Run-Time Verification of Optimistic Concurrency

Ali Sezgin[1], Serdar Tasiran[1], Kivanc Muslu[1], and Shaz Qadeer[2]

[1] Koc University, Istanbul, Turkey
{asezgin,kmuslu,stasiran}@ku.edu.tr
[2] Microsoft Research, Redmond, WA, USA
qadeer@microsoft.com

Abstract. Assertion based specifications are not suitable for optimistic concurrency where concurrent operations are performed assuming no conflict among threads and correctness is cast in terms of the absence or presence of conflicts that happen in the future. What is needed is a formalism that allows expressing constraints about the future. In previous work, we introduced tressa claims and incorporated prophecy variables as one such formalism. We investigated static verification of tressa claims and how tressa claims improve reduction proofs.

In this paper, we consider tressa claims in the run-time verification of optimistic concurrency implementations. We formalize, via a simple grammar, the annotation of a program with tressa claims. Our method relieves the user from dealing with explicit manipulation of prophecy variables. We demonstrate the use of tressa claims in expressing complex properties with simple syntax.

We develop a run-time verification framework which enables the user to evaluate the correctness of tressa claims. To this end, we first describe the algorithms for monitor synthesis which can be used to evaluate the satisfaction of a tressa claim over a given execution. We then describe our tool implementing these algorithms. We report our initial test results.

1 Introduction

The main challenge in reasoning about concurrent programs is taking into account the interactions among threads on shared memory. An effective way to cope with the complexity of concurrent shared-memory programming is to specify and verify partial safety properties which are typically expressed as assertions over program variables.

For implementations based on optimistic concurrency, our experience suggests that expressing properties about concurrency control mechanisms in the form of assertions is unnatural and counter-intuitive. In optimistic concurrency, a thread accesses a shared resource as if there are no competing threads for the same resource and eventually validates whether this assumption was correct. If it was, then it *commits*; if not, it *rolls-back* any visible global change and, optionally, re-starts. Correctness in these implementations, such as those of non-blocking data structures or Software Transactional Memories (STM's) [1], cannot

G. Roşu et al. (Eds.): RV 2010, LNCS 6418, pp. 384–398, 2010.

be easily expressed as variations of assertions. Instead, one needs to express the desired properties in terms of future behavior, for instance, what needs to hold at the present program state if the method call or transaction completes without conflict.

In our previous work [2], we have introduced tressa claims and incorporated prophecy variables in order to relate the program state at which a tressa claim is executed to the rest of the execution. Our objective was to simplify the use of QED [3] in reduction proofs of optimistic concurrent programs. Intuitively, $\mathsf{tressa}\,\varphi(p)$ executed at state s expresses the belief that $\varphi(p)$ holds at s as long as the rest of the execution agrees with the *current* value of the prophecy variable p. For instance, imagine that the (boolean) return value res of a method m is mapped to the prophecy variable, $pRes$: An execution of m returns true iff $pRes$ is equal to true during the execution of m. Then, the expression $\mathsf{tressa}\,pRes \Rightarrow \phi$ executed at state s claims that ϕ at s is required to hold only in those executions in which m returns true.

In this paper, we investigate the tressa construct with an eye towards specification and run-time verification. Tressa claims are suitable for specifying properties for optimistic concurrency because they provide a means to relate the outcome of a sequence of events yet to occur with the program state at which the tressa claim is executed. Reading in contrapositive form, the tressa claim of the previous paragraph expresses the requirement that if ϕ is false, then m should return false. This pattern appears often in optimistic concurrency if, for instance, ϕ expresses non-interference.

Instead of cluttering our specification methodology by prophecy variables, we define a grammar for expressing tressa claims. It is possible to address the value last/first written to or read from a variable by a particular subset of all active threads, or the value of a variable immediately after a desired method terminates. For instance, the above tressa claim would be replaced with $\mathsf{tressa}\mathsf{Exit}(res) \Rightarrow x = y$, where $\mathsf{Exit}(res)$ is the value of res immediately after m terminates. Thanks to this approach, the user is not required to look for and then annotate the proper places in the code where prophecy variables have to be managed, a process which is both error-prone and tedious.

The truth value of a tressa claim is a function of the program state where the tressa claim occurs and the execution suffix following this occurrence. Given the rather simple semantics, they are amenable to low complexity run-time verification which could help uncover subtle bugs of concurrency. Accordingly, we develop a run-time verification tool for tressa claim based specifications. We first describe monitor synthesis algorithms. We show that the complexity of monitor synthesis is linear in the sum of the size of the tressa claims checked during the execution.

We then present our framework which implements these algorithms. Our implementation is built on top of the CHESS tool [4]. CHESS allows complete coverage of interleaving executions up to a desired bound on the number of context switches. Since the number of context switches as well as the number of different variables and threads that manifest interesting bugs are typically small [5], the

use of CHESS makes our tool stronger than random testing, that is, synthesizing tressa monitors over random interleavings. We demonstrate our tool over a sample of interesting implementations.

Related Work. The specification formalisms to generate monitors for run-time verification usually employ a variant of linear temporal logic, LTL [6,7,8]. As non-regular properties cannot be expressed in LTL, more expressive formalizations have recently been developed [9,10,11]. Our formalization is not more expressive than the latter formalizations; the strength in our approach comes from two aspects. First, tressa claims have a relatively simple syntax with intuitive constructs. This we believe will lead to a short learning phase. Second, the constructs we use transfer the burden of identifying, in the code, places corresponding to an event of interest from the user to the run-time verification tool. This removes the possibility of incomplete or erroneous annotation (with auxiliary variables) by the user. As far as the complexity of monitor synthesis and run-time verification is concerned, we propose an on-the-fly algorithm of linear time complexity in the number of tressa claims and of logarithmic space complexity in the length of the execution, on par with other recent work [9,10].

2 Motivation

In this section, we will give an example where assertion based reasoning fails to capture the natural correctness requirement.

Specifications with tressa claims. Consider the code given in Fig. 1 which is a simplified version of an atomic snapshot algorithm (e.g., [12]). The snapshot algorithm aims at obtaining a consistent view of a set (here, a pair) of addresses shared among concurrent threads each of which might either be trying to take

```
public Pair Snapshot(int a, int b)
{
 int va, vb, da, db;
 boolean s = true;

 atomic{ va = m[a].v; da = m[a].d; }          // Rda
 atomic{ vb = m[b].v; db = m[b].d; }          // Rdb

 // if method is to succeed, da and db form a consistent snapshot.

 atomic{ if (va<m[a].v) { s = false; } }      // Vala
 atomic{ if (vb<m[b].v) { s = false; } }      // Valb

 if (s) { return new Pair(da,db); }           // Comm
 else { return null; }                        // Abrt
}

public void Write(int a, int d)
{
 atomic{ m[a].d = d; m[a].v ++; }
}
```

Fig. 1. A collection that implements an atomic read of two distinct variables, `Snapshot`, and random access updates, `Write`

a snapshot or updating a shared address. For convenience, we assume that code blocks tagged with `atomic` are executed atomically (without interleaving).

The method `Snapshot` takes in two addresses, and tries to return a consistent pair of values stored in these two addresses. In any lock-based implementation, shared variables are accessed only after obtaining their exclusive ownership, e.g., via locks. This implementation, however, uses optimistic concurrency because as `m[b]` is being read (line `Rdb`), the lock for `m[a]` is not owned and any thread is free to update `m[a]`'s value. This is an *absence of conflict* assumption which needs to be eventually validated (the second round of reads of the same addresses at lines `Vala` and `Valb`). Each location is assumed to contain a version number which is incremented whenever a value is written by a call to `Write`. It is this version number which `Snapshot` uses in validation: a different (greater value) version number than the local copy indicates that its copy of the address is stale.

A correct snapshot algorithm should either terminate unsuccessfully and return a default value such as `null`, or, it should appear to take an instantaneous snapshot (of `m[a].d` and `m[b].d`) and return the values read. The implementation in Fig. 1 is correct in this sense. Intuitively, if the version number `m[a].v` has not been incremented by another concurrent write between lines `Rda` and `Vala`, then it is ensured that `m[a].d` is unchanged in this time interval, and `m[a].d` is equal to `da`. A similar argument holds for lines `Rdb`, `Valb`, `m[b].d` and `db`. It is most natural to put a claim about this desired property at the point in the program where it needs to hold, i.e., between `Rdb` and `Vala`.

Observe that if `Snapshot` is to terminate successfully, it must be the case that `m[a].d`= `da` and `m[b].d`=`db` between the lines `Rdb` and `Vala`. However, this guarantee is not about the past or the execution prefix, but rather of possible future behavior or the execution suffix. As such, any assertion, which can only relate execution prefixes to program states, placed between the first pair of reads and the rest of the method will fail to capture this property. For instance, we cannot assert `s==>(da==m[a].d)` between `Vala` and `Valb` because immediately after `m[a]` is found to be untouched (local and global version numbers are found to be equal at `Vala`), a context switch might occur and another thread might update `m[a]` which will violate the assertion. Assertions before `Vala` or after `Valb` will fail in a similar manner to express the desired property.

Generalizing the above arguments for arbitrary optimistic concurrency implementations, we can state that typically the programmer will need to specify a condition to be satisfied at the present state where the condition itself depends on the rest of the execution. For this class of specifications, we propose the use of tressa claims. Simply put, a tressa claim holds true only if the remaining part of the execution conforms to the claim. Capturing the remaining part of the execution is accomplished via special constructs (prophecy variables) denoting values of variables attained after a certain event occurs. This enables us to relate the present state to the eventual outcome and express the desired correctness property naturally.

```
...
atomic{ vb = m[b].v; db = m[b].d; }

tressa Exit(s) ==> (da==m[a].d && db==m[b].d);

atomic{ if (va<m[a].v) { s = false; } }
...
```

Fig. 2. A possible specification for `Snapshot` expressed in terms of future behavior

Now, consider the modification given in Fig. 2 containing a tressa claim. The claim states that if the method commits (in the future) (`Exit(s)`, which denotes the value of `s` when `Snapshot` terminates, is `true`), the current values of the pair of `da` and `db` will constitute a consistent snapshot of addresses `a` and `b`. The claim is located where we expect the condition to hold and (the implicit prophecy variable) `Exit(s)` allows the claim to be checked for only successfully terminating `Snapshot` executions.

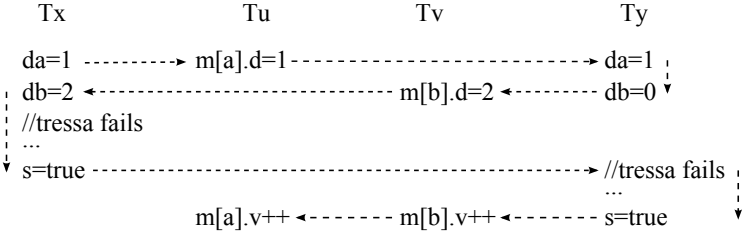


Fig. 3. An interleaving that exhibits an inconsistent snapshot

Bug Manifestation. In order to illustrate how tressa claims can help distinguish faulty behavior from correct ones, consider another implementation which uses the same `Snapshot` method, but has a broken non-atomic `Write` whose body is given as:

```
m[a].d = d; atomic{ m[a].v ++; }
```

That is, in the `Write` method, there can be arbitrarily many actions between the update of the data value, `m[a].d`, and the incrementing of the version number, `m[a].v`. This is a faulty implementation and the tressa claim we introduced in Fig. 2 will be violated in some executions and thus catch the bug. Incidentally, the underlying correctness criterion for our sample implementation is *linearizability* [13], but a discussion on linearizability goes beyond the scope of this paper. This buggy version allows executions which cannot be linearized, an example of which is given below.

Consider the execution given in Fig. 3. Dashed arrows represent time flow, each column represents the execution of a thread whose id is given at the top

of each column and each row corresponds to a time instant when one of these threads makes a transition. In this sample execution, threads `Tx` and `Ty` execute `Snapshot(a,b)`, taking snapshots of addresses `a` and `b`. Threads `Tu` and `Tv` execute `Write(a)` and `Write(b)` updating the contents of `a` and `b`, respectively. Assume that the initial value of each address is `0`.

Thread `Tx` reads 0 for `m[a].d`, 2 for `m[b].d`. Thread `Ty` reads 1 for `m[a].d`, 0 for `m[b].d`. Since the version numbers are updated non-atomically, in this particular execution, both snapshot methods will conclude success and return their snapshot. However, both of the returned snapshots is inconsistent in any possible linearization of this sequence. This is an erroneous execution and the tressa should fail; it indeed does. When the tressa claim is evaluated in thread `Tx`, `Exit(s)` is true, because `Tx` does not observe any of the updates done by `Tu` and `Tv` and ends `Snapshot` deciding non-interference (`s` is set to true when `Tx` terminates), but `da` (0) is not equal to the current value of `m[a].d` (1). Similarly, the tressa of thread `Ty` also fails as its copy of `Exit(s)` is also true but the values of `db` (0) and `m[b].d` (2) are not equal.

Once the violation is generated, the user will be presented with the counter example which clearly identifies the failing tressa claims. Since the tressa claim depends on the outcome of the `Snapshot` method, and the values held in `da` and `db` after executing `Rdb`, its failure means that even though there was an interference from concurrent threads, the validation part of `Snapshot` failed to detect this. This would point to two possible sources of failure: (i) the validation part is erroneous, or (ii) the interference is not properly reported. In our case, it is the latter and the tressa violation helps the user identify the nature of the bug.

The tressa claim we gave above was an approximate specification of linearizability (or atomicity). This is for illustration purposes only. We will present other examples of tressa claims which express properties tailored to the implementation under consideration and not just general correctness criteria like linearizability.

3 Formalization

Programs. A *concurrent program text* is a collection of *procedures*, where each procedure is written according to a given grammar representing the underlying programming language. Each procedure has well-defined *entry* and *exit* points. Each well-formed sentence in the programming language assumed to be executing atomically is called a *statement*. A *program* is a mapping from a set of live threads Tid to the procedures of the program text. Intuitively, a program identifies which thread is running which procedure. We imagine a potentially infinite set of *program states*. Each program state holds all the necessary control information and the valuation of each program variable. We ignore the details of how this information is encoded in a program state. We distinguish a subset of program states, called the *initial states*, which are intuitively those states from which a program can start its execution.

Each program generates a set of *runs*, alternating sequences of program states and *dynamic statements*. For $t \in Tid$ and s a statement, the pair (t, s) is called

a dynamic statement. Each run, $q_0 \xrightarrow{d_1} q_1 \ldots \xrightarrow{d_n} q_n$, where q_i's represent states, d_i's represent dynamic statements, satisfies the usual initial and transition conditions: The state q_0 is an initial state and for each triple $q_{i-1} \xrightarrow{d_i} q_i$, it is possible to make a transition from program state q_{i-1} to q_i by executing the dynamic statement d_i. The semantics of each transition is governed by the programming language.

Tressa Claims. A tressa claim is a special statement of the form $\mathsf{tressa}\,\phi$, where tressa is assumed to be part of the programming language lexeme and ϕ is a tressa predicate, an element of the set $Pred$ whose syntax is given below.

$$
\begin{aligned}
Pred &::= rel_k(Expr^k) \mid Pred \wedge Pred \mid \neg Pred \\
Expr &::= Term \mid f_k(Expr^k) \\
Term &::= \mathsf{First}(var, AType, tSet) \mid \mathsf{Last}(var, AType, tSet, cond) \mid \mathsf{Exit}(var) \mid var \\
AType &::= \mathsf{Rd} \mid \mathsf{Wr} \mid \mathsf{RW}
\end{aligned}
$$

Each predicate is either the result of the application of some k-ary relation to k expressions ($Expr$) or a boolean combination of predicates. Each expression is either some term ($Term$) or some k-ary function f_k over expressions. $AType$ is the access type indicator: Rd is used for only read accesses, Wr for write accesses, and RW for read or write accesses.

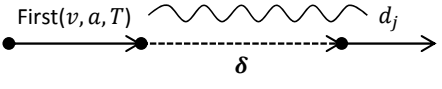
Example. Our example is drawn from software transactional memory implementations. Consider the code given in Fig. 4, snippets from a program intended as a test harness for the Bartok STM implementation [14]. It starts by initializing the transaction (`DTM.Start`). The value stored in the shared object `o1` is then transactionally read (`DTM.OpenForRead`) and this read value incremented by 1 is then transactionally written (`DTM.OpenForUpdate`) into `o2`. We require that if the transaction succeeds after doing these two operations, the values in `o1` and `o2` turn out to be exactly as updated by this transaction, i.e., (`o1.d`)+1=`o2.d`. This is expressed by the first tressa claim. We also require that an object that is only read (not updated) in a transaction should have its value constant throughout the execution span of a transaction that commits. In our code, `o1` is one such object and the second tressa claim expresses this property.

As another interesting property, we want to express correct roll-back in the case of aborting a transaction. In our STM, each update is logged in a thread local list. As a variable is updated, an entry is inserted into this list, specifying the overwritten value. Then, to roll back the changes made to the shared address space, the list is traversed in reverse order, canceling the effect of each update until all the variable values are restored. In order to express correct undoing, we record the value of `o2.d` prior to the update done by this transaction in the local variable `pre_start`. We then require that the very last value written by an aborting transaction be equal to the initial value kept in `pre_start`. Notice that, due to the possibly more than one entry in the log list for `o2.d`, it would be harder to express this property using either assertions or temporal logical formulations.

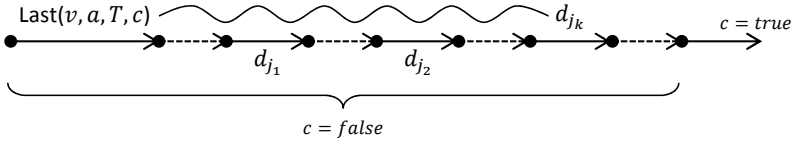
```
public void Foo(Xact Tx) {
 ...
 DTM.Start(Tx);            //Transaction Tx starts
 ...
 DTM.OpenForRead(Tx, o1);
 tmp = o1.d;
 tmp = tmp + 1;
 ...
 DTM.OpenForUpdate(Tx, o2);
 pre_start = o2.d;
 o2.d = tmp;

 tressa (Exit(success) ==> (o2.d == o1.d + 1);
 ...
 // At every later read of o1.d
 tressa (Exit(success) ==> (tmp == o1.d + 1));
 ...
 success = DTM.Commit(Tx); // Transaction Tx attempts to commit.
                           // success==true if it commits.
 if (!success) {
  done = false;
  tressa (Last(o2,Wr,{this},done)==pre_start);
  DTM.UndoUpdates(Tx);
  done = true;
 }
}
```

Fig. 4. Specifying properties in a code built on the Bartok STM

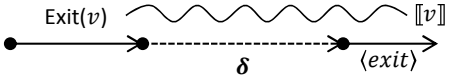
Semantics. We will describe the valuation of each term using diagrams. For the following, we say that a dynamic statement (t, st) *m*atches (v, a, T) if $t \in T$, st accesses v and the access type agrees with the access-type a.



$\mathsf{First}(v, a, T)$ denotes the value of the variable v after the first occurrence of a matching dynamic statement in the execution suffix. If the action sequence represented by δ above does not contain any dynamic statement that matches (v, a, T), the value of the term is the value written (or the value read) by d_j, as the wavy line suggests. If no such d_j exists, $\mathsf{First}(v, a, T) = \perp$.



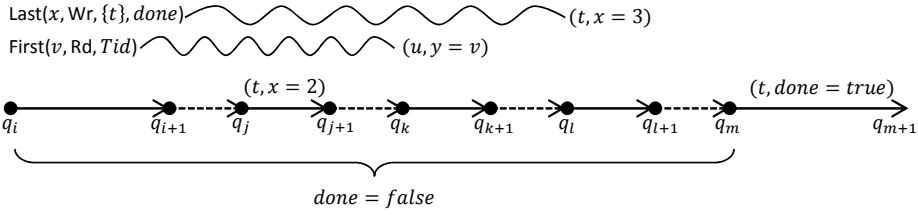
$\mathsf{Last}(v, a, T, c)$ denotes the value of the variable v after the last occurrence of a matching dynamic statement prior to c becoming true. In the diagram above, we assume that all d_{j_l} match (v, a, T) but only the very last one, d_{j_k}, determines the value of $\mathsf{Last}(v, a, T, c)$. If c stays false until termination or no matching statement occurs prior to c becoming true, $\mathsf{Last}(v, a, T, c) = \bot$.



$\mathsf{Exit}(v)$ denotes the value of the variable v immediately after the procedure p in which the term occurs terminates. In the diagram above, letting t denote the thread which executed the instruction containing $\mathsf{Exit}(v)$, we assume that the number of calls to p and that of returns from p executed by t are equal (recursion is allowed). If the execution blocks before p terminates, $\mathsf{Exit}(v) = \bot$.

A tressa claim is *ready* in a run if none of its terms evaluates to $\bot$. A program run *violates* $\mathsf{tressa}\,\phi$ if $\mathsf{tressa}\,\phi$ occurs during the run, the tressa claim is ready and ϕ evaluates to false. A program p *satisfies* a tressa claim tc if no instance of p has a run violating tc.

Let us further illustrate our formalization with the following example. Assume that ϕ is given as $\mathsf{First}(v, \mathsf{Rd}, Tid) = \mathsf{Last}(x, \mathsf{Wr}, \{t\}, done)$ and let $(t, \mathsf{tressa}\,\phi)$ be the dynamic statement executed at q_i.



In the above diagram, we assume that the dashed lines represent sequences of statements that match neither (v, Rd, Tid) nor $(x, \mathsf{Wr}, \{t\})$. At q_j, an update to x is done by t, but since this is not the last matching statement prior to *done* becoming true, its value is ignored. Then, at q_k, the dynamic statement that matches (v, Rd, Tid) occurs. The value of v returned by this read determines the value of $\mathsf{First}(v, \mathsf{Rd}, Tid)$ which we assume to be 5. At q_l, another update to x is done by t. According to the diagram, this is the last update to x by t prior to *done* becoming true at q_{m+1}. Thus, $\mathsf{Last}(x, \mathsf{Wr}, \{t\}, done) = 3$ in this run. Since all the terms of ϕ are determined by the execution segment ending at q_{m+1}, ϕ can be evaluated, which is found to be false. Thus, $\mathsf{tressa}\,\phi$ fails. It is important to note that we need the execution segment $q_i \ldots q_{m+1}$ to evaluate the tressa claim but the claim itself fails at q_{i+1}.

4 Run-Time Verification

In this section, we first describe the algorithms we use to check tressa claims over a given run. Then, we give an overview of the implementation of these algorithms.

4.1 Monitoring Algorithm

In this section, we explain how a tressa claim occurring in a given run can be evaluated. We present an algorithm that manipulates the tressa claims it observes throughout the course of an execution. We first explain what is being done at each transition, then delve into specific operations.

Algorithm 1. Monitoring Transitions

```
 1: procedure STEP(dstmt)
 2:     for all te ∈ TressaTable do
 3:         te.termSet ← EvalTerms(te.termSet,dstmt)
 4:     end for
 5:     for all pred ∈ ParseStmt(dstmt) do
 6:         InitTressa(pred)
 7:     end for
 8:     for all te ∈ TressaTable do
 9:         if te.termSet = ∅ then
10:             Check(te)
11:             Remove te from TressaTable
12:         end if
13:     end for
14: end procedure
```

Algorithm 1 shows the three phases that the monitor performs by each transition. It gets the label of the transition, the dynamic statement *dstmt*, as an input parameter. All the tressa claims which have been seen so far in the execution and whose value is not yet determined are kept in the *TressaTable*. In the first phase (lines 2-4), *TressaTable* is traversed and each term whose value is yet to be determined is analyzed and any of its terms that becomes determined is evaluated (line 3). In the second phase (lines 5-7), all the tressa claims that occur in *dstmt* are handled. In the third and final phase (lines 8-13), the tressa claims whose value can be calculated after this transition are found (line 9), their value is calculated (line 10) and is removed from *TressaTable* (line 11). We should point out that the implementation of this algorithm is slightly different where each variable points to the set of terms that are affected by the accesses to that variable. Then, instead of checking all live tressa claims, only those terms which depend on the accessed variables are checked. We used this alternative algorithm for ease of presentation.

Algorithm 2 shows how a new tressa claim is handled. Each entry in the *TressaTable* holds the predicate of the tressa claim and a set containing all the distinct terms of the predicate. These are assigned to a new entry, *tressaEntry*, at lines 2 and 3, respectively. Then, all the Val-terms are evaluated (lines 6-7) and each of these terms are removed from the set of undetermined terms (line 8). For all other term types, the value of the term is set to $\perp$ (line 9-10). For the Exitterms, we record the most recent value of the variable mentioned in the term (line 12-13). The resulting entry is inserted into the *TressaTable* (line 14). Observe that if a tressa claim has only Val-terms, its value is ready to be evaluated immediately after it occurs in the execution.

Algorithm 2. Initializing for a New Tressa

```
 1: procedure InitTressa(pred)
 2:     tressaEntry.pred ← pred
 3:     tressaEntry.termSet ← Parse(pred)
 4:     termSet ← tressaEntry.termSet
 5:     for all term ∈ termSet do
 6:         if IsVTerm(term) then
 7:             term.val ← EvalVTerm(term)
 8:             tressaEntry.termSet ← tressaEntry.termSet \ {term}
 9:         else
10:             term.val ← ⊥
11:         end if
12:         if IsETerm(term) then
13:             term.preval ← Eval(term)
14:         end if
15:     end for
16:     Insert tressaEntry into TressaTable
17: end procedure
```

Algorithm 3 shows how a set of terms is evaluated. It receives as input a term set, *termSet*, and a dynamic statement, *dstmt*. Each term in *termSet* initially has its value set to $\perp$, denoting undetermined value. We evaluate each term depending on its type and *dstmt*. For instance, if the term is a First-term and if *dstmt* is a dynamic statement which matches the term (line 4), the value of the term is evaluated and assigned to the term (line 5).[1] Similar checks and assignments are made for each type except for the Val type as their values were already evaluated when the tressa was initialized as in Algorithm 2. If the term's value is determined, then it is removed from the set of undetermined terms (lines 13-14). The algorithm returns the new set of undetermined terms (line 17).

Algorithm 2 has time complexity linear in the length of the tressa claim where the length of a tressa claim is given as the number of distinct terms the claim contains. Algorithm 3 has time complexity linear in the number of terms the parameter *termSet* contains because each call takes constant time. Algorithm 1 then has time complexity $O(size_{tt})$ where $size_{tt}$ is the sum of the lengths of all tressa claims observed throughout the execution. In terms of space complexity, the only non-constant complexity comes from EvalETerm which counts the number of pending calls in case of recursion. This introduces a logarithmic space complexity in the length of the execution.

4.2 Implementation

We start by giving an architectural overview of the implementation. We then highlight several implementation related issues.

[1] The call to EvalETerm is simplified. Due to recursion, we actually keep track of the recursion depth by counting the number of pending calls.

Algorithm 3. Term Evaluation

```
1: procedure EvalTerms(termSet,dstmt)
2:     cSet ← termSet
3:     for all term ∈ cSet do
4:         if IsFTerm(term) ∧ IsCompat(term,dstmt) then
5:             term.val ← EvalFTerm(term,dstmt)
6:         end if
7:         if IsLTerm(term) ∧ IsCompat(term,dstmt) then
8:             term.val ← EvalLTerm(term,dstmt)
9:         end if
10:        if IsETerm(term) ∧ IsCompat(term,dstmt) then
11:            term.val ← EvalETerm(term,dstmt)
12:        end if
13:        if term.val ≠ ⊥ then
14:            termSet ← termSet \ term
15:        end if
16:    end for
17:    return termSet
18: end procedure
```

Architecture. Figure 5 gives an operational description of our testing framework. In the first phase, the user annotates the test harness s/he wishes to verify with tressa claims according to a partial specification of correctness. The test harness is the input program wrapped with a specific test scenario. We currently accept programs written in C#, but in principle, any program written for the .NET framework can be easily handled.

In the second phase, we perform controlled executions of the given program. The annotated program is first processed by the Tressa library. The tressa library contains the code implementing the monitoring algorithm explained in Sec. 4.1. The output of this process, the original test harness along with its monitors, is fed into Chess. The Chess tool is responsible for two main tasks. First, it explores all possible interleavings of the `Test Input` (see below). Second, as each

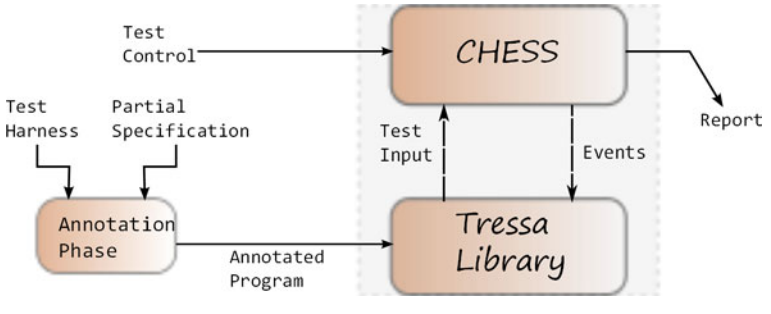


Fig. 5. The architectural diagram of the testing framework

interleaving is being executed, all memory accesses and method calls and exits are reported as events back to the Tressa library.

Tressa claims are evaluated as soon as they can be evaluated per the algorithm given in Sec. 4.1. If the tressa claim is satisfied, no further action is taken. Otherwise, the execution halts with a report specifying the failing tressa. CHESS can then be used to reproduce the failing execution for the analysis of the bug.

The CHESS Tool [4]. The use of CHESS is rather simple. The programmer specifies a particular test scenario on which s/he would like to check the outcome of her/his program. CHESS then runs the scenario so that all possible interleavings, up to a bound on the number of context switches specified by the programmer, are explored. It is well known that a concurrent program typically manifests its bug with only a few context switches over a few variables [4,5]. Thus, a small scenario with a context switch bound of two is likely to unravel subtle concurrency bugs. Case in point, the counter-example we gave in Sec. 2 for the buggy implementation (see Fig. 3), there is a single context switch per thread, there are four threads and two distinct addresses. Complete coverage of a set of interleavings reduces one degree of uncertainty about the outcome of a test-case. The user is still responsible for coming up with a scenario and a correct bound that would uncover the bug, but the additional uncertainty about whether the correct schedule would be chanced is removed.

Conceptually, CHESS achieves complete coverage by placing semaphores prior to all accesses to volatile variables or calls to synchronization methods from the `C# System.Threading` namespace such as the `Thread.Start()` method used for creating a thread or the `Thread.Join()` method used for waiting for a child thread to terminate. The places where semaphores are placed are the candidate context switching points. As CHESS explores a particular interleaving, each time a candidate context switching point is reached, exactly one semaphore is put in a non-blocking state which forces the scheduler to choose the desired thread. CHESS records down the set of interleavings it has already explored and as long as there remain unexplored interleavings, it resets the test harness, runs the program according to a new interleaving. A detailed explanation of the exact mechanism is beyond the scope of this paper.

The Tressa Library. The crux of the implementation lies in the Tressa library which implements the monitoring algorithm of Sec. 4.1. Tressa claims are generated by constructing a `Tressa` object whose constructor takes in its predicate in the form of a string. This string is then parsed into its constituent terms. Each term has a reference to the `Tressa` object it is part of. Additionally, for each of its terms, a handle denoting the term (its type and contents) and the variable referred to in the term is inserted into a global table.

An additional feature of CHESS has been crucial in the operation of the monitor implemented in the Tressa Library. As CHESS explores a particular execution, it keeps track of the relevant synchronization operations via an event generation mechanism. Each memory access, method call, method return and the like generate events which are then caught by CHESS in order to have complete control

over the execution of the `Test Input`. We are making use of these events to instrument each memory access as well as the method calls/returns so that the monitoring algorithm is called at every relevant transition. For instance, every time a variable is read, it generates an event which supplies the address accessed, the value read. Thus, accesses to variables are instrumented. As a variable is accessed, it is checked whether there are terms dependent on it by inspecting the global table for an entry for that variable. If the variable's entry is non-empty, for each term in its entry, a check is performed to see whether the term becomes determined. This step can be seen as a call to the STEP with the instrumented instruction as its input parameter, *dstmt*.

Following the algorithms given in Sec. 4.1, when a term becomes determined, it is removed from the variable's entry. For each term about to be removed, we also check whether the owning tressa claim has still undetermined terms. If all the terms of a tressa claim become determined, the truth value of the predicate is calculated. If the predicate evaluates to false, the violation along with the tressa claim causing it is reported. Thanks to the bug reproduction capability of the CHESS tool, the user can then trace the buggy execution to see what the cause for the violation is. Currently, the Tressa library is implemented in C#. We are using the `CLR` wrapper of CHESS, and the overall system is run under the `.NET` framework.

5 Experiments

In this section, we report our experience with an initial proof-of-concept implementation where instrumentation of memory accesses is done manually. We are currently working on automating the instrumentation and will provide a public release of the implementation.

We have tried our framework on three programs: the atomic snapshot implementation (see Sec. 2), a concurrent stack implementation [15] and a model of the Bartok STM [14]. We ran the examples on a Mac laptop with 1GB of RAM running at 2.8GHz.

In the atomic snapshot implementation, we have tried both the correct and the buggy versions on the test scenario given in Fig. 3. As expected, thanks to full coverage provided by CHESS, the bug was caught by our implementation after exploring 191 different schedules. The correct implementation generated no violations in 979 total schedules which is the total number of schedules with at most two context switches per thread.

For the concurrent stack, we ran a test scenario of three threads, two of them pushing five elements, the third popping five elements. In the `push` method we placed the tressa claim $\mathsf{tressa}\,\mathsf{First}(\mathtt{top}, \mathsf{Wr}, \{t\}) = \mathsf{Val}(n)$ which expressed the property that the first write by the thread t currently executing the method into the stack (`top`) is equal to the element with which `push` is called (n). Contrary to our expectations, the tressa failed after 113 schedules. The reason was due to an elimination round which bypasses pushing if there is a concurrently pending pop operation. This example highlights that even deceptively simple looking

tressa claims can be valuable tools in comprehending or debugging concurrent implementations. Later correcting this incorrect tressa claim removed the failure.

In the Bartok model, we uncovered a subtle bug by running a scenario with two threads, each running a transaction of at most two instructions. The bug was caught after 104 schedules, which takes about ten seconds, by the tressa claim $\mathsf{tressa}\ \mathtt{o.owner} \neq t \wedge \mathtt{o.IsOwned} \Rightarrow \neg\mathsf{Exit}(s)$. This claim, placed after a read of `o`, states that if `o` is currently owned by some other transaction, then this transaction should not commit. Corrected version was verified after completing all 112 possible schedules in approximately ten seconds.

References

1. Larus, J.R., Rajwar, R.: Transactional Memory. Morgan & Claypool (2006)
2. Sezgin, A., Tasiran, S., Qadeer, S.: Tressa: Claiming the future. In: VSTTE (2010)
3. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL 2009, pp. 2–15. ACM, New York (2009)
4. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Draves, R., van Renesse, R. (eds.) OSDI, pp. 267–280 (2008)
5. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: ASPLOS, pp. 329–339 (2008)
6. Pnueli, A.: The temporal logic of programs. In: FOCS 1977: Foundations of Computer Science, pp. 46–57 (1977)
7. Havelund, K., Goldberg, A.: Verify your runs. In: VSTTE, pp. 374–383 (2005)
8. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78, 293–303 (2009)
9. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 277–306. Springer, Heidelberg (2004)
10. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009)
11. Rosu, G., Chen, F., Ball, T.: Synthesizing monitors for safety properties: This time with calls and returns. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 51–68. Springer, Heidelberg (2008)
12. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM 40(4), 873–890 (1993)
13. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
14. Harris, T.L., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing memory transactions. In: PLDI, pp. 14–25 (2006)
15. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. J. Parallel Distrib. Comput. 70, 1–12 (2010)