables that cross the assertion boundary. The existence of a sequential assertion can even cause a program to change its memory layout, potentially hiding the effects of a buffer overflow bug. However, any bug that is found by a parallel assertion should be a real bug, representing a condition that can actually occur during program execution.

Requirement 4: Event ordering must *respect program semantics*. Let $\mathcal{P}$ be a program, $\mathcal{P}_A$ the same program with assertions, and $\mathcal{T}$ a timeline for $\mathcal{P}_A$. Then there must be some legal execution of $\mathcal{P}$ in which every read and every write occur in the same order, on the same threads, to the same locations, and have the same values as the read and write events in $\mathcal{T}$.

Requirements 1–3 may require a stricter memory model than required by the underlying program semantics, potentially requiring fences and synchronization around some operations. As a strict memory model is a valid implementation of a weak memory model, an implementation can achieve requirements 1–3 without violating requirement 4.

1) Timeline Events: Given these requirements, we can now define timestamped events on a timeline. In addition to reads and writes, there are other possible events that need to be recorded on a timeline.

Formally, an event is a tuple $\langle EventType, EventData, tid\rangle$. $EventType$ is one of four possible types listed below. $EventData$ represents auxiliary information that given event types may need to store. tid is the thread id of the thread causing the event.

A location loc is a tuple $\langle address, VariableType\rangle$ where $address$ is a memory address and $VariableType$ is a type in the underlying implementation, such as `int` or `struct foo*`. A location can be thought of as a set of bytes, ranging from $[address, address + \text{sizeof}(VariableType)]$. Two locations are equal if they have the same address and type. Two locations intersect if any of their bytes overlap.

The following four types of events are sufficient to allow the evaluation of Parallel Assertions.

- $\langle READ, \langle loc, val\rangle, tid\rangle$ A read by thread tid returned value val from location loc.
- $\langle WRITE, \langle loc, val\rangle, tid\rangle$ A read by thread tid stored value val into location loc.
- $\langle$BEGIN_PASSERT$, \phi, tid\rangle$ A parallel assertion with condition id ϕ has been triggered, i.e. entered its scope.
- $\langle$END_PASSERT$, \phi, tid\rangle$ The parallel assertion with condition id ϕ has completed, i.e. exited its scope. Note that $BEGIN$ and END must form pairs in the execution trace.

All actions should be atomic as per requirement 1. For accesses to native data types such as `int`, this is true by default. However, extended types such as `long long` may be accessed non-atomically. A parallel assertion checking implementation may treat these accesses as atomic, and raise an error if a conflicting update occurs that violates this atomicity, since this likely violates the programmer's assumptions.

2) Timestamps: Requirements 1–3 together ensure that every program execution has a consistent total ordering of events. Given a timeline, every event can be uniquely associated with a timestamp $Time(event)$ which is its order in the timeline. Thus, $Time(e_a) < Time(e_b)$ iff e_a happened before e_b.

B. Instrumented State

On the one hand, it is more convenient and natural to describe an assertion as a Boolean predicate over the state of a program than as an assertion over a sequence of events. On the other hand, the correctness of a parallel program depends on both state and actions. We resolve this difficulty by defining an *Instrumented State* which includes both the memory state at a time t, as well as variables describing the events occurring at that time t. Although for clarity we describe augmented state as unbounded arrays of variables, any implementation that can simulate these arrays is valid. An implementation which can

use static or dynamic analysis to determine that only some portion of the total state is necessary to evaluate an assertion, needs to track only that portion of the state.

The state $\mathcal{S}$ of an instrumented program at any time t, denoted $\mathcal{S}_t$, consists of three kinds of variables. We write $\mathcal{S}_t.\mathtt{v}$ for the value of variable v at time t.

1) Program Variables: These are standard program variables.

- $\mathbf{M}[\mathbf{loc}]$: This is the program memory.

2) Observation Variables: These variables encode which events are occurring at time t. Encoding events as indicator variables allows the parallel assertion condition to be expressed as a simple Boolean predicate over the instrumented state, making it easier and more familiar for programmers. In the following, ϕ_k refers to the k^{th} unique assertion in the program. Since assertion scopes are dynamic, a single syntactic assertion may be dynamically associated with several distinct ϕ_k.

- $\mathbf{R}[\mathbf{loc}, \mathbf{tid}]$: True if the event at time t was a read by thread tid from location loc,false otherwise
- $\mathbf{W}[\mathbf{loc}, \mathbf{tid}]$: True if the event at time t was a write by thread tid to location loc, false otherwise
- $\mathbf{LIVE}[\phi_\mathbf{k}]$: True at all time t between $\langle \text{BEGIN\_PASSERT}, \phi_k, tid\rangle$ and its corresponding, $\langle \text{END\_PASSERT}, \phi_k, tid\rangle$, false otherwise.

3) History Variables: Observation variables record the events occurring at time t. An assertion may also depend on the ordering between events: has one event occurred before a second one? History variables allow assertions to reference a limited amount of history. This variable records whether its argument, a Boolean predicate $\theta(\mathcal{S}_t)$, has ever been true. Our experience suggest that such a history variable offers a good trade-off between describing real-world program correctness criteria, and limiting complexity.

A parallel program may contain multiple independent parallel assertions ϕ_k, each of which will need an independent set of history variables. Each of these history variables will be associated with a different Boolean predicate on the instrumented state θ_j. All history variables associated with ϕ_k are reset to false when ϕ_k begins.

- $\mathbf{H}[\phi_\mathbf{k}][\theta_\mathbf{j}]$: $\exists \tau : \tau \leq t \wedge \theta_j(\mathcal{S}_\tau) \wedge \mathcal{S}_\tau.LIVE[\phi_k]$. True if θ_j has ever been true in the past while ϕ_k was live.

C. Instrumented State Transition Function

The program begins with: $R[loc, tid] = W[loc, tid] = LIVE[\phi_k] = H[\phi_k][\theta_j] = false$ for all $loc, tid, \phi_k, \theta_j$.

The instrumented state is updated atomically according to a transition function $T(Event_i, \mathcal{S}_t) \rightarrow \mathcal{S}_{t+1}$ (Fig. 2). This function zeroes all observation variables, and then updates the program, observation, and history variables.

D. Parallel Assertion Scope

A parallel assertion is associated with a defined block of program code, which we refer to as the *parallel assertion scope*. A parallel assertion is active from the first event that executes within the scope, until the last event that executes within the scope. Formally, the event $\langle \text{BEGIN\_PASSERT}, \phi, tid\rangle$ occurs immediately before the first statement on thread tid that is within the scope i.e. there is no event on any thread in the timeline between this event and the next event in this thread. Similarly, the event $\langle \text{END\_PASSERT}, \phi, tid\rangle$ occur immediately after the end of the last statement executed within the scope on thread tid. This requirement refers to the last statement *executed*, which may differ from the last statement in program order if code exits the scope using a `break` or `return` statement. No matter how the scope is exited, *END_PASSERT* should be the unique exit to the scope.

```
T(Event e, State S_t)
atomic {
   S_{t+1} = S_t;
   foreach(l,t)
     S_{t+1}.R[l,t] = S_{t+1}.W[l,t] = false;
   switch(e)
      case ⟨READ, ⟨loc, val⟩, tid⟩:
         S_{t+1}.R[loc,tid] = true;
      case ⟨WRITE, ⟨loc, val⟩, tid⟩:
         S_{t+1}.W[l,t] = true;
         S_{t+1}.M[loc] = val;
      case ⟨BEGIN_PASSERT, φ_k, tid⟩:
         S_{t+1}.LIVE[φ_k] = true;
      case ⟨END_PASSERT, φ_k, tid⟩:
         S_{t+1}.LIVE[φ_k] = false;
   foreach(φ,θ)
      S_{t+1}.H[φ][θ] = S_t.H[φ][θ] ∨ (S_{t+1}.LIVE[φ] ∧ θ(S_{t+1}));
   return S_{t+1};
}
```

Figure 2. Instrumented State Transition Function

A parallel assertion scope is dynamic — scopes can contain function calls and loops. Intuitively, the lifetime of a parallel assertion scope is similar to that of an automatic (stack) variable which is defined in the first instruction of the scope, and potentially used by the last instruction in the scope.

A parallel assertion holds if and only if its condition ϕ is true for all times during which the assertion is active. Note that the condition must hold *at all times*, not merely during the times when statements within the assertion scope are executing. If the thread that the assertion is associated with is swapped out by the scheduler, the assertion must continue to hold until the final statement within the assertion scope.

Every scope is associated with a thread, and variable accesses are defined relative to that thread. Multiple threads may have active scopes, which may refer to the same sections of program code. Parallel assertions can nest, both directly and through calls to functions which themselves contain parallel assertion scopes. There can therefore be many simultaneously active parallel assertion scopes.

Important note: *Inter*-thread behaviour is unaffected by the existence of a thru scope. A parallel assertion checks, but does not enforce, ordering of events between program threads.

E. Parallel assertion condition

An assertion condition ϕ is a Boolean predicate over an *instrumented state*, $\mathcal{S}_t$, of a program at time t. As such, it can be any side-effect free Boolean formula over the memory state of the program at time t, $\mathcal{S}_t.M$. In addition, we define the following Boolean predicates that use the instrumented state, and may be used as sub-formulas in ϕ. tid is the thread id of the thread with the parallel assertion.

- **LocalWrite(x)** [Keyword `LW(x)`]
  $\mathcal{S}_t.W(x, tid)$
  True if the asserting thread is writing the location x.
- **RemoteWrite(x)** [Keyword `RW(x)`]
  $\exists thrd : tid \neq thrd \wedge \mathcal{S}_t.W(x, thrd)$
  True if a thread other than asserting thread is writing the location x.
- **LocalRead(x)** [Keyword `LR(x)`]
  $\mathcal{S}_t.R(x, tid)$
  True if the asserting thread is reading the location x.
- **RemoteRead(x)** [Keyword `RR(x)`]
  $\exists thrd : tid \neq thrd \wedge \mathcal{S}_t.R(x, thrd)$
  True if a thread other than asserting thread is reading the location x.
- **HasOccurred(θ)** [Keyword `HASOCCURRED(expr)`]
  $\mathcal{S}_t.H[\phi][\theta]$
  HasOccurred functions as a latch — it takes a Boolean predicate θ, and returns true if and only if θ is true now, or has ever been true in the past while this dynamic instance of the assertion was active.

F. Checking Parallel Assertions

An instrumented state $\mathcal{S}_t$ contains all information necessary to evaluate a parallel assertion at time t. An assertion ϕ holds if $\forall \tau : \mathcal{S}_\tau.LIVE[\phi] \Rightarrow \phi(\mathcal{S}_\tau)$; it fails if otherwise.

What should happen when a parallel assertion fails depends on the purpose of the parallel assertion system. An implementation may choose when and how to report the error.

One option is that a parallel assertions should be *fail-stop*. If an event e_t with timestamp t causes the system to enter a state where any active assertion fails to hold, then the system should immediately stop. No event with timestamp $t' > t$ should be executed. A system using parallel assertions to validate a critical system might choose this approach, and enforce this condition through runtime monitoring.

A relaxed condition might be that assertion violations must be reported during the scope of the failed assertion. If assertion ϕ is active between times t_{start} and t_{end}, and ϕ fails, then no event with timestamp $t' > t_{end}$ should be executed. A system which uses parallel assertions to check a computation before doing IO could use this relaxed condition. There is a trade-off between the latency of the checker in reporting an error, and the efficiency of the checker. Because a relaxed checker is synchronized at the assertion level, not the instruction level, it is less likely to be on the critical path for program execution.

A third possible condition, useful for debugging, is that any failed assertion should be reported when convenient for the checking implementation. This decouples execution from checking, and allows for full-speed execution. Since errors will potentially be reported long after they occur, an implementation may choose to maintain state associated with detected errors, such as a program counter trace or call stack, to aid the programmer in characterizing the error.

G. Examples

1) Interference: Consider the thread interference bug in Example I-a. The following parallel assertion would detect this bug.

```
Thread A                          Thread B
thru {
                                  buffer = NULL;
  copy_to_buffer(str,buffer);
} passert(buffer != NULL);
```

Example I-b. Assertion to detect interference bug

2) Statement Atomicity: The atomicity violation in Example II-a could be detected using the following assertion. Parallel assertion semantics require begin/end and load/store events to be atomic, but do not require atomicity at the statement or block level. A parallel assertion implementation could therefore still observe an atomicity violation in this case.

```
          Initially x = 0;
Thread A                          Thread B
thru {
  x++;                            x++;
} passert(!RW(x));
```

Example II-b. Assertion to detect statement atomicity bug

3) Thread Safety: The locking violation in Example III-a can be detected using a simple check that no other thread read or wrote the protected variables. Since this assertion checks reads as well as writes, it allows the programmer writing thread A to detect a violation even in the case where no shared program state was modified.

```
Thread A                          Thread B
lock(buf.lock);                   //Cheating thread
                                  //didn't acquire lock
thru {
  buf.val = foo;
                                  tempSize = buf.size;
                                  tempVal = buf.val;
  buf.size = foosize;
}passert(!RR(buf)&&
         !RW(buf));
unlock(buf.lock);
```

Example III-b. Assertion to detect thread safety bug

4) ABA Problem: To see the value of HasOccurred, consider the ABA problem described in Example IV-a. In this case, the program is correct if either no other thread modified the pointer, or if that modification was detected and the asserting thread aborted the write. This condition can be expressed as follows. Notice that this example also shows why we need LW in addition to the RW operator.

```
Thread A                                Thread B
cur = getInsertionPoint(4);
thru {
  tempnext = cur->next;
                                        delete(5);
                                        add(3);
  newnode = makeNode(4,tempnext);
  compareAndSet(cur->next,
    tempnext,newnode);
} passert(!LW(cur->next)
|| !HASOCCURRED(RW(cur->next)));
```

Example IV-b. Assertion to detect the ABA bug

H. Evaluation Example

Example III-b uses a parallel assertion to check whether a buffer is accessed in a thread-safe manner. A possible execution timeline, and the associated instrumented state, are shown in Table I.

The assertion begins at time $t = 2$, following the acquisition of the lock at $t = 1$ (a combined read/write event). Until time $t = 4$, no other thread reads the buffer, and so the assertion holds. At time $t = 4$, thread 2 reads the buffer, and the assertion is violated, and similarly for time $t = 5$.

Note that at time $t = 6$, the assertion condition is once again true, since no other thread is accessing the buffer. However, this state is irrelevant since the assertion would have already failed at time 4.

V. Proof of concept implementation

We developed a proof-of-concept implementation as a modification to the LLVM compiler suite [5]. We implemented the assertion statement as a new AST node in the clang front-end. Assertion conditions are expressed in ordinary C, and can contain any side-effect free, function call free C predicate which can include predicates on observation and history variables as discussed earlier.

Although the timeline was defined as a total order on all events, the only events needed to evaluate the assertion condition are the start and end of assertion scope, and the read/write events on variables in the assertion condition. These events are timestamped and logged during program execution.

For efficiency, we decoupled logging from checking by using a per-thread log and a global hardware timestamping mechanism. The timestamp, location, and value of all relevant reads and writes are stored into a thread-local log. Using multiple logs prevents the log from becoming a serialization point, and the global timestamping mechanism allows events to be ordered between threads.

We implemented the logger by modifying the assembly generation stage of the LLVM compiler. Every read and write of assertion condition variables is replaced by a call to a logging function. This logging function acquires a per variable lock, and then calls a hardware timestamping mechanism (RDTSC on x86 processors[6]) which guarantees a monotonically increasing timestamp across multiple cores. (Alternatively, the Lamport logical clock [7] could be used for event timestamps.) The lock ensures that all events appear *atomically*, fulfilling requirement 1. It also creates an implicit memory fence, guaranteeing that the timestamp records a time within the locked region. Since the hardware timestamp is monotonically increasing and synchronized across all threads, and since the access must occur atomically with the timestamp, requirement 2 is fulfilled. This mechanism does not increase the set of possible program behaviors, satisfying requirement 3.

In addition, every assertion scope event must be logged. Since an assertion scope event must be atomic and ordered with respect to the variables upon which it depends, the logging function for an assertion scope event must acquire the per-variable locks for all variables referenced in the assertion condition. We acquire the locks in sorted order to avoid deadlock. Since locks contain implicit memory fences, they also serve a dual purpose of guaranteeing requirement 4.

The assertion condition is converted into a function. At the beginning of an assertion scope, we dynamically calculate the address and size of every variable mentioned in the assertion condition, and record this information as well as a timestamp and the address of the assertion condition function to the log.

A checker thread reads the logs, and merges them into a single timeline. For each event on that timeline, the checker scans the list of currently active parallel assertions. If the event is referenced by the assertion condition, then the checker calls the associated assertion checker function. If the assertion condition checking function returns true, then the assertion holds for that timestamp. If it returns false, the assertion fails at the timestamp, and an error is reported.

A. Preliminary Results

From our experiments, our implementation appears to run 10-20 times slower than native un-instrumented code. Such a slowdown may be acceptable in some debugging contexts, but is obviously non-ideal. However, much of this slowdown is a result of our minimally optimized proof-of-concept implementation, not of the complexity of parallel assertions themselves. We are working on a number of possible optimizations, which we believe will dramatically increase the efficiency of our checker. These optimizations and possible related architectural support are beyond the scope of this paper.

VI. Evaluation

One of the challenges to using either sequential or parallel assertions is knowing which properties to assert. Sometimes, the programmer knows the property that needs to hold, and expects it to hold, but is not sure that it actually holds. In other cases, the programmer may not know exactly which properties will hold. They can build a mental model of the program, and then use assertions to explore whether that model accurately represents the actual program under test.

We tested the effectiveness of parallel assertions by annotating a number of programs. All annotated programs have been successfully run through our tool.

TABLE I
INSTRUMENTED STATE FOR AN EXECUTION OF EXAMPLE III-B

t	Event	buf.lock	buf.size	buf.val	LIVE[ϕ]	W[buf,tid_A]	R[buf,tid_B]	RR[buf]	RW[buf]	ϕ
0		0	oldSize	oldVal	false	false	false	false	false	true
1	⟨READ, buf.lock,0,tid_A⟩ ⟨WRITE, buf.lock,1,tid_A⟩	1	oldSize	oldVal	false	false	false	false	false	true
2	⟨BEGIN_PASSERT,ϕ, tid_A⟩	1	oldSize	oldVal	true	false	false	false	false	true
3	⟨WRITE, buf.val,foo,tid_A⟩	1	oldSize	foo	true	true	false	false	false	true
4	⟨READ, buf.size,oldSize,tid_B⟩	1	oldSize	foo	true	false	true	true	false	**false**
5	⟨READ, buf.val,foo,tid_B⟩	1	oldSize	foo	true	false	true	true	false	**false**
6	⟨WRITE, buf.size,foosize,tid_A⟩	1	foosize	foo	true	true	false	false	false	true
7	⟨END_PASSERT,ϕ, tid_A⟩	1	foosize	foo	false	false	false	false	false	true
8	⟨WRITE, buf.lock,0,tid_A⟩	0	foosize	foo	false	true	false	false	false	true

A. Microbenchmarks

We created a number of microbenchmarks based on examples from two standard references on parallel programming techniques and bugs, including *The Art of Multiprocessor Programming* [4], and *Modern Operating Systems* [8]. These represent common multithreaded programming errors and thus would benefit from parallel assertions.

1) Bounded Buffer: BoundedBuffer is a bounded buffer implementation using semaphores [8]. We use assertions to check that the same location is not accessed by two threads simultaneously. We further check that the head and tail pointer modification and data insertion is atomic. These conditions are guaranteed as long as locking is done correctly.

2) Dual Stack: DualStack is a parallel stack implementation using the reserve and fulfill technique [4]. The implementation contains a subtle error that can lead to data corruption. If a thread executes push in between the reserve and fulfill of another thread's push, a stack location is overwritten. We check for this situation by asserting that no other thread writes the location during the entirety of the push function. We also check that no thread attempts to read the location until the location is marked as full to check that the data is written before it is read.

3) Fine Grained List: FineGrainedList is a fine grained locking implementation of a list based set [4]. We added assertions to check whether locks are acquired correctly when traversing the list. This is done by asserting that once the add function has read the pointer specifying the insertion point, that pointer remains unwritten until completion of add. A similar assertion checks that the pointer being altered during remove is not accessed by any other thread.

B. PARSEC Benchmarks

PARSEC [9] is a well-known suite of parallel programs. These programs are meant to be a representative sample of modern workloads, and as such are therefore a further test of whether parallel assertions can be used on real programs. We are engaged in ongoing to work to add assertions to all PARSEC benchmarks; here we report on the randomly selected sample we have completed so far.

1) Black Scholes: Black Scholes uses do-all parallelism to calculate the value of a set of options. It was remarkably easy to annotate with assertions — we simply added assertions that every privatized variable was neither read nor written by any thread other than its owner.

2) Dedup: Dedup compacts a filesystem by replacing duplicated sections of files with pointers into a shared hash table. Dedup uses a somewhat complicated locking structure. Many concurrent accesses are not directly protected by locks, instead, the code warns through comments that mutexes must be held when calling these unprotected functions. We added assertions to check that these operations proceeded atomically, as specified in the comments.

In addition, elements are connected to mutexes through a hash lookup. We added assertions that any accesses for which a mutex had been acquired should be performed atomically.

3) Fluidanimate: Fluid animate is a numerical solver for fluid mechanics. The simulation is divided into a number of chunks, and each thread is responsible for calculating its chunk. We added an assertion that no element is accessed by more than one thread, and were surprised to find that this assertion failed. Examination of the code revealed that the boundaries between chunks are shared between two threads, and can be modified by both. In this case, the use of parallel assertions revealed that our assumptions about the parallaization technique used were incorrect, and helped deepen our understanding of the underlying program. In fact, we believe that this may be a productive way for a programmer to explore the concurrency model of a new program — add assertions that reflect their understanding, and see which ones hold on real executions.

4) Streamcluster: Streamcluster is a solution to the online clustering problem in data mining. The data points are statically partitioned into thread workloads during the parallel gain computation. The parallel gain computation iterates over a subset of data points in the input data set several times. We added assertions to check that on each iteration a thread access to a data point has occurred, no other thread accesses that point.

5) Swaptions: Swaptions uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions. The array of swaptions is divided into a set of blocks, and each blocks is assigned to a thread. Each thread then uses a monte-carlo simulation to determine the cost of the swaption. We added assertions to check that swaptions were correctly privatized, with only one thread reading and writing each element.

VII. Related Work

There has been a wide variety of work on tools to help programmers write correct parallel code.

Some of this work focuses on new languages and runtime systems to ensure that parallel programs are correct by construction. Work along this line includes transactional memory systems, e.g. [10], concurrent functional programming languages [11], stream programming [12], and automatic parallelization techniques [13]. While this work is valuable, the widespread adoption of threaded programming models, and the large amounts of legacy code in imperative threaded languages suggest the need for tools to verify multi-threaded C programs.

Verifying parallel programs is an active area of research, and there is a wide range of related work. This work can be categorized along two axes: What properties they check, and how they check those properties.

A. What they check — Property Specification

In general, there is a trade-off between the completeness of a program specification, and the ease with which that specification can be developed. At one end of the scale are verification tools that use implicit correctness criteria, and simply check for conditions that often represent concurrency bugs. Some of these tools, such as [14], [15], and [16] check for race conditions. Others, such as the lockset algorithm [17], check to ensure that locks are used in a standard, consistent manner. Vlachos et al. [18] propose a tool that uses taint analysis to check parallel programs for potentially dangerous uses of unsanitized input. Lucia et al. [19] weaken the data-race freedom condition and check for *conflict freedom of synchronization-free regions*, which they argue can be efficiently guaranteed at runtime.

These tools are useful for solving problems within their domain. However, they both under and over specify. A race free program does not imply a correct program. On the other hand, a program with data races is not necessarily an incorrect one. The canneal benchmark in the PARSEC suite, for example, deliberately allows data races because its probabilistic algorithm is capable of recovering from the errors they introduce. In general, tools with implicit specifications do well at catching standard bugs in standard programs. They do less well on programs that use non-standard tricks to improve performance. A lock-set algorithm, for example, cannot check a modern non-blocking algorithm.

At the other extreme, approaches like Owicki-Gries [20] require the programmer to specify Hoare triples and invariants for all operations within their program. Elmas, Tasiran and Qadeer [21] require the programmer to create an abstract model of their program, and then check whether the program undergoing testing is a correct refinement of the model. Theorem provers, such as HAVOC [22], require the programmer to provide function contracts and loop invariants. HAVOC attempts to limit the need for programmer annotation by attempting to infer and then prove possible additional contracts from a partial specification. However, theorem proving tools still require the programmer to provide a relatively complete set of annotations. These are powerful tools in the hands of verification engineers. However, they require a considerable amount of skill and effort to use effectively, and it is unclear how useful they are to ordinary programmers.

In the middle are tools which provide programmers with a set of primitives that allow them to express common correctness criteria. Kovacs et al. [23] present a framework for writing full temporal assertions on message passing programs. Although these temporal assertions are powerful, they are also complex. They require the programmer to write a Java class for each temporal property they wish to assert. Since a temporal logic property may depend on events that will happen in the future, their semantics introduces the concept of a partial tree. A temporal assertion which depends on future events will have the value "unknown" ($\perp$) until these events occur, limiting the use of temporal assertions for parallel runtime validation.

Vechev et al. [24] uses a set of Java primitives and the QVM virtual machine [25] to allow programmers to assert properties about object ownership in multithreaded programs. If a thread knows that no other thread has a reference to an object, then it can know that no conflicting accesses will be made to that object.

JASS [26] extends Java by allowing programmers to express parallel correctness conditions as conditions on objects. Object method calls can be annotated with pre and post-conditions. Objects can also be annotated with class invariants, trace invariants, and refinement relations. Properties are checked when the object is stable, i.e. not being modified by any method call. These techniques are designed for systems like Java, where accesses occur through method calls and where data privatization and object synchronization is enforced by a runtime system. They are less applicable in languages without these features, like C.

B. How they check

Just as there is a trade-off in property specification between completeness and programmer effort, there is a trade-off in property checking between completeness and checker effort. These two trade-offs are orthogonal — it is possible to check whether a complicated temporal logic property holds on a single trace, and it is possible to run a theorem prover to find all possible data-races in a program.

At the one extreme, some tools are trace based. They efficiently monitor single runs of a program. This limits their observational power — they can only find bugs that manifest in the observed execution trace. However, such tools can be fast. Vlachos et al. [18] present a hardware assisted runtime checker for parallel computations that can check for misuse of tainted data with low overhead.

Trace based techniques can be extended in several ways. Trace based tools can use predictive analysis [27] to determine whether an error *could* have occurred on a given trace, even if it did not actually do so due to non-deterministic scheduling.

Tools such as DMP [28] and CHESS [29] perturb the execution of the program under test in ways that they hope will cause more bugs to manifest.

Model checkers, such as SPIN [30], can explore the transition relation of a program, effectively testing all possible execution traces at once. A model checker may have to consider all possible thread interleavings over all possible traces. STORM [31] reports good results using bounded model checking to explore all execution traces that have less than k context switches.

Theorem provers such as HAVOC [22] use contract based reasoning to abstract complex code. Ideally, if the code is modular and the contracts are well defined, HAVOC can factor a large program as a set of smaller, more tractable programs. HAVOC has been used to prove the correctness of the synchronization protocol in a 300,000 line Microsoft Windows component.

Our work adds a unique point to this spectrum by providing flexible user defined assertions that cover a range of concurrent programming bugs. While we describe checking parallel assertions on a single trace, either on or off-line, these assertions could also be verified over all traces through model checking/theorem proving.

VIII. Conclusion

Parallel assertions provide a powerful and intuitive tool that allows programmers to express their knowledge about their code as parallel correctness criteria, using familiar syntax and a minimal set of new predicates. Parallel assertions are efficient to write, easy to reason about, and allow programmers to detect many different types of errors that could previously only be identified with specialized tools. They are also efficient to implement, since the correctness of a parallel assertion at a given time depends only on the instrumented state of the program at that time. A parallel execution checker can run as a concurrent monitor for a parallel program.

Our initial work explores some of the possible uses of parallel assertions. Our proof-of-concept implementation, and our experience annotating programs with assertions, demonstrate that parallel assertions can be effective at defining and catching parallel bugs. One exciting possibility is that they may also be able to help prevent them. A parallel assertion checker, paired with an effective exception handling system, may allow programs to detect and recover from concurrency bugs. The optimizations necessary to make such a system efficient are the subject of ongoing work. Parallel assertions are potentially a powerful debugging tool with significant potential for future applications.

References

[1] W. Zhang, C. Sun, and S. Lu, "ConMem: detecting severe concurrency bugs through an effect-oriented approach," in *ASPLOS '10*. ACM, 2010, pp. 179–192.

[2] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, pp. 74–88, March 1992.

[3] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical investigation," in *ISSRE '06*, Nov. 2006, pp. 204 –212.

[4] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, Mar. 2008.

[5] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002.

[6] "Intel® 64 and IA-32 architectures software developer's manual," 2010. [Online]. Available: http://developer.intel.com/Assets/PDF/manual/253667.pdf

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978.

[8] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[9] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[10] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott, "Lowering the overhead of software transactional memory," in *ACM SIGPLAN Workshop on Transactional Computing*, Jun 2006.

[11] J. Armstrong, "A history of Erlang," in *HOPL III*. ACM, 2007.

[12] W. Thies, "Language and compiler support for stream programs," Ph.D. Thesis, MIT, Cambridge, MA, Feb 2009.

[13] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *MICRO 2007*, dec. 2007, pp. 69 –84.

[14] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware Java runtime," *SIGPLAN Not.*, vol. 42, pp. 245–255, June 2007.

[15] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *PPoPP '03*. ACM, 2003, pp. 167–178.

[16] A. Muzahid, D. Suárez, S. Qi, and J. Torrellas, "SigRace: signature-based data race detection," *SIGARCH Comput. Archit. News*, vol. 37, pp. 337–348, June 2009.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, pp. 391–411, November 1997.

[18] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry, "ParaLog: enabling and accelerating online parallel monitoring of multithreaded applications," *SIGARCH Comput. Archit. News*, vol. 38, pp. 271–284, March 2010.

[19] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races," in *ISCA '10*, 2010.

[20] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Commun. ACM*, vol. 19, pp. 279–285, May 1976.

[21] T. Elmas, S. Tasiran, and S. Qadeer, "VYRD: verifying concurrent programs by runtime refinement-violation detection," in *PLDI '05*. ACM, 2005, pp. 27–37.

[22] T. Ball, B. Hackett, S. Lahiri, S. Qadeer, and J. Vanegue, "Towards scalable modular checking of user-defined properties," in *Verified Software: Theories, Tools, Experiments*. Springer, 2010, vol. 6217, pp. 1–24.

[23] J. Kovacs, G. Kusper, R. Lovas, and W. Schreiner, "Integrating temporal assertions into a parallel debugger," in *Euro-Par 2002 Parallel Processing*. Springer Berlin / Heidelberg, 2002, vol. 2400, pp. 159–252.

[24] M. Vechev, E. Yahav, and G. Yorsh, "PHALANX: parallel checking of expressive heap assertions," in *ISMM '10*. ACM, 2010, pp. 41–50.

[25] M. Arnold, M. Vechev, and E. Yahav, "QVM: an efficient runtime for detecting defects in deployed systems," *SIGPLAN Not.*, vol. 43, pp. 143–162, October 2008.

[26] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim, "Jass - Java with assertions," *Electr. Notes Theor. Comput. Sci.*, vol. 55, no. 2, 2001.

[27] A. Farzan and P. Madhusudan, "The complexity of predicting atomicity violations," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, vol. 5505, pp. 155–169.

[28] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, "DMP: Deterministic shared-memory multiprocessing," *Micro, IEEE*, vol. 30, no. 1, pp. 40 –49, jan.-feb. 2010.

[29] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *OSDI'08*. USENIX Association, 2008, pp. 267–280.

[30] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, Sep. 2003.

[31] S. K. Lahiri, S. Qadeer, and Z. Rakamaric, "Static and precise detection of concurrency errors in systems code using SMT solvers," in *CAV*, vol. 5643. Springer, 2009, pp. 509–524.