Verifying Functional Behaviour of Concurrent Programs

Marina Zaharieva-Stojanovski
University of Twente
the Netherlands

Marieke Huisman
University of Twente
the Netherlands

Stefan Blom
University of Twente
the Netherlands

ABSTRACT

Specifying the functional behaviour of a concurrent program can often be quite troublesome: it is hard to provide a *stable* method contract that can not be invalidated by other threads. In this paper we propose a novel modular technique for specifying and verifying behavioural properties in concurrent programs. Our approach uses history-based specifications. A history is a process algebra term built of *actions*, where each action represents an update over a heap location. Instead of describing the object's precise state, a method contract may describe the method's behaviour in terms of actions recorded in the history. The client class can later use the history to reason about the concrete state of the object.

Our approach allows providing simple and intuitive specifications, while the logic is a simple extension of permission-based separation logic.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*correctness proofs, formal methods, programming by contract*

General Terms

Verification

Keywords

concurrency, modular verification, histories, separation logic

1. INTRODUCTION

Verifying program correctness means proving that the program behaves as described by its formal specification. In a concurrent program, an inconsistent behaviour may occur due to thread interleavings and potential data-race conditions. Existing techniques for verifying concurrent software often focus on proving data-race freedom in a program [4, 12, 3]. Although this is an essential property for a concurrent program, it does not guarantee that the program *behaves* as the programmer expects. In practice, specifying and verifying the functional behaviour of a concurrent program in a modular way is often quite challenging.

We illustrate this by an example: Lst. 1 shows a simple shared data structure, a class *Counter*. The *increase*() method is implemented correctly and is data-race free, as the shared location *data* is protected by a lock. However, because synchronisation happens inside the method (*internal synchronisation*), it is difficult to describe the behaviour via the method contract. An expression $data = \backslash old(data) + 1$ is not an acceptable postcondition in a concurrent setting, because the value of *data* is unstable after the lock release. As a result, method contracts in scenarios like this often do not fully express the method behaviour, which also limits proving properties for the client class that uses the data structure. If the *Client* object in Lst. 1 creates a *Counter* object c with initial value $c.data = 0$, and then forks two parallel threads, each of them increasing $c.data$ by 1, we can not prove in a modular way that after joining both threads, the value of $c.data$ is 2.

The same example was presented in 1976 in the work of Owicki and Gries [16], where they propose the first formal method for verifying parallel programs. To verify this program, they use *auxiliary (ghost)* variables, which however, break modularity. Modular reasoning about programs with internal synchronisation is still a challenge.

```
class Counter {
  int data;
  Lock lock;
//... constructors

//postcondition = ... ?;
 void increase(){
   lock.lock();
     data=data+1;
   lock.unlock();
 }
}
class Client{
 //   ...
 Counter c = new Counter(0);
 t1.fork(); t2.fork();  //both threads t1 and t2 call c.increase()
 t1.join();  t2.join();
}
```

Lst. 1: A shared Counter data structure

In this paper we develop a new method for modular rea-

FTfJP '14, July 29 2014, Uppsala, Sweden

http://dx.doi.org/10.1145/2635631.2635849

soning about partial correctness of behavioral properties in concurrent programs. Our logic is based on *permission-based separation logic* [3], while the specification language is based on JML (Java Modeling Language) [13]. We target programs with *internal synchronisation*, as the example in Lst. 1.

The general idea of the approach is the following. We introduce *actions* as part of the specification language: an action over a heap location x describes a change of the value of x and is observed by the environment as an atomic change. For example, the action for incrementing an integer value by 1 may be specified as:

$$\mathsf{action}\ inc\,[int\ x] \quad \equiv \quad \backslash old(x) + 1$$

When specifying the precise value of a location x in the method post-state is difficult (as in Lst.1), the programmer may specify the behaviour of the method in terms of actions over x executed within the method. Every action over x is recorded in a history of changes H associated to x. In particular, every heap location x is associated with a Hist predicate, which stores a history H (modelled as a *process algebra term* [8]) in which all actions over x are recorded.

The history predicate Hist is a *splittable token* and thus, may be shared among several parallel threads. Each thread is responsible for recording its local changes in the owned part of the token. When all threads have finished their updates, the client class may collect all token parts and merge all changes recorded by all threads. We can then reason about the new value (or the set of possible values) of x.

The *Counter.increase*() method may be specified as:

```
//@ requires Hist(data, π, V, H);
//@ ensures  Hist(data, π, V, H · inc),
```

where *inc* is the action specified above. The method contract describes only the local changes in the history: the current thread has increased the value of *data* by 1.

The main contribution of the paper is a novel methodology for modular verification of behavioural properties in concurrent programs. The problem addressed in the paper is very common in numerous concurrent programs. Importantly, the approach that we introduce is rather straightforward: it allows providing simple and intuitive specifications; the logic that we propose is a simple extension of permission-based separation logic. We are working on integrating this technique in the VerCors tool set [2, 1].

Outline We give a short overview of the process algebra theory in Sec. 2 and permission-based separation logic in Sec. 3. Further, in Sec. 4 we present our approach for reasoning about concurrent programs. In Sec. 5 we compare our work with other existing approaches and we discuss future plans.

2. ALGEBRA OF COMMUNICATING PROCESSES

The *algebra of communicating processes (ACP)* [8] is a mathematical approach for reasoning about system behaviour in terms of algebraic process expressions. The basic primitives in ACP are *actions* from the set $A = \{a, b, c, ...\}$, each of them representing an indivisible process behaviour. To describe various processes $\{p_1, p_2, ...\}$, actions are combined using algebraic operators, the most fundamental of which are the *sequencing composition* $(\cdot)$ and the *alternative composition* $(+)$. For example, the expression $a+(b\cdot c)$ expresses a process composed of an action a *or* a sequence of actions b *and* c. Further, two special actions are used: the *deadlock action* δ and the *silent action* τ (an action without behaviour). We have: $\delta \cdot p = \delta$, $\delta + p = p$ and $\tau \cdot p = p$.

Parallel composition of two processes is described by the binary *merge operator* $(\|)$, i.e., an alternative composition of all possible interleavings between both processes: $p_1 \parallel p_2 = (p_1 \between p_2) + (p_2 \between p_1) + (p_1 \mid p_2)$. The operator $\between$ is the *left merge operator*, which describes a parallel composition of two processes where the initial step is always the first action of the left-hand operator: $(a \cdot p_1) \between p_2 = a \cdot (p_1 \parallel p_2)$. The *communication merge* $(|)$ expresses a parallel composition of two processes where the first step is a communication between the first actions of each process: $a \cdot p_1 \mid b \cdot p_2 = a \mid b \cdot (p_1 \parallel p_2)$. For atomic actions, the *communication function* $(|)$ is defined through the function $\gamma : A \times A \mapsto A$: $a \mid b = \gamma(a, b)$. In Sec. 4.2 we show how we use the *communication function* to provide synchronisation between processes.

3. PERMISSIONS, FRAMING, STABILITY

Separation Logic and Permissions.

Permission-based separation logic [17, 15] is a program logic (an extension of Hoare Logic [10]) used to reason about multithreaded programs. Every access to a heap location is associated with a fractional permission π, i.e., a value in the domain $(0, 1]$ [3, 5]. At any point in time, a thread might hold a permission to access a location. To change a location x, a thread must hold a *write* permission for x, i.e., $\pi = 1$; while for reading a location, any *read* permission is required, i.e., $\pi > 0$. The soundness of this logic ensures that the sum of all threads' permissions for a certain location never exceeds 1, which guarantees that a verified program is data-race free.

The basis of this logic is the *separating conjunction* operation $(*)$: $P{*}Q$ states that P and Q hold for *disjoint* parts of the heap and thus, may be used by two parallel threads. Furthermore, the *separating implication* or *magic wand* $P{-\!\!*}Q$ asserts that, if the current heap is extended with a disjoint part satisfying P, then Q holds for the extended heap.

Permission for a location x is expressed via the predicate $\mathsf{PointsTo}(x, \pi, v)$, which indicates that x points to a location for which the thread has a permission π, and the value of x is v. Proof rules for writing and reading are described by the following Hoare triples (where " $?u$ " means *any value* and we name this value "u"):

$$[Write] \quad \{\mathsf{PointsTo}(x, 1, ?u)\} \quad x = v; \quad \{\mathsf{PointsTo}(x, 1, v)\}$$

$$[Read] \quad \{\mathsf{PointsTo}(x, \pi, v)\} \quad l = x; \quad \{\mathsf{PointsTo}(x, \pi, v) * l == v\}$$

The PointsTo predicate is a *splittable token*, and may be distributed among different parallel threads. This is shown by the $[SplitPerm]$ rule, where $*\text{-}*$ denotes a *separating equivalence* (two-way magic wand):

$$[SplitPerm] \quad \mathsf{PointsTo}(x, \pi, v) {*\text{-}*} \mathsf{PointsTo}(x, \pi_1, v) * \mathsf{PointsTo}(x, \pi_2, v), \quad \pi = \pi_1 + \pi_2$$

Framing and Stability.

Permission-based separation logic is based on the concept of *framing*: every shared location x in a formula must be

```
   class Counter {
2   ...
   //@ pred res_inv = PointsTo(data, 1, ?v);
4    lock = new Lock/*@<res_inv>@*/;

6  //@ requires    //lock_not_held;
   //@ ensures     //lock_not_held;
8  void increase(){
     lock.lock();
10   /*{PointsTo(data, 1, ?v)}*/
       data=data+1;
12   /*{PointsTo(data, 1, v+1);}*/
     lock.unlock();
14   /*{true}*/
    }  }
```

Lst. 2: The Counter class - specification with locks

framed, i.e., the formula must express a positive permission π to x. Holding a permission guarantees that the value of x is *stable* and can not be changed by any other thread. Framing is implicitly maintained with the PointsTo predicate: in general, we can reason about the value of x only via the $\mathsf{PointsTo}(x, \pi, v)$ predicate. This predicate in a way binds together the *knowledge* of the value v at a location x with *an access permission* to x.

Using Locks.

Permission-based separation logic can be used to reason about programs with locks [9, 14]. For each lock, a special predicate is defined, called a *resource invariant*, describing which permissions the lock protects. For example, the resource invariant *res_inv* associated to the lock expresses that the lock protects a write permission to *data*, see Lst. 2, lines 3, 4. When a thread acquires the lock, it gets the associated resource invariant (except for reentrant acquiring) (line 10). Upon final lock release, the thread returns the resource invariant back to the lock (line 14).

4. APPROACH

The specification of the *Counter* class (see Lst. 2) is strong enough to verify data-race freedom: however, it does not state anything about the behaviour of the *increase* method. Although we can not reason about the value of *data* in the method poststate, we would like the postcondition to express that the method has properly changed the value of *data*. This raises the question: *How can we reason about the value of a heap location x, without holding any permission to x?*

4.1 Separation of the PointsTo Predicate

The proof outline of the *increase* method (see Lst. 2) shows that one can reason about the value of *data* only while the permission to *data* is held. Once the lock is released and the PointsTo predicate is lost (line 13), we lose also the information about the value of *data*. Our intention is to provide a technique that allows a resource invariant to store permissions to certain locations, while the information about the values for these locations can be handled independently.

The key of our concept is the following rule:

$$[\mathit{Separate}]$$
$$\mathsf{PointsTo}(x, 1, v) \;\text{*-*}\; \mathsf{Perm}(x, 1, v) * \mathsf{Hist}(x, 1, \{v\}, \epsilon)$$

The [*Separate*] rule splits the PointsTo predicate in two separate parts: i) $\mathsf{Perm}(x, 1, v)$ predicate, which keeps the access permission for the location x and its current *local* value v and ii) $\mathsf{Hist}(x, 1, V, H)$ predicate, which stores some *global* information about how the value of x has been changing in the past. In particular, the parameter V is a set of possible values that x initially had and H is a *history* of changes of the value of x. The history H is modelled as an ACP process algebra term [8], where every action is a *change of x* (we discuss actions more precisely later in Sec. 4.2). Initially, the history is an empty process, $H = \epsilon$.

The second parameter π in the Hist predicate is used to make it a splittable token, as stated by the following rule:

$$[\mathit{SplitHist}]$$
$$\mathsf{Hist}(x, \pi, V, H) \;\text{*-*}\; \mathsf{Hist}(x, \pi_1, V, H_1) * \mathsf{Hist}(x, \pi_2, V, H_2), \quad \pi = \pi_1 + \pi_2, \quad H = H_1 \parallel H_2$$

where $\parallel$ is the standard ACP parallel composition operator. Later, in Sec. 4.2 we explain how H_1 and H_2 are chosen when splitting the Hist token (when forking a new thread).

Reasoning about the value of x is possible either by using the PointsTo predicate, or by using both Perm and Hist predicates. When we reason about methods with internal synchronisation, the resource invariant typically stores the Perm predicate, while the Hist token is independently split and distributed among different parallel threads. When a thread changes the value of x, it has to acquire the lock to obtain a write permission and additionally has to record the change in a form of an action in the owned part of the Hist token. When all threads are joined, their local histories are merged together. Then, a *full* $\mathsf{Hist}(x, 1, V, H)$ token is obtained, which contains a complete information about the global knowledge of x. Then, the value V may be updated to a set of new possible values of x, while the history H is reinitialised to $H = \epsilon$ (this is discussed in Sec. 4.5).

4.2 A History as a Communication Process

Actions.

As discussed above, the history H in the $\mathsf{Hist}(x, \pi, V, H)$ predicate is modelled as an ACP process, where the primitives in the process H represent *actions over x*, i.e., a change of the value of x. An action is defined as part of the program specification with the following syntax:

$$\mathsf{action}\ \mathit{act\_label}\ [\mathsf{Type}\ x]\ (\overline{\mathsf{Type}}\ \bar{l}) \equiv f(\bar{l}, \backslash old(x))$$

The syntax shows that every action is labeled with a name (*action label*), and is parameterised by a special single parameter x that represents the location that is changed. We call this the *location parameter*. The action may further contain an additional list of parameters $\bar{l}$; it is important that in this list we do not allow any heap location.

The right hand-side of the action definition is the *interpretation of the action*, for an action a we denote $rs(a)$. Every action over x is interpreted as a function over the list of parameters $\bar{l}$ and the value $\backslash old(x)$, i.e., the value of x at the moment before the action starts. The function returns the value of x after the action is finished. In practice, an action is not necessarily atomic, but is observed by the other threads as an atomic change.

For every action, the history H carries the action label together with the concrete values of the action parameters $\bar{l}$. The location parameter is not mentioned because it is already stored in the Hist predicate associated to H.

Below, we show examples of three actions. The action a represents incrementing an integer value by k; action b describes adding an element to a list; while action c represents an assignment to a specific value k.

$$\begin{array}{lcl} \mathsf{action}\ a\,[\mathsf{int}\ x]\,(\mathsf{int}\ k) & \equiv & \backslash old(x) + k \\ \mathsf{action}\ b\,[\mathsf{list}\ l]\,(\mathsf{int}\ elem) & \equiv & cons(elem, \backslash old(l)) \\ \mathsf{action}\ c\,[\mathsf{int}\ x]\,(\mathsf{int}\ k) & \equiv & k \end{array}$$

History Merging.

As the $[SplitHist]$ rule shows, when the $\mathsf{Hist}(x, \pi, V, H)$ token is split (when forking a new thread), two histories H_1 and H_2 should be provided for which $H = H_1 \parallel H_2$. Each thread records its own changes in a separate history H_1 or H_2. When threads are joined and H_1 and H_2 are merged, only the new actions from both histories, i.e., those actions recorded after splitting, should be interleaved.

To this end, we extend the set of actions A with an additional set A_s of *synchronisation action labels*. For each label $s \in A_s$, the set A_s also contains its complement $\bar{s} \in A_s$ ($\bar{\bar{s}} = s$). We define that two complementary synchronisation actions communicate in a *silent action*, while communication between any other two actions returns a *deadlock*.

$$\begin{array}{l} \gamma(s, \bar{s}) = \tau \\ \gamma(a, b) = \delta \text{ if } a \notin A_s \vee (a \in A_s \wedge b \neq \bar{a}) \end{array}$$

Furthermore, when a *full* $\mathsf{Hist}(x, 1, v, H)$ token contains a history $H = s \cdot H_1 + H_2$, $s \in A_s$, we can evaluate H to H_2, resulting in the token $\mathsf{Hist}(x, 1, V, H_2)$.

The synchronisation actions and the communication function ($|$) can impose some constraints when evaluating the parallel composition between two processes. For example the expression $p_1 \cdot s \cdot p_2 \parallel q_1 \cdot \bar{s} \cdot q_2, (s, \bar{s} \notin p_1, q_1)$ results in a process $(p_1 \parallel q_1) \cdot (p_2 \parallel q_2)$, i.e., actions from process p_1 and q_2 (or p_2 and q_1) are not interleaved. In practice, the synchronisation actions are used as follows: when a thread t_1, holding a token $\mathsf{Hist}(x, \pi, V, H)$ forks a thread t_2, the token is split:

$$\mathsf{Hist}(x, \pi, V, H) \text{-}\!*\mathsf{Hist}(x, \pi/2, V, H \cdot s) * \mathsf{Hist}(x, \pi/2, V, \bar{s}),$$

where $s \in A_s$. Threads t_1 and t_2 then start to run in parallel, each of them recording its changes to x into its local history, $H \cdot s$ and $\bar{s}$ respectively. When threads are joined, the new histories $H \cdot s \cdot H_1$ and $\bar{s} \cdot H_2$ are merged such that only the actions happened after forking the thread are interleaved: $H \cdot s \cdot H_1 \parallel \bar{s} \cdot H_2$ is *trace equivalent* to $H \cdot (H_1 \parallel H_2)$.

The current approach does not support scenarios where one thread is joined by several threads. We consider that these scenarios are not very common; however, we plan to lift this limitation, generally by storing the same complementary synchronisation action in the histories of all joining threads.

4.3 Program Specifications

Lst. 3 shows the full specification of the *Counter* class containing two methods: *increase*() and *set*(*int*). The *lock* object which protects the field *data* now stores only the permission to *data* (line 3). An action labeled a is defined to represent incrementing an integer value by k (line 5), while the action b describes overriding an integer value (line 6).

Having the Hist predicate, we can easily specify the behaviour of both methods. In their prestate it is required that the current thread holds (part of) the Hist token associated to *data* (lines 8, 18), while the postconditions guarantee that

```
   class Counter{
2    int data;
     Lock lock; /* res_inv = Perm(data, 1, ?u); */
4    . . .
   //@ action a[int x](int k) ≡ \old(x) + k;
6  //@ action b[int x](int k) ≡ k;

8  //@ requires Hist(data, π, V, H);
   //@ ensures Hist(data, π, V, H.a(1));
10   void increase(){
       lock.lock();
12 //@ start a[data](1);
       data=data+1;
14 //@ commit a[data](1);
       lock.unlock();
16   }

18 //@ requires Hist(data, π, V, H);
   //@ ensures Hist(data, π, V, H.b(k));
20 void set(int k){
       lock.lock();
22 //@ start b[data](k);
       this.data = k;
24 //@ commit b[data](k);
     lock.unlock();
26 }
```

Lst. 3: The Counter class - complete specification

the proper change over *data* is recorded in the history H (lines 9, 19). Thus, no permission to *data* is needed in the pre- or poststate of the method: the permission is obtained inside the method via the lock object.

It is required that the program segment where a certain action occurs is explicitly specified in the program. Therefore, we introduce two specification commands: i) $\mathsf{start}(\mathsf{a}\,[\mathsf{x}]\,(\mathsf{l}))$ indicates the beginning of the action and ii) $\mathsf{commit}(\mathsf{a}\,[\mathsf{x}]\,(\mathsf{l}))$ indicates the end of the action after which the action must be recorded in the history (see Lst. 3, lines 12, 14 and 22, 24). We consider that actions are correctly typed (a *start* action is always followed by a corresponding *commit* command). Further, actions over a same location do not overlap: this is important in order to avoid recording the same update several times in the history.

4.4 Verification Methodology

To check whether the program meets the specification, the verifier must: i) ensure that the *start* and *commit* specification commands are properly added when required; ii) ensure that the actions added to the history have indeed happened.

Ensuring start and commit existence.

When updating the value of a certain location x, we want to ensure that the change is registered somewhere. When using the PointsTo predicate, the newly assigned value is directly recorded into the predicate itself, see Hoare triple [Write], Sec. 3. With our approach, the PointsTo predicate is split into the predicates Perm and Hist. Thus, in addition to the triple [Write] and [Read], we need to introduce another rules for writing and reading that should be used when the PointsTo predicate is split. In particular, we have to ensure that the assignment to x happens indeed as part of an action over x that later will be added to the history of changes of x.

To start an action over x, a write access to x is required,

```
    //@ requires Hist(data, π, V, H);
2   //@ ensures Hist(data, π, V, H·a(1));
    void increase(){
4     /*{Hist(data, π, V, H)}*/
      lock.lock();
6     /*{Perm(data, 1, ?v) * Hist(data, π, V, H)}*/
      //@ start a[data](1);
8     /*{Perm(data, 1, v) * HistPerm(data, π, V, H, v)}*/
        data=data+1;
10    /*{Perm(data, 1, v+1) * HistPerm(data, π, V, H, v)*/
      //@ commit a[data](1);
12    /*{Perm(data, 1, v+1) * Hist(data, π, V, H·a(1))}*/
      lock.unlock();
14    /*{Hist(data, π, V, H·a(1)}*/
    }
```

Lst. 4: Proof outline of the *increase* method

i.e., the $\mathsf{Perm}(x, 1, v)$ predicate. Within the action, the current thread must not lose this predicate. This is important to ensure that actions over the same location do not interleave. Additionally, the start command requires (part of) the Hist token associated to x, which is consumed and replaced with a new HistPerm token, which exists until the action ends. This token is in a way a permission obtained from the history that allows writing at the location x, with a guarantee that the changes will be recorded later. The current value of x is then copied into the HistPerm predicate. This is described by the following Hoare triple:

$$[Start] \qquad \begin{array}{c}\{\mathsf{Perm}(x,1,v) * \mathsf{Hist}(x,\pi,V,H)\}\\ \mathsf{start}\ a[\overline{x}](\overline{\bar{l}});\\ \{\mathsf{Perm}(x,1,v) * \mathsf{HistPerm}(x,\pi,V,H,v)\}\end{array}$$

The new Hoare triples for writing and reading a location x (in addition to the rules $[Write]$ and $[Read]$) are defined as:

$$[WrtHist] \quad \begin{array}{c}\{\mathsf{Perm}(x,1,?u) * \mathsf{HistPerm}(x,\pi,V,H,v)\}\\ x = w;\\ \{\mathsf{Perm}(x,1,w) * \mathsf{HistPerm}(x,\pi,V,H,v)\}\end{array}$$

$$[RdHist] \quad \{\mathsf{Perm}(x,\pi,v)\}\ l = x;\ \{\mathsf{Perm}(x,\pi,v) * l == v\}$$

Ensuring actions correctness.

Before the action ends, the verifier checks whether the specified action is properly executed. The Hoare triple for committing an action states:

$$[Commit]$$
$$\begin{array}{c}\{\mathsf{Perm}(x,1,w) * \mathsf{HistPerm}(x,\pi,V,H,v) * w == rs(a[v](\bar{l}))\}\\ \mathsf{commit}\ a[\overline{x}](\overline{\bar{l}});\\ \{\mathsf{Perm}(x,1,w) * \mathsf{Hist}(x,\pi,V,H\cdot a(\bar{l}))\}\end{array}$$

With the execution of the *commit* command, the action is recorded in the history under the condition that the old value of x, i.e., v, is properly updated to a new value w according to the action interpretation. Lst. 4 shows the proof outline for the *increase* method.

4.5 Reasoning using a History

As discussed above, to reason about the value of a heap location x we need the *full* $\mathsf{Hist}(x, 1, V, H)$ token, which ensures that x is in a *stable state* and no thread can modify its value. The set of possible values for x can be calculated

```
    class Client{
2     void main(){
        Counter c = new Counter(0);
4   /*{Hist(c.data, 1, {0}, ϵ)}*/
      Thread t = new Thread(c);
6       t.start(); // t calls c.increase();
    /*{Hist(c.data, 1/2, {0}, s)} (s is a sync. act.)*/
8       c.set(4);
    /*{Hist(c.data, 1/2, {0}, s·b(4))}*/
10      t.join();
    /*{Hist(c.data, 1, {0}, s·b(4) || s̄·a(1)}*/
12  /*{Hist(c.data, 1, [[(b(4) || a(1))]]^{0}), ϵ)}*/
    /*{Hist(c.data, 1, [[(b(4)·a(1) + a(1)·b(4))]]^{0}), ϵ)}*/
14  /*{Hist(c.data, 1, {4,5}, ϵ)}*/
      }
16  }
```

Lst. 5: A Client class - reasoning using histories

after interpreting all actions from the history. This is stated by the rule:

$$\mathsf{Hist}(x,1,V,H) \text{-}\!*\mathsf{Hist}(x,1,[[H]]^V,\epsilon),$$

where $[[H]]^V$ returns a set of possible values for x after the evaluation of the process H of actions over x, where the initial value of x was any $v \in V$.

We define the $[[H]]^V$ operation inductively as follows (note that the $\|$ operator can be reduced to $\cdot$ and $+$):

$$\begin{array}{rrl} i) & [[\epsilon]]^V & = V \\ ii) & [[H_1 + H_2]]^V & = [[H_1]]^V \cup [[H_2]]^V \\ iii) & [[a(\bar{l}) \cdot H]]^V & = [[H]]^{V'},\ V' = \{rs(a[v](\bar{l}))|v \in V\} \end{array}$$

Lst. 5 shows an example of a client that uses a *Counter* object c. During the initialisation phase of the object c the $\mathsf{PointsTo}(c.data, 1, 0)$ predicate is obtained from which the permission part, $\mathsf{Perm}(c.data, 1, 0)$, is transferred into the lock. Thus, the client obtains the Hist predicate for *data* (line 4). The client starts a new thread t and then both threads running in parallel use the same *Counter* object: thread t increments the value $c.data$ by 1 (line 6), while the client thread assigns $c.data$ to 4 (line 8). The Hist token is divided into two parts (line 7), so both threads record the change in their own history. At the end, both histories are merged (line 11). The client, holding the full Hist token can reason that the value of *data* is either 4 or 5 (line 14).

5. CONCLUSIONS AND RELATED WORK

This paper introduced a new history-based technique for modular reasoning about concurrent programs. The technique allows one to provide intuitive method specifications that describe only the *local effect* of a thread, in terms of abstract (user-specified) *actions*. This reduces the need to reason about fine-grained thread interleavings. The technique is an extension of permission-based separation logic.

Comparable to our approach, is the work on *linearisability* [19, 20]. A method is linearisable if the system can observe it as if it is atomically executed. Linearisability is proved by identifying *linearisation points*, i.e. points where the method takes effect. This allows one to specify a concurrent method in the form of sequential code, which is inlined in the client's code (replacing the call to the concurrent method). In a similar spirit, Elmas et al. [7] abstract away from reasoning about fine-grained thread interleavings, by

transforming a fine-grained program into a corresponding course-grained program. The general idea behind the code transformation is that consecutive actions are merged in a proper way to increase atomicity up to the desired level.

Compared to these approaches, our technique provides more flexibility, because the interpretation of the abstract actions is user-specified. In particular, it may consist of several complex operations. Additionally, we postpone how the action is to be interpreted, and first build an abstract process algebra term to model the history. This means that any process algebra optimisation can be applied on the history as well. Finally, in contrast to the work presented above, our technique is also suited to reason about object-oriented code with dynamic thread creation.

Another approach to reason about the functional behaviour of concurrent programs is by using *Concurrent Abstract Predicates* [6], which extends separation logic with *shared regions*. A specification of a shared region describes possible interference, in terms of actions and permissions to actions. These permissions are given to the client thread to allow them to execute the predefined actions according to a hardcoded usage protocol. A more advanced logic is the extension of this work to iCAP (Impredicative Concurrent Abstract Predicates) [18], where a concurrent abstract predicate may be parameterised by a protocol defined by the client. In a similar spirit, Jacobs et al's [11] propose to reason about a data structure with internal synchronisation, by augmenting the client program with ghost code that is passed as an argument to the module. This results in a kind of a higher-order programming, in order to allow auxiliary variable updates into the module.

Compared to this work, our technique allows more natural specifications where a method contract may describe the thread's local changes, and there is no need to specify a protocol or any auxiliary ghost code. The abstraction provided by specifying actions helps to keep the specifications and program code clean, and requires only a few annotations.

Future Work Our next goal is to reason about more complex concurrent data structures. For this, we expect that our technique can be applied, if the specifications are expressed in terms of actions over a ghost field that represents the real data structure. Next, we plan to extend the definition of an action to allow more expressive specifications, as well as to support actions over multiple locations. We also intend to integrate our history-based approach in reasoning about distributed software.

Aknowledgements We would like to thank Bart Jacobs and Dilian Gurov for their helpful comments. This work was supported by ERC grant 258405 for the VerCors project.

6. REFERENCES

[1] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors project: setting up basecamp. In *PLPV*, pages 71–82, 2012.

[2] S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. In *Formal Methods (FM) 2014*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.

[3] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.

[4] C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[5] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[6] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[7] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

[8] W. Fokkink. *Introduction to Process Algebra*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 2000.

[9] C. Haack, M. Huisman, C. Hurlin, and A.Amighi. Permission-based separation logic for Java, 201x. Conditionally accepted for LMCS.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[11] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[12] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 31(1), 2008.

[13] G. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. *JML Reference Manual*, Feb. 2007.

[14] P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, pages 49–67, 2004.

[15] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comp. Sci.*, 375(1-3):271–307, 2007.

[16] S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[17] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on LICS 2002*, pages 55–74. IEEE Computer Society.

[18] K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.

[19] V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[20] V. Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464, 2010.