Verifying Concurrent Programs using Separation Logic extended with Histories

Twan Coenraad
University of Twente
P.O. Box 217, 7500 AE Enschede
the Netherlands
t.coenraad@student.utwente.nl

ABSTRACT

Recently, an extension to separation logic has been proposed to make it simpler to verify the correct functioning of concurrent programs. Until now, one usually checks whether data races are absent using for example the de facto standard Owicki-Gries. However, verifying absence of data races does not scale, moreover functional properties cannot be verified. Therefore, histories are added to a program to take its state into account. Histories are updated if values are modified. Using this information, it can be verified that a method behaves as specified. In this paper, this proposed method is used to verify with VerCors some larger concurrent examples to find out if it is as simple as typically used techniques for data race checking are, while being able to verify even functional requirements of a method. It turns out that verifying those examples is doable, but at this moment not trivial. In the future this will be improved.

Keywords

separation logic, histories, concurrency, verification, VerCors

1. INTRODUCTION

Typically, created software will have unwanted behaviour during development and even when in production, often referred to as *bugs*. There are mainly two methods to cope with this. One is to create tests to make sure that, within set boundaries, the program works as expected. This is a straightforward approach which will work as long as it is accepted that any unforeseen state will have probably detrimental behaviour. It can be fine grained to every extent. By trying to cover most occurring edge cases, this could be sufficient and work well. In most programming languages there is also large support for this approach. However, by nature some edge cases will be missed and unpredictable behaviour will occur in those circumstances. This can be unwanted if this happens in software where life-threatening situations or financial risks are covered. The other approach to handle these bugs is to verify that the program works as intended. This is typically a more complex approach, given that you have to formalize the features a program should have, instead of testing specific behaviour in specific situations. Also, it is often less adaptable to a changing environment than tests are and requires thus more maintenance. Tool support for this is still under development, however the DSL[1] is improving and tool support is extended. Tools for verifying sequential programs are of production quality, e.g. CodeContracts[2] and OpenJML[3]. For concurrent programs, verification tools of beta quality exist, e.g. VerCors[4] and VeriFast[5].

For sequential programs, both solutions have been worked out and can be used in a professional context, albeit that verification is not as widely deployed as tests are. Until separation logic was developed, verification of programs running multiple threads did not scale, since every new thread had to be verified again together with the existing ones. This could be done by using for example de facto standard Owicki-Gries' method [6]. Next to this, most approaches focus on proving data-race freedom in a program [3]. This guarantees that critical sections are not hit simultaneously, but does not prove that the behaviour of the program is as intended. At any moment within a thread a shared variable can be modified by another one, therefore nothing can be guaranteed about the outcome. Verification is done modularly with separation logic and given every module is verified, scaling is not a problem any more. By taking also histories (which records all modifications in the history of a program) into account [8], scaling and guarantees about the outcome are taken care of.

In this paper, the history-based approach is reviewed to see whether it is as simple as claimed in [8] to verify concurrent programs. VerCors is a tool with support for these histories, but it is not yet known to what extent it can be used to verify complicated pieces of (concurrent) software. Therefore, we will use VerCors as our verification tool. We start by taking a basic counter to show how to decorate and annotate code with separation logic and histories. This example is then extended to show what is possible with this technique. For example, it can be tried to make the example include loops and to make it work concurrently. These are relatively simple programming structures that can be used to compose more complex programs. After verifying both, some simple (sorting) algorithms can be validated. All of this will be documented as a handbook on how to verify programs with separation logic and histories using VerCors.

1.1 Related work

23th Twente Student Conference on IT June 22st, 2015, Enschede, the Netherlands.

[1] Domain Specific Language
[2] See `https://github.com/Microsoft/CodeContracts`.
[3] See `http://openjml.org/`.
[4] See `http://fmt.ewi.utwente.nl/puptol/vercors-verifier/`.
[5] See `http://people.cs.kuleuven.be/~bart.jacobs/verifast/`.

Some similar case studies have been done for other verification tools and techniques. For Owicki-Gries' method a systematic exploration has been done and published by Apt [1]. For VeriFast Jacobs tested and proved [5] that it verifies programs and structures like a chat server, linked list and iterator, composite, a JavaCard applet and a GameServer. In another published paper by Philippaerts, the Belgian Electronic Identity Card and Policy Enforcement Point for Network Admission Control scenarios were extensively validated [7]. Also, for VeriFast a tutorial is written that explains in detail how it is used [4]. For VeriFast this were all verifications on sequential programs. This paper focuses on verifying concurrent programs using separation logic extended with histories, written as a small step-by-step guide.

2. RESEARCH OBJECTIVES

It is expected in that separation logic requires the same effort to verify a program as for data race checking [8]. The first method does not state anything about the result it has, though the latter method can be used to make guarantees about the functional behaviour. It is however unknown how difficult it is to specify and verify a program with this approach. Therefore, this research will focus on the ease of use of the latter method.

How well can one, within a few weeks and without knowledge on separation logic, verify several real life examples using VerCors with permission-based separation logic extended with histories?

2.1 Research questions

1. How does one specify and verify loops in VerCors?
2. How does one specify and verify a concurrent program in VerCors?
   (a) ... that is deterministic?
   (b) ... that is non-deterministic?
3. How does one specify and verify a sorting algorithm in VerCors?

3. BACKGROUND INFORMATION

3.1 Permissions

To reason about the partial correctness of the behaviour in (concurrent) programs, permission-based separation logic is used [8]. Code is separated in atomic blocks. When a shared memory location is accessed, a fractional permission (denoted as $\pi \in (0, 1]$) is required. To write, a permission of 1 has to be held, otherwise the method is read-only and a fraction of the permission is taken, thus making it impossible for other methods to write that location.

3.2 Actions

Actions are used to describe changes of values, which are observed by the environment as events happening at once, thus *atomically*. A history is used to record these actions. This history is shared between concurrent threads.

3.3 Example

In a sequential context, no histories are needed to prove that a counter is incremented by exactly 1. However, when used in a concurrent context, histories are needed to reason about the outcome, to take interleavings into account. To verify a specification in VerCors, a language based on JML [6] is used. Similar to JML, annotations are used to specify blocks of code. How both are applied to Listing 1 is shown below.

```
1 void doIncr() {
2   x = x + 1;
3 }
```

Listing 1. Simple counter example

3.3.1 *Definitions*

First, definitions have to be declared to describe any method with a minimal set of actions and processes. Actions are abstract methods that only describe behaviour, processes do have implemented behaviour (albeit that they both are using keyword *process* in VerCors). Listing 2 declares x as an integer, which makes it possible to reason about a certain x.

```
1 int x;
```

Listing 2. Declaration of a variable

Next, in Listing 3 a process is declared which prescribes the behaviour of the `incr()` method.

```
1 modifies x;
2 ensures x == \old(x) + 1;
3 process incr();
```

Listing 3. Process incr()

3.3.2 *Decorating*

Now all is set to decorate Listing 1 and to specify its behaviour for verification with VerCors.

```
1 given process p;
2 given frac q;
3 requires Hist(h, q, p);
4 requires Perm(h.x, 1);
5 ensures  Perm(h.x, 1);
6 ensures  Hist(h, q, p * h.incr());
```

Listing 4. Pre and post conditions doIncr()

Firstly, we explain the pre and post conditions in Listing 4. Given a certain state and fraction (as explained in Section 3), we start by requiring a certain history to add actions on (line 3). Next to that, a write permission (thus, the whole fraction) is needed (line 4). Afterwards, the permission must still hold and the history must be extended (using the $*$ operator) with a *incr*() action (line 5).

Now the method's body can be annotated[7].

```
1 void doIncr(History h) {
2   {
3     //@ action h, q, p, incr();
4     x = x + 1;
5   }
6 }
```

Listing 5. doIncr()

Within the body of *doIncr*() a block is defined (lines 2−5), with an `@ action` comment, making it an action block for

[6] Java Modelling Language

[7] At this moment it is needed to add a history to the method's signature as shown in line 1 of Listing 5, but this is likely to be removed in the future.

the verifier. The action that is defined, can be read as given a history, permission and process, one single *incr*() is added to the history. The only execution line is 4, where x is incremented by 1.

Combining all listings above, the program can be successfully verified using VerCors.

4. RESEARCH

Given the simple counter as build in Section 3.3, we now continue building new structures. In the end we will have all structures available to combine them into one program. That program will have two concurrent threads, modifying one single variable, which is protected with a lock. This example will be fully verifiable by VerCors.

4.1 Loops

Verifying a simple assignment once is now done, but repeating such statements is a common programming pattern. Those loops are usually build with *for* or *while* keywords and a corresponding block. To verify that a loop is correct, a loop invariant has to be found that is valid before, during and after the loop. This, together with an action that alters the history within the loop, makes it possible to ensure that a loop is working correctly.

4.1.1 Example

Instead of incrementing i once, we now do it n times, see Listing 6.

```
1 void doIncrLoop(int n){
2   int i = 0;
3   while (i < n){
4       x = x + 1;
5       i = i + 1;
6   }
7 }
```

Listing 6. doIncrLoop()

To start, the header of the previous example can be reused, with an added *requires* in the top and a replaced *ensures* at the bottom.

```
1 requires n >= 0;
2 (... pre and post conditions as doIncr()
      ...)
3 ensures Hist(h, q, p * h.loop(n));
```

Listing 7. doIncrLoop()'s pre and post conditions

A requirement about n is added (line 1). Also, the post condition on history is changed, since history is extended with *loop*(n) (line 3). *loop*() is a new process that alters the history n times with an *incr*() step, which results in x incremented by n.

```
1 modifies x;
2 requires n >= 0;
3 ensures  x == \old(x) + n;
4 process  loop(int n) = n > 0 ? loop(n - 1) *
      incr() : empty();
```

Listing 8. loop()

As stated before, a loop invariant has to be declared, which is evaluated after each iteration of the loop. Typically it limits the iterator and ensures that after each iteration the state is changed accordingly. In this example, $0 \leq i \leq n$ holds (thus including the case that $i == n$). Next to this, the *ensures* statement holds, till the i^{th} step. Both are concatenated by using the $**$ operator.

```
1 loop_invariant 0 <= i ** i <= n ** Perm(h.x,
      1) ** Hist(h, q, p * h.loop(i));
```

Listing 9. loop invariant

At last, history needs to be changed. The action that states this reads as: given a history, permission and process extended with *loop*(i), one single *incr*() is added to the history.

```
1 action h, q, p * h.loop(i) , h.incr();
```

Listing 10. Loop invariant

The complete annotated code can be found in the source code published together with this paper [2]. VerCors verifies this code successfully. Unfortunately, at this time of writing, *for* loops are not supported. However, this can be ignored since every *for* loop can be rewritten to an equivalent *while* loop.

4.2 Threads with deterministic and separated behaviour

4.2.1 Example

We now start by observing an example that uses threads with deterministic behaviour. We define two workers that increment both their own counter.

```
1  public class Worker extends Thread {
2    private int input;
3    public Worker(int input) {
4      this.input = input;
5    }
6    public void run() {
7      output = input + 1;
8    }
9    public int getOutput() {
10     return output;
11   }
12 }
```

Listing 11. Worker class

The workers defined in Listing 11 can be run by code described in Listing 12.

```
1 int i = 0; int j = 1;
2 Worker w1 = new Worker(i);
3 Worker w2 = new Worker(j);
4
5 w1.start();
6 w2.start();
7
8 try { w1.join(); w2.join(); } catch (
      InterruptedException e) {}
```

Listing 12. main()

This code is verified by VerCors in several steps. At this moment, some Java syntax is unsupported, so we have to drop the *try/catch* block and add the exception to the *throws* block in the method's declaration.

As explained in Section 3, verification is done modularly. After all modules are created, we compose them such that the above-mentioned example can be verified. Therefore, some workarounds have been made. We start by adding pre and post conditions to threads on the $Worker$, named

preFork() and *postJoin*(). These are roughly equivalent to the *requires* and *ensures* as were given in the previous examples 3.3.

```
1 public resource preFork(frac p) = Value(input
      ) ** p != none ** Perm(output, p);
2 public resource postJoin(frac p) = Value(
      input) ** p != none ** PointsTo(output, p
      , input + 1);
```

Listing 13. Resources

Using this syntax, *preFork()* and *postJoin*() can be inherited and applied to instances of *Worker* as we will see in the next section. The precondition *preFork()* tells us that we have a read permission on *input*. *Value*(*x*)[8] ensures that we have read permission over *p*. When the method is called, a fraction *p* is given. This should not be empty and it should give us permission to work on *output*. Afterwards, the post condition *postJoin*() tells that, given a permission on fraction *p*, *output* = *input* + 1 using the *PointsTo*() shorthand[9].

The constructor of *Worker* needs a post condition with the statement that given permission to write to *input*, *input* is set. Also, a *preFork*(1) is ensured to have enough permissions to fulfil the precondition of *start*(), as expected by *Thread*.

```
1 //@ ensures preFork(1) ** Value(input) **
      this.input == input;
2 public Worker(int input){
3   this.input = input;
4   //@ fold this.preFork@Thread(1);
5   //@ fold this.preFork@Worker(1);
6 }
```

Listing 14. Thread example's worker's constructor

Within the constructor, two folds of *Thread* and *Worker* are done to tell the verifier that it substitutes the right-hand side of *preFork()* with the left-hand side. This is needed to comply to the post condition and the *preFork()* of *Thread* is needed as *Worker* depends on it.

The *run*() method only needs an *unfold* (replacing the left-hand side by its right-hand equivalent) of *preFold* on *Worker* to be able to verify the increment line. *preFold* gives the sufficient permission to alter *output* based on *input*. *Thread* requires then to have a *postJoin*() on *Thread*, which depends on Worker to be *postJoin*()ed as well and thus are both folded (replacing the right-hand side by its left-hand equivalent).

```
1 //@ unfold preFork@Worker(1);
2 output = input + 1;
3 //@ fold this.postJoin@Thread(1);
4 //@ fold this.postJoin@Worker(1);
```

Listing 15. Thread example's worker's run()

We start by running just one *Worker* in the *main*() method. At this moment, because of technical limitations, the *try/catch* block of the *main*() method needs to be removed and exceptions are thus elevated by a *throws* declaration. After *start*() and *join*() have run, the join event is also added for program's verification.

```
1 w.join()/*@ with { p = 1; } @*/;
2
3 //@ assert w.output == 7;
4 //@ open w.postJoin@Worker(1);
5 //@ unfold w.postJoin@Worker(1);
6 //@ assert w.output == 8;
```

Listing 16. Post join specifications

First, we fill the parameter *p* (in Listing 16, line 1), that is specified in *Thread*. The two *asserts* check that the behaviour of the worker is as intended. In between there are an *open* and an *unfold* action. *open* limits the scope of *Worker*[10], to make it possible to *unfold* it and reason about the output state, see listing 13.

The advantages of the modular technique now become visible. To extend this example and reason about two workers, working simultaneously, we only need to double the *asserts*, *open* and *unfolds* in *main*() as follows and it just verifies as well.

```
1  Worker w1 = new Worker(7);
2  Worker w2 = new Worker(8);
3  //@ assert w1.input == 7;
4  //@ assert w2.input == 8;
5  w1.start();
6  w2.start();
7  w1.join()/*@ with { p = 1; } @*/;
8  w2.join()/*@ with { p = 1; } @*/;
9  //@ assert w1.input == 7;
10 //@ assert w2.input == 8;
11 //@ open w1.postJoin@Worker(1);
12 //@ open w2.postJoin@Worker(1);
13 //@ unfold w1.postJoin@Worker(1);
14 //@ unfold w2.postJoin@Worker(1);
15 //@ assert w1.output == 8;
16 //@ assert w2.output == 9;
```

Listing 17. Thread example's main()

4.3 Using a lock

4.3.1 Example

Regarding the goal set, locks are needed to make sure a variable is not modified at the same time. Typically this looks like the code in Listing 18[11].

```
1 Lock lock = new ReentrantLock();
2 lock.lock();
3
4 try {
5   x = 35; // critical section
6 } finally {
7   lock.unlock();
8 }
```

Listing 18. Lock example

A template implementation for locks is given. Now, we will show how this is incorporated into the previous threading example. At first, the *Worker* is extended. We use a *SubjectLock* object which contains a shared variable and is a lock at the same time. *SubjectLock* is derived from

[8] `Value(x) = ∃ frac p * Perm(x, p)`
[9] `PointsTo(x, frac, value) = Perm(x, frac) ** x == value`
[10] Inheritance makes it possible for *w* to be a *Worker* or one of its potential subclasses at runtime. Predicates are extended by inheritance and not overridden by subclasses, as is usual for e.g. methods. By using *open*, predicates are not extended by possible subclasses that *w* can be at runtime, but limited to the specified class, i.e. *Worker*.
[11] As *try/finally* blocks are not supported yet in VerCors, the re-entrant lock is replaced by a simpler variant.

LockTemplate which is part of VerCors' source code. *preFork()* and *postFork()* from Listing 13 have to be altered.

```
resource preFork(frac p) = p == write **
    Value(this.l) ** Value(this.l.subject) **
     Value(this.s) ** this.l.subject == this.
    s ** [1/4](this.l.valid());
resource postJoin(frac p) = p == write **
    Value(this.l) ** Value(this.l.subject) **
     Value(this.s) ** this.l.subject == this.
    s ** [1/4](this.l.valid());
```

Listing 19. Lock example's resources

Both methods do now need a write permission to *preFork()* and to *postJoin*(). Next to that, we are going to alter s, therefore we need permission on l, $l.subject$ (which is s) s itself and make sure that $l.subject$ equals s. At last, a $\frac{1}{4}$ is given as argument[12] on *valid*() as is required later on by *lock*(). Every value for p is okay, as long as its total sum (i.e. amount of threads $\cdot$ p) is less or equal to 1.

Then the constructor's pre and post conditions are altered likewise. What it specifies speaks for itself.

```
//@ requires Value(l.subject) ** l.subject ==
     s ** [1/4](l.valid());
//@ ensures  Value(this.l) ** this.l == l;
//@ ensures  Value(this.s) ** this.s == s;
//@ ensures  preFork(1);
public Worker(Subject s, SubjectLock l) {
  this.l = l;
  this.s = s;
  //@ fold this.preFork@Thread(1);
  //@ fold this.preFork@Worker(1);
}
```

Listing 20. Lock example's constructor

In the *run*() method two *Lock()/unfolds* and *unlock()/folds* are added respectively above and below the critical section to have the right to alter s's value (stored as $s.x$).

```
//(... folds as in threads example...)
l.lock()/*@ with { p=1/4; count=0; }@*/;

//@ unfold l.inv();
//@ unfold l.subject.inv();

s.x = 35; // critical section

//@ fold l.subject.inv();
//@ fold l.inv();
l.unlock()/*@ with { p=1/4; count=1; }@*/;

//(... unfolds as in threads example...)
```

Listing 21. Lock example's worker's run()

l.subject.inv() refers to the invariant `resource inv() = Perm(x, 1)` to be able to alter x's value. As l is a wrapper for *subject*, *l.inv*() refers to a simple `resource inv() = Value(subject) ** subject.inv()`, thus giving us permission to get the invariant on *subject*.

Then, *main*() is altered similarly. Also here *folds* and *unfolds* are applied to *subject* and *lock*.

```
Subject s = new Subject();
//@ fold s.inv();
SubjectLock lock=new SubjectLock(s);
//@ fold lock.inv();
lock.commit();

Worker w1 = new Worker(s, lock);
//(... see code in threads example ...)
//@ unfold w2.postJoin@Worker(1);

lock.uncommit();
//@ unfold lock.inv();
//@ unfold s.inv();
```

Listing 22. Lock example's main()

The added *commit*() and *uncommit*() are required by *SubjectLock* and ensure that within *Worker*, *lock*() and *unlock*() can be called.

4.4 Combining histories, threads and locks

Although we now have incorporated locks into our (single, yet threaded) example, nothing can be asserted about *Subject*'s s in the post condition after *uncommit*(). This is because of the lose of full control, the very moment *uncommit*() takes place. Therefore, histories are added, very similar as done in Section 3.

Again, we alter the needed resources.

```
resource preFork(frac p) = p == write **
    Value(this.l) ** Value(this.l.subject) **
     Value(this.s) ** this.l.subject == this.
    s ** ([1/2]this.l.valid()) ** Hist(s,
    1/2, empty);
resource postJoin(frac p) = p == write **
    Value(this.l) ** Value(this.l.subject) **
     Value(this.s) ** this.l.subject == this.
    s ** ([1/2]this.l.valid()) ** Hist(s,
    1/2, s.incr(1));
```

Listing 23. Combined example's resources

Both pre and post conditions are extended with a history. As before, *preFork()* indicates that we start with an empty history which is filled with an *incr*() action afterwards in *postJoin*().

Now the constructor is changed to receive a *History* as if it were a *Subject* in the previous example. Moreover, the pre condition requires to start with an empty history.

```
//@ requires Value(l.subject) ** l.subject ==
     s ** ([1/2]l.valid()) ** Hist(s, 1/2,
    empty);
//@ ensures Value(this.l) ** this.l == l;
//@ ensures Value(this.s) ** this.s == s;
//@ ensures preFork(1);
public Worker(History s, SubjectLock l) {
  this.l = l;
  this.s = s;
  //@ fold this.preFork@Thread(1);
  //@ fold this.preFork@Worker(1);
}
```

Listing 24. Combined example's constructor

Next, the critical section is extended to write to history as well. Nothing new is introduced here.

```
//@ assert Hist(s, 1/2, empty);
{
  //@ action s, 1/2, empty, s.incr();
  s.x = s.x + 1;
}
```

[12] `[p]Perm(x, q) == Perm(x, p*q)`

```
6 //@ assert Hist(s, 1/2, s.incr(1));
```

Listing 25. Combined example's worker's critical section in run()

At last program's *main*() is extended to have a *History* as argument for both *Worker*s and have that history be shared equally.

```
1  History s = new History();
2  s.x = 35;
3  //@ create s;
4  //@ split s, 1/2, empty, 1/2, empty;
5  //@ fold s.inv();
6  SubjectLock lock = new SubjectLock(s);
7  //@ fold lock.inv();
8  lock.commit();
9
10 Worker w1 = new Worker(s, lock);
11 Worker w2 = new Worker(s, lock);
12
13 //(... see code in threads example ...)
14
15 //@ unfold s.inv();
16 //@ merge s, 1/2, s.incr(1), 1/2, s.incr(1);
17 //@ destroy s, s.concurrentIncr(1, 1);
18 //@ assert s.x == 37;
```

Listing 26. Combined example's main()

On line 3 of *main*() a history is created, to split it in half (on line 4) and give it as argument to both workers. On split, it is specified how the split histories look like. Here they are both (still) empty. After *unfold* histories are merged and thereafter history is destroyed to push all changes into *s*. An assertion is set optionally to show that the right behaviour was captured by VerCors and processed.

```
1 modifies x;
2 requires n >= 0 && m >= 0;
3 ensures  x == \old(x) + n + m;
4 process  concurrentIncr(int n, int m) = incr(
      n)  incr(m);
```

Listing 27. concurrentIncr()

When destroying the history, *concurrentIncr*() is invoked. This is a small process, that like *incr*() increments a variable, yet then concurrently using the || operator.

Also, this example fully verifies in VerCors.

5. RESULTS AND DISCUSSION

Earlier, the following questions were formulated:

How well can one, within a few weeks and without knowledge on separation logic, verify several real life examples using VerCors with permission-based separation logic extended with histories?

5.1 Research questions

1. How does one specify and verify loops in VerCors?
2. How does one specify and verify a concurrent program in VerCors?
   (a) ... that is deterministic?
   (b) ... that is non-deterministic?
3. How does one specify and verify a sorting algorithm in VerCors?

One of the main issues of this research was the understanding of the new technique. It turned out that separation logic with histories is on a high level quite simple to understand. With only a few insights on how it should be applied, one can grasp this technique quite simply. Also, to see what advantages this new technique has over Owick-Gries' one are apparent. The modular, scalable way of composing programs works quite intuitively and as shown in the last example (see Section 4.4), almost no customization is needed to comply to the requirements of VerCors.

Specifying and verifying loops in VerCors turned out to be as well quite simple and can be found in Section 4.1. The examples found in the source code, together with some tinkering made it very clear on how the verifier does its job on this very small example. Some additional questions towards the author of this VerCors were asked to learn about the exact workings. It stand out that, at this moment, VerCors is a tool that needs a lot to know about what is going on in the program, next to the program itself. This results into a need to know lots about the inner working of VerCors. Given little domain knowledge, this is undesirable and could be largely improved by deriving most of now manually defined actions from source code. A smaller footprint in specifications makes it not only better to get a grasp on VerCors, but also removes the need to write lots of boilerplate code.

Specifying and verifying concurrent programs turned out to be unsupported by VerCors at moment of beginning research. Support for it was built quiet quickly. However, some breaking syntax changes were made, resulting in a redo of e.g. the loop example in Section 4.1. This frustrated making great strides forward, although at the same time some some boilerplate code became superfluous, as it was already derived from source code, which relieved. At this moment, some common programming patterns (e.g. *for* loops, *try/catch* blocks) are not supported. This made it necessary to alter execution code, to make it verifiable for the tool. Altering does not take a lot of time, but should be unnecessary. With this take in mind, it was possible to verify a deterministic program as shown in Sections 4.3 and 4.4. Unfortunately, non-deterministic programs or sorting algorithms were not specified and verified in VerCors at all.

In the future, the above-mentioned programming patterns will likely be added. Now, the current lack of a library in VerCors obscure what user code is and what tool code is. This makes it difficult to learn yourself how to use VerCors. It would be easier to use VerCors when you only have to comply to some (common) method contracts. At this time, a lot of tool knowledge is required to verify larger structures. This makes it difficult to come up with a verification statement yourself. Addition of a library could most certainly fix this. That library should be documented well, as it makes it possible to not have to rely on given examples, but to reason about the problems and come up with your own solution.

6. CONCLUSION

In this research, separation logic extended with histories was examined. We wanted to see if this technique is simple enough to understand when you have little domain knowledge. We have seen that the technique itself is simple enough to understand. However, the verification tool VerCors that has support for this approach is more diffi-

cult to get to get the hang of. It was found that the small examples that do come with VerCors are a good starting point. Based on these examples, it is doable to find out yourself how the tool works in these situations by tinkering and extending them eventually a little bit. However, when trying to verify more complicated, yet common programming patterns, it turns out that some of theses patterns are not fully supported at this moment. In some cases this was fixed by applying small workarounds and rewriting code. It is difficult to say how well real programs are supported. In a way, all examples described in this paper do verify using VerCors. On the other hand, what is shown is just the tip of the ice berg and in the future even more difficult structures must be be verified. Moreover, the program lacks proper documentation at this moment. This makes it hard to find out yourself how you should decorate a program to verify it. By adding all commonly used patterns, the tool will probably be experienced as developer friendly and thus usable for people who do not have great knowledge of permission-based separation logic and histories. Then the advantages of using separation logic with histories are clear.

7. FUTURE WORK

We have seen how to verify loops with histories and how to verify threads that run simultaneously on different variables. Thereafter, locks were introduced, which were useful in an example where one variable was altered simultaneously by two different threads. Moreover, all has been documented how all is built from scratch and how the library parts of the source code are incorporated into our own code.

In future work this example can be extended even further. For example, a producer consumer example can be build, beginning with one producing thread that puts its output in a shared (un)bounded queue or stack and another consuming thread that pops values. At first, this can be done in a busy-waiting fashion, but it would be even better to use the *wait/notify* pattern. Also other common concurrent structures like *conditions*, *barriers* and *volatiles* can be verified. Later, more producer and consumers can be added and if all is set-up right, this should be a formality, like the example in Section 4.4) showed us. Piece by piece all common patterns in multi-threaded software should be elaborated and documented as well. Next to these specific concurrent structures, also ordinary *try/catch* blocks, exception handling and others should be added to the tool kit. It is acceptable when much boilerplate is needed at first, as long as it is reduced later on. When all common programming patterns are supported and documented well, it will be much easier to verify any multi-threaded program using the separation logic extended with histories.

8. REFERENCES

[1] K. R. Apt, F. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer Publishing Company, Incorporated, 3rd edition, 2009.

[2] S. Blom and T. Coenraad. Source code belonging to Verifying Concurrent Programs using Separation Logic extended with Histories. `http://fmt.ewi.utwente.nl/education/bachelor/230/`.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[4] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In K. Ueda, editor, *Programming Languages and Systems*, volume 6461 of *Lecture Notes in Computer Science*, pages 304–311. Springer Berlin Heidelberg, 2010.

[6] S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Ithaca, NY, USA, 1975. AAI7612884.

[7] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82(0):77 – 97, 2014. Special Issue on Automated Verification of Critical Systems (AVoCS'11).

[8] M. Zaharieva-Stojanovski, M. Huisman, and S. Blom. Verifying functional behaviour of concurrent programs. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, FTfJP'14, pages 4:1–4:6, New York, NY, USA, 2014. ACM.