History-based Verification of Functional Behaviour of Concurrent Programs

Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski

University of Twente, the Netherlands

Abstract. We extend permission-based separation logic with a *history-based* mechanism to simplify the verification of *functional properties* in concurrent programs. This allows one to specify the local behaviour of a method *intuitively* in terms of actions added to a *local* history; local histories can be combined into global histories, and by resolving the global histories, the reachable state properties can be determined.

1 Introduction

This paper is about verifying *functional properties* of concurrent programs. Although crucially important, these properties are notoriously difficult to verify. A functional property describes what the program is actually expected to do; thus it needs to be manually specified. Moreover, a practical verification technique should be modular, which requires specifying the behaviour of every component (method/thread). Sadly, this causes problems in a concurrent program, because any external thread can change the behaviour of the thread that we describe.

Example 1. We illustrate this problem on a version of the well-known Owicki-Gries example [16], listed below: two threads are running in parallel, each of them incrementing the value of a shared location x by 1. Access to x is protected by the lock lx. If the value of x initially was 0, we would like to prove that at the end, after both threads have finished their updates, the value of x equals 2.

```
void main(){
  x=0;
  incr() || incr();
  print(x);
}
```

```
void incr(){
 lx.lock()
  x=x+1;
 lx.unlock();
}
```

Ideally, we want to specify the code thinking only *locally*. Thus, a postcondition Q of the method $incr()$ would describe that the value of x is increased by 1, i.e., $Q : x == \backslash\mathsf{old}(x) + 1$. Unfortunately, this is not possible, because the expression Q is not *stable*, i.e., it can be invalidated by other parallal threads.

It seems that the lock lx controls where and when we can express something about the value of x. We could try to express the behaviour of x via an *invariant* associated to the lock (as proposed in [16]). However, specifying such an invariant is not easy, because it must be preserved by the behaviour of all threads.

Our Approach In this paper we propose an alternative approach for reasoning about behaviour of concurrent programs, based on using *histories*. A history is a *process algebra* term used to trace the behaviour of a chosen set of shared locations L. When the client has some initial knowledge about the values of the locations in L, it initialises an empty *global* history over L. The global history can be split into *local* histories and each split can be distributed to a different thread. One can specify the local thread behaviour in terms of *abstract actions* that are recorded in the local history. When threads join, local histories are merged into a global history, from where the possible new values of the locations in L can be derived. Therefore, a local history remembers what a single thread has done, and allows us to postpone the reasoning about the current state until no thread uses the history. The approach is based on a variant of *permission-based separation logic* [4, 1]. As a novelty, we extend the definition of the *separating conjunction (∗)* to allow splitting and merging histories.

Every action from the history is an instance of a predefined specification action, which has a contract only and no body. For example, to specify the *incr* method, we first specify an action a, describing the update of the location x (see the code below). The behaviour of the method *incr* is then specified as an extension of a local history over L with the action $a(1)$. This local history is used only by the current thread, which makes history-based specifications stable.

```
//@ requires true;
//@ ensures x == \old(x)+k;
action a(int k);
```

```
//@requires H_L;
//@ensures  H_L· a(1),
  void incr(){...};
```

We reason about the *main*() method as follows. Initially, the only knowledge is $x == 0$. After execution of both parallel threads, a history $H_L = a(1) \parallel a(1)$ is obtained. We can then calculate all traces in H_L and conclude that the value of x is 2. Note that each trace is a sequence of actions, each with a pre- and postcondition; thus this boils down to reasoning about a sequential program.

Using histories allows modular and *intuitive* specifications that are not more complicated than sequential specifications. Reasoning about the history H involves calculating thread interleavings. However, we do not consider this as a weakness because: i) the history abstracts away all unnecessary details and makes the abstraction simpler than the original program; ii) the history mechanism is integrated in a standard modular program logic, such that histories can be employed to reason only about parts of the program where modular reasoning is troublesome; and iii) we allow the global history to be *reinitialised* (to be emptied), and moreover, to be destroyed. Thus, the management of histories allows keeping the abstract parts small, which makes reasoning more manageable.

Contributions We propose a novel approach to specify and verify behaviour of coarse-grained concurrent programs that allows intuitive modular specifications. We provide a formalisation of the approach on an object-oriented language with dynamic thread creation, and integrate it in our VerCors tool set [2]. The technique has also been experimentally added on top of the VeriFast logic [18].

2 Background

Permission-based separation logic Our approach is based on permission-based separation logic (PBSL) [17, 15, 1], a logic for reasoning about multi-threaded programs. In PBSL every access to a shared location is associated with a *fractional permission* $\pi \in (0, 1]$. To change a location x, a thread must hold a *write* permission to x, i.e., $\pi = 1$; to read a location, any *read* permission i.e., $\pi > 0$, is sufficient. For every newly initialised shared location with a value v, the current thread obtains a write permission, represented by the predicate $\mathsf{PointsTo}(x, 1, v)$. Permissions may be split into fractions and distributed among threads: $\mathsf{PointsTo}(x, \pi_1+\pi_2, v) \mathbin{*\!\!-\!\!*} \mathsf{PointsTo}(x, \pi_1, v) * \mathsf{PointsTo}(x, \pi_2, v)$ (the operator $*\!\!-\!\!*$ is read "splitting" (from left to right) and "merging" (from right to left)). Soundness of the logic ensures that a verified program is data-race free, because the sum of all threads' permissions for a given location never exceeds 1.

Locks To reason about locks, we use the protocol described by Haack et al. [1]. Following the work in [16, 15], for each lock they associate a *resource invariant* *inv*, i.e., a predicate that describes the locations that the lock protects. A newly created lock is still *fresh* and not ready to be acquired. The thread must first execute a (specification-only) *commit* command that transfers the permissions from the thread to the lock and changes the lock's state to *initialised*. Any thread then may acquire the initialised lock and obtain the resource invariant. Upon release of the lock, the thread returns the resource invariant back to the lock.

The µCRL language To model histories, we use µCRL [9]. µCRL is powerful and sufficiently expressive for our needs because it allows process algebra terms parametrised by data. Basic primitives in the language are *actions* from the set $\mathcal{A}$, each of them representing an indivisible process behaviour. There are two special actions: the *deadlock action* δ and the *silent action* τ (an action with no behaviour). *Processes* $\{p_1, p_2, ...\}$ are defined by combining actions and recursion variables, which may also be parametrised. With ϵ we denote the *empty process*.

To compose actions, we have the following basic operators: the *sequencing composition* $(\cdot)$; the *alternative composition* $(+)$; the *parallel composition* $(\|)$; the *abstraction operator* $(\tau_{\mathcal{A}'}(p))$, which renames all occurrences of actions from the set $\mathcal{A}'$ by τ; the *encapsulation operator* $(\partial_{\mathcal{A}'}(p))$, which disables unwanted actions by replacing all occurrences of actions in $\mathcal{A}'$ by δ; the *sum operator* $\sum_{d:D} P(d)$, which represents a possibly infinite choice over data of type D; and the *conditional operator* $p \lhd b \rhd q$, which describes the behaviour of p if b is true and the behaviour of q otherwise.

Parallel composition is defined as all possible interleavings between two processes: $p_1 \parallel p_2 = (p_1 \between p_2) + (p_2 \between p_1) + (p_1 \mid p_2)$. The *left merge* $(\between)$ operator defines a parallel composition of two processes where the initial step is always the first action of the left-hand operator, while with the *communication merge* $(|)$ operator, the first step is a communication between the first actions of each process: $a \cdot p_1 \mid b \cdot p_2 = a \mid b \cdot (p_1 \parallel p_2)$. The result of a communication between two actions is defined by a function $\gamma : \mathcal{A} \times \mathcal{A} \mapsto \mathcal{A}$, i.e., $a \mid b = \gamma(a, b)$.

```
class Counter {
  int x;
//@pred inv = Perm(x,1,v);
 Lock lx = new Lock/*@<inv>@*/();

//@accessible {x};
//@assignable {x};
//@requires k>0;
//@ensures x=\old(x)+k;
//@action inc(int k);

//@requires Hist(L,π,R,H) * x ∈ L
//@ensures Hist(L,π,R,H·inc(1))
 void incr(){
  lx.lock();
/*Hist(L,π,R,H)*Perm(x,1,v)}*/
//@ action inc(1){
/*Hist(L,π, R, H)*APerm(x,1,v)*/
      x = x+1;
/*Hist(L,π,R,H)*APerm(x,1,v+1)}*/
//@ }
/*Hist(L,π,R,H·inc(1))*Perm(x,1,v+1)}*/
  lx.unlock();
/*Hist(L.π,R,H·inc(1))*/
 }
}
```

```
class Client{
 Thread t1; Thread t2;

void main(){
  Counter c = new Counter();
/*PointsTo(c.x,1,0)}*/
  t1 = new Thread(c);
  t2 = new Thread(c);
/*PointsTo(c.x,1,0)}*/
//@ crHist({c.x}, c.x==0);
/*Perm(c.x,1,0)*Hist({c.x},1,c.x==0,ϵ)}*/
//@ c.lock.commit();
/*{Hist({c.x},1,c.x==0,ϵ)}*/
  t1.fork();     // t1 calls c.incr();
/*Hist({c.x},1/2,c.x==0,ϵ)}*/
  t2.fork();     // t2 calls c.incr();
/*Hist({c.x},1/4,c.x==0,ϵ)}*/
  t1.join();
/*Hist({c.x},1/2,c.x==0, c.inc(1))}*/
  t2.join();
/*Hist(c.x,1,c.x==0, c.inc(1)‖c.inc(1))}*/
//@ reinit({c.x}, c.x==2);
/*Hist({c.x},1,c.x==2,ϵ)*/
 }
}
```

Lst. 1. The Counter example

3 Modular History-Based Reasoning

In this section we discuss informally our approach, illustrating it on a Java-like variant of the Owicki-Gries example, see Lst. 1.

The classical approach is to associate the lock lx with a resource invariant $inv = \mathsf{PointsTo}(x, 1, v)$ [15, 1]. However, the PointsTo predicate stores both access permission to x and the information about the value of x. Therefore, in the *incr* method, after releasing the lock, all information about the value of x is lost, and describing the method's behaviour in the postcondition is problematic. Therefore, our approach aims to separate permissions to locations from their values (the functional properties). While a resource invariant stores permissions to locations, the values of these locations are treated separately by using a *history*.

A history refers to a set of locations L (we call it *a history over* L) and is used to record all updates made to any of the locations in L. The same location can not appear in more than one existing history simultaneously. A history is represented by a predicate $\mathsf{Hist}(L, 1, R, H)$, which contains the *complete* knowledge about the values of the locations in L. The predicate R captures the knowledge about these values in the *initial state*, i.e., the state when no action has been recorded in the history. Further, H is a µCRL process [9] that represents the *history of updates* over locations in L. The second parameter in the Hist predicate is used to make it a splittable predicate: a predicate $\mathsf{Hist}(L, \pi, R, H)$, where $\pi < 1$ contains only *partial* knowledge about the behaviour of L.

Creating a history A history over L is created by the specification command $\mathsf{crhist}(L, R)$, where R is a predicate over locations in L that holds in the current

state. This command requires a full $\mathsf{PointsTo}(l, 1, v)$ predicate for each location $l \in L$, converts it to a new $\mathsf{Perm}(l, 1, v)$ predicate, and produces a history predicate $\mathsf{Hist}(L, 1, R, \epsilon)$. The $\mathsf{Perm}(l, 1, v)$ predicate has essentially the same meaning as $\mathsf{PointsTo}(l, 1, v)$; however, it indicates that there also exists a history that refers to l, and any change of l must be recorded in this history. In this way we prevent existence of more than one history over the same location.

In Lst. 1, the resource invariant is defined using the Perm (instead of $\mathsf{PointsTo}$) predicate (line 3). Thus, the lock stores the permission to x only, while independently there exists a history that records all updates to x. The client creates a history over a single location x in line 36. After the permissions are transferred to the lock (line 38), the client still keeps the full Hist predicate. This guarantees that the value of x is stable even without holding any access permission to x.

Splitting and merging histories The history may be redistributed among parallel threads by splitting the predicate $\mathsf{Hist}(L, \pi, R, H)$ into two predicates $\mathsf{Hist}(L, \pi_1, R, H_1)$ and $\mathsf{Hist}(L, \pi_2, R, H_2)$, where $H = H_1 \parallel H_2$ and $\pi = \pi_1 + \pi_2$. The basic idea is to split H such that $H_1 = H$ and $H_2 = \epsilon$. However, if we later merge the two histories, we should know at which point H was split. Concretely, if we split H, and then one thread does an action a, and the other thread an action b, after merging the histories, the result should be a history $H \cdot (a \parallel b)$.

To ensure proper *synchronisation* of histories, we add *synchronisation barriers*. That is, given two history predicates with histories H_1 and H_2, and actions s_1 and s_2 such that $\gamma(s_1, s_2) = \tau$, we allow one to extend the histories to $H_1 \cdot s_1$ and $H_2 \cdot s_2$. We call s_1 and s_2 *synchronisation actions* (we usually denote them with s and $\overline{s}$). When threads join (a thread can join at most once in the program), all partial histories over the same set of locations L are *merged*.

In Lst. 1 the Hist predicate is split when the client forks each thread (lines 40 and 42). Thus both threads can record their changes in parallel in their own partial history. Note that in this example there is no need of adding a synchronisation barrier, because we split the history when it is still empty. We illustrate synchronisation barriers later in Example 2.

Recording actions We extend the specification language with *actions*. An action is defined by a *name* and *parameters*, and is equipped with a *specification*: a pre- and postcondition; an *accessible* clause which defines the *footprint of the action*, i.e., a set of locations that are allowed to be accessed within the action; and an *assignable* clause, which specifies the locations allowed to be updated.

Lst. 1 shows a definition of an action *inc* (lines 6 - 10), which represents an increment of the location x by k. Note that the action contract is written in a pure JML language [13], without the need to explicitly specify permissions, as they are treated separately. In particular, action contracts are used to reason about traces of histories, which (as discussed above) are actually sequential programs.

We can associate a program segment *sc* with a predefined action, by using the specification command $\mathsf{action}\ a(\overline{v})\{sc\}$, see lines 17 - 21 in Lst. 1. We call *sc* an *action segment*. In the prestate of the action segment, a history predicate

$\mathsf{Hist}(L, \pi, R, H)$ is required, which captures the behaviour of a's footprint locations, i.e., $\forall l \in \mathit{footprint}(a).l \in L$. At the end of the action segment, the action is recorded in the history, see line 22 in Lst. 1. For this, it is necessary that the action segment implements the specification of the action a.

Restrictions within an action An action must be observed by the environmental threads as if it is *atomic*. Thus, it is essential that within the action segment the footprint locations of the action are *stable*, i.e., they can not be modified by any other thread. To ensure this, we impose several restrictions on what is allowed in the action segment (a formal definition is given in Sec. 4). In the prestate of the action a, we require that the current thread has a positive permission to every footprint location of a, which must not be released within the action segment. Concretely, within an action segment, we allow only a specific subcategory of commands. This excludes lock-related operations (acquiring, releasing or committing a lock), forking or joining threads. Nested actions are also forbidden in order to prevent a thread to record the same action twice.

In this way, two actions may interleave only if they refer to disjoint sets of locations, or if their common locations are only readable by both threads. It might be possible to lift some of these restrictions later; however, this would probably add extra complexity to the verification approach, while we have not yet encountered examples where these restrictions become problematic.

Updates within an action If a history H over l exists, the access permission to l is provided by the $\mathsf{Perm}(l, \pi, v)$ predicate (instead of $\mathsf{PointsTo}(l, \pi, v)$). Every update to l must then be part of an action that will be recorded in H. Thus, the $\mathsf{Perm}(l, \pi, v)$ predicate is "valid" only within an action segment with a footprint that refers to l. To this end, within the action segment, the $\mathsf{Perm}(l, \pi, v)$ predicates are exchanged for predicates $\mathsf{APerm}(l, \pi, v)$. Thus, our logic allows a thread to access a shared location when it holds an appropriate fraction of either the $\mathsf{PointsTo}$ or the APerm predicate (see lines 17 - 21 in Lst. 1).

Reinitialisation and destroying When a thread has the full $\mathsf{Hist}(L, 1, R, H)$ predicate, it has complete knowledge of the values of the locations in L, and the locations are then *stable*. The Hist predicate remembers a predicate R that was true in the previous initial state σ of the history, while the history H stores the abstract behaviour of the locations in L after the state σ. Thus, it is possible to *reinitialise* the Hist predicate, i.e., reset the history to $H = \epsilon$ and update the R to a new predicate R' that holds over the current state. Thus, reasoning about the continuation of the program will be done with an initial empty history.

The specification command $\mathsf{reinit}(L, R')$ converts the $\mathsf{Hist}(L, 1, R, H)$ predicate to a new $\mathsf{Hist}(L, 1, R', \epsilon)$. Reinitialisation is successful when the new property R' can be proven to hold after the execution of any trace w from the set of traces in H, i.e., $\forall w \in \mathsf{Traces}(w).\{R\}w\{R'\}$. As stated above, each trace is a sequence of specified actions and thus, can be seen as a sequential program.

In Lst. 1, the history is reinitialised at line 48. The new specified predicate over the location x is: $x == 2$. Notice that at this point, the client does not hold

```
class ComplexCounter {
 int data; int x; int y;
//@pred invx=Perm(x,1,v)*Perm(data,1/2,u);
//@pred invy=Perm(y,1,v)*Perm(data,1/2,u);
Lock lockx=new Lock/*@<invx>@*/();
Lock locky=new Lock/*@<invy>@*/();
/*@ accessible {x, data};
  @ assignable {x};
  @ ensures x = \old(x) +data;
  @ action addx();
  @ accessible {y, data};
  @ assignable {y};
  @ ensures y = \old(y) +data;
  @ action addy();
  @ accessible {data};
  @ assignable {data};
  @ requires k>0;
  @ ensures data = \old(data) +k;
  @ action inc(int k);
  @ accessible {data};
  @ assignable {data};
  @ ensures data = \old(data)+n;
  @ proc p(int n) = inc(1).p(n−1)◁ n>0 ▷ϵ;
@*/
//@ requires Hist(L, π,R,H) ∗ data,x ∈ L
//@ ensures Hist(L, π,R,H·addx())
void addX(){
  lockx.lock();
//@ action addx(){
    x=x+data;
//@ }
  lockx.unlock();
 }
//@ requires Hist(L, π,R,H) ∗ data,y ∈ L
//@ ensures Hist(L, π,R,H·addy())
void addY(){
  locky.lock();
//@ action addy(){
    y=y+data;
//@ }
  locky.unlock();
 }
//@ requires Hist(L, π,R,H) ∗ data ∈ L
//@ ensures Hist(L, π,R,H·p(n))
void incr(int n){
  if (n>0){
    lockx.lock(); locky.lock();
//@ action inc(1){
      data++;
//@ }
    lockx.unlock(); locky.unlock();
    incr(n−1);
  }
 }
}
```

Lst. 2. Complex Counter example

any permission to access x. However, holding the full Hist predicate is enough to reason about the current value of x.

Finally, the history may be destroyed using the $\mathsf{dsthist}(L)$ specification command. The $\mathsf{Hist}(L, 1, R, \epsilon)$ predicate and the $\mathsf{Perm}(l, 1, v)$ predicates for all $l \in L$ are exchanged for the corresponding $\mathsf{PointsTo}(l, 1, v)$ predicates. Thus, this will allow the client to create a history predicate over a different set of locations.

Example 2. We illustrate our approach on a more involved example, with recursive method calls and a location protected by two different locks. The class *ComplexCounter* (Lst. 2) contains three fields: *data*, x and y. A lock *lockx* protects write access to x and read access to *data*, while *locky* protects write access to y and read access to *data*. Both locks together protect write access to *data*.

Methods $addX()$ and $addY()$ increase respectively x and y by *data*, while the recursive method $incr(n)$ increments *data* by n. The synchronised code in methods $addX()$, $addY()$ and $incr(n)$ is associated with a proper action. We also specify a recursive *process* p, line 30. The contract of the $incr(n)$ method shows that the contribution of the current thread is not an atomic action, but a process that can be interleaved with other actions. The contract of the process must correspond to the contracts of the actions it is composed of.

Lst. 3 presents a *Client* class that creates a *ComplexCounter* object c and shares it with two other parallel threads, t_1 an t_2. The client thread updates $c.data$ (lines 15, 21), while the threads t_1 and t_2 update the locations $c.x$ and

```
   class Client{
 2  ThreadX tx; ThreadY ty;
   void main(){
 4   ComplexCounter c=new ComplexCounter();
     tx = new ThreadX(c);    ty = new ThreadY(c);
 6  /* PointsTo(c.data,1,0)*PointsTo(c.x,1,0)*PointsTo(c.y,1,0) */
     //@ crHist(L, R);                //create history
 8  /* Perm(c.data,1,0)*Perm(c.x,1,0)*Perm(c.y,1,0)}*Hist(L,1,R,ϵ) */
     //@ c.lockx.commit();
10   //@ c.locky.commit();
     /*Hist(L,1,R,ϵ)}*/     //split history
12   /*Hist(L,1/2,R,ϵ) * Hist(L,1/2,R,ϵ)*/
     tx.fork();     // tx calls c.addx();
14   /*Hist(L,1/2,R,ϵ)*/
     c.incr(10);
16   /*Hist(L,1/2,R,p(10))*/        //split history
     /*Hist(L,1/4,R,p(10)) * Hist(L,1/4,R,p(10))*/    //sync. barrier
18   /*Hist(L,1/4,R,p(10)·s)) * Hist(L,1/4,R,p(10)·s̄))*/    //sync. barrier
     ty.fork();     // ty calls c.addy();
20   /*Hist(L,1/4,R,p(10)·s))*/
     c.incr(10);
22   /*Hist(L,1/4,R,p(10)·s· p(10))*/
     tx.join();     ty.join();             //merge
24   /*Hist(L,1,R,p(10)·s· p(10) || addx() || s̄·add(y))} */
     //@ reinit(L, 10<=c.x+c.y<=40);
26   /*Hist(L,1,10<=c.x+c.y<=40,ϵ)*/
    }
28  } // L={c.data,c.x,c.y} R=c.data==0 ∧ c.x==0 ∧c.y==0
```

Lst. 3. Complex Counter example - the Client class

$c.y$ (lines 13, 19). We want to prove that in the *Client*, at the end after both threads have terminated, the statement $10 \leq c.x + c.y \leq 40$ holds.

The final values of $c.x$ and $c.y$ depend on the moment when $c.data$ has been updated. Thus, the history should trace the updates of all locations, $c.x, c.y$ and $c.data$. Each thread instantiates actions that refer to different sets of locations, but all actions are recorded in the same history. When the threads terminate, the client has the complete knowledge of all values, in the form of a process algebra term $H = p(10) \cdot s \cdot p(10) \parallel addx() \parallel \overline{s} \cdot add(y)$ (line 24). By reasoning about the history H (see Sec. 4), we can prove that the property $R' = 10 \leq c.x + c.y \leq 40$ holds in the current state, and reinitialise the history to $\mathsf{Hist}(L, 1, R', \epsilon)$.

When reasoning about the process H, its definition is expanded by applying the axioms of process algebra and unfolding it until the result is a *guarded process*. Then, all parallel compositions are replaced by defined processes. To perform this, the user has to specify all parallel compositions that might occur (for more details we refer to [3]).

Complex data structures Our technique is also suitable to reason about more complex coarse-grained data structures (e.g. lists, sets). Shortly, method contracts of the data structure can be expressed in terms of histories over a *ghost field* that represents the structure, while a *class invariant* [21] can ensure that the ghost field corresponds to the actual structure. For an example of reasoning about a concurrent *Set* data structure we refer to [3].

$n \in \mathsf{int} \quad b \in \mathsf{bool} \quad o, t \in \mathsf{ObjId} \quad \pi \in (0,1] \quad i \in \mathsf{RdVar} \quad j \in \mathsf{RdWrVar}$
$x \in \mathsf{Var} = \mathsf{RdVar} \cup \mathsf{RdWrVar} \quad a(\overline{v}) \in \mathsf{UAct} \quad s \in \mathsf{SAct}$ (synchr. action)
$qt \in \{\exists, \forall\} \quad \oplus \in \{*, \wedge, \vee\} \quad op \in \{==, !, \wedge, \vee, \Rightarrow, +, -, ...\}$

(class) $cl ::= \mathsf{class}\ C\ \langle \mathsf{pred}\ inv \rangle\ \{\overline{fd}\ \overline{md}\ \overline{pd}\} \mid \mathsf{thread}\ CT\{\mathsf{run}\}$
(field) $fd ::= Tf$
(method) $md ::= \mathsf{requires}\ F\ \mathsf{ensures}\ F\ T\ m(\overline{Vi})\{c\}$
(type) $T, V, W ::= \mathsf{void} \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{perm} \mid \mathsf{process} \mid \mathsf{pred} \mid C\ \langle \mathsf{pred} \rangle \mid CT$
(value) $v, w, u ::= \mathsf{null} \mid n \mid b \mid o \mid i \mid \pi \mid op(\overline{v}) \mid H(\overline{v}) \quad \pi ::= 1 \mid \mathsf{split}(\pi)$
(action) $act ::= \mathsf{accessible}\ L\ \mathsf{requires}\ F\ \mathsf{ensures}\ F\ \mathsf{action}\ a(\overline{T}\ \overline{i});$
(process) $proc ::= \mathsf{accessible}\ L\ \mathsf{requires}\ F\ \mathsf{ensures}\ F\ \mathsf{process}\ p(\overline{T}\ \overline{i}) = H;$
$H ::= \epsilon \mid \delta \mid \tau \mid s \mid a(\overline{v}) \mid H_1 \triangleleft op(\overline{i}) \triangleright H_2 \mid \sum_{d \in D} p(d)$
$\mid H \cdot H \mid H + H \mid H \parallel H$
(predicate) $pd ::= \mathsf{pred}\ P = F$
(formula) $F, G ::= e \mid e.P \mid F \oplus F \mid \mathsf{PointsTo}(e.f, \pi, e)$
$\mid \mathsf{Perm}(e.f, \pi, e) \mid \mathsf{Hist}(L, \pi, R, H) \mid \mathsf{APerm}(e.f, \pi, e)$
$\mid (qt\ T\ x)F \mid e.\mathsf{fresh}() \mid e.\mathsf{initialized}() \mid \mathsf{Join}(e)$
(expression) $e ::= j \mid v \mid op(\overline{e})$
(command) $c ::= v \mid j = \mathsf{return}(v); c \mid T\ j; c \mid T\ i = j; c \mid hc; c$
(head comm.) $hc ::= j = v; \mid j = op(\overline{v}); \mid j = v.f; \mid j = \mathsf{new}\ C\ \langle v \rangle; \mid j = v.m(\overline{v});$
$\mid v.f = v; \mid \mathsf{if}\ v\ \mathsf{then}\ c\ \mathsf{else}\ c;$
$\mid v.\mathsf{lock}(); \mid v.\mathsf{commit}(); \mid v.\mathsf{unlock}(); \mid v.\mathsf{fork}(); \mid v.\mathsf{join}();$
$\mid \mathsf{crhist}(L, R) \mid \mathsf{action}\ v.a(\overline{v})\{sc\} \mid \mathsf{reinit}(L, R) \mid \mathsf{dsthist}(L)$
$sc ::= j = v \mid j = v.f \mid j = \mathsf{new}\ C\ \langle v \rangle \mid v.f = v \mid T\ j; sc \mid T\ i = j; sc$
$\mid \mathsf{if}\ v\ \mathsf{then}\ sc'\ \mathsf{else}\ sc'' \mid sc'; sc'' \mid j = v.m(\overline{v})$

Fig. 1. Language syntax

4 Formalisation

We formalise our approach on a Java-like language. Java uses *fork*(*start*) and *join* primitives to allow modeling various scenarios that are not supported by the simpler parallel operator $\|$. Our system is based on the Haack's formalisation of a logic/PBSL [1] to reason about Java-like programs.

Language syntax Figure 1 combines the syntax of our programming and specification language. Apart from the special actions (δ, τ), we allow: synchronisation actions $s \in \mathsf{SAct}$ and update actions $a(\overline{v}) \in \mathsf{UAct}$. The definition of classes, fields, methods etc. are standard. We often use l to denote a location (instead of writing $v.f$), and L for set of locations. *Thread* classes are a special type of classes with a single *run* method. In addition to the usual definition, values can also be fractional permissions. These are represented symbolically: 1 denotes a *write* permission, while $\mathsf{split}(\pi)$ denotes a fraction $\frac{\pi}{2}$. The language also defines actions (act), which only have a specification; and processes ($proc$), which have a specification and a body, defined as a proper process expression.

To reason about histories, we use the predicates Hist and APerm, and the specification commands: $\mathsf{crhist}(L, R)$, $\mathsf{dsthist}(L)$, $\mathsf{reinit}(L, R)$ and $\mathsf{action}\ v.a(\overline{v})\{sc\}$, where sc is a special subcategory of commands allowed within an action segment. This subcategory includes only calls to methods whose body has the form sc. Commands $t.\mathsf{fork}()$ and $t.\mathsf{join}()$ are used to start or join a thread t respectively. After forking a thread object t, the receiver obtains the $\mathsf{Join}(t)$ predicate, which is a required condition for joining the thread t. This ensures that a single thread is started and joined only once in the program.

To reason about locks, we use the predicates $e.\mathsf{fresh}()$ and $e.\mathsf{initialized}()$ and the $v.\mathsf{commit}()$ command (as discussed in Sec. 2). Every object may be used as a *lock*. Locations protected by the lock are specified by a predicate *inv*, with a default definition $inv = \mathsf{true}$. Each client object may optionally pass a new definition for *inv* as a class parameter when creating the lock object.

Semantics of histories A histories H is a µCRL proces algebra term. The set of actions is: $\mathcal{A} = \mathsf{UAct} \cup \mathsf{SAct} \cup \{\tau, \delta\}$, while the communication function is:

$$\gamma(a,b) = \begin{cases} \tau & \text{if } a,b \in \mathsf{SAct} \text{ define a synchronisation barrier} \\ \bot & \text{otherwise} \end{cases}$$

The semantics of H is defined in terms of its traces. We use the standard single step semantics $H \xrightarrow{a} H'$ for H moving in one step to H', extended to:

$$H \stackrel{\epsilon}{\Rightarrow} H \qquad H \stackrel{a}{\Rightarrow} H' \Leftrightarrow H \xrightarrow{\tau}{}^{*} \xrightarrow{a} \xrightarrow{\tau}{}^{*} H', \text{ for } a \neq \tau \qquad H \stackrel{aw}{\Rightarrow} H' \Leftrightarrow H \stackrel{a}{\Rightarrow} \stackrel{w}{\Rightarrow} H'$$

The *global completed trace semantics* of a term H is defined as:

$$\mathsf{Traces}(H) = \{w \mid \partial_{\mathsf{SAct}}(\tau_{\mathsf{FAct}}(H)) \stackrel{w}{\Rightarrow} \epsilon\},$$

where FAct is the set of finished actions: $\mathsf{FAct} = \{a \in \mathsf{SAct} \mid \forall b \in \mathcal{A}.\gamma(a,b) = \bot\}$.

Operational semantics We model the state as: $\sigma = \mathsf{Heap} \times \mathsf{ThreadPool} \times \mathsf{LockTable} \times \mathsf{InitHeap} \times \mathsf{HistMap}$. The first three components are standard, while all history-related specification commands operate only over the last two.

- $h \in \mathsf{Heap} = \mathsf{ObjId} \rightharpoonup \mathsf{Type} \times (\mathsf{FieldId} \rightharpoonup \mathsf{Value})$ represents the shared memory, where each object identifier is mapped to its *type* and its *store*, i.e., the values of the object's fields: We use $\mathsf{Loc} = \mathsf{ObjId} \times \mathsf{FieldId}$.
- $tp \in \mathsf{ThreadPool} = \mathsf{ThrId} \rightharpoonup \mathsf{Stack}(\mathsf{Frame}) \times \mathsf{Cmd}$ defines all threads operating on the heap. The local memory of each thread is a stack of frames, each representing the local memory of one method call: $f \in \mathsf{Frame} = \mathsf{Var} \rightharpoonup \mathsf{Val}$.
- $lt \in \mathsf{LockTable} = \mathsf{ObjId} \rightharpoonup \mathsf{free} \uplus \mathsf{ThrId}$ defines the status of all locks. Locks can be free, or acquired by a thread:
- $h_i \in \mathsf{InitHeap} = \mathsf{Loc} \rightharpoonup \mathsf{Val}$ (*initial heap*), maps every location for which a history exists to its value in the initial state of the history.
- $hm \in \mathsf{HistMap} = \mathsf{Set}(\mathsf{Loc}) \rightharpoonup \overline{\mathsf{Action}}$ stores the existing histories: it maps a set of locations L to a sequence of actions over L. An action is represented by a tuple $act = \mathsf{ActId} \times \overline{\mathsf{Val}}$, composed of the *action identifier* and *action parameters*. Two histories always refer to *disjoint* sets of locations: $\forall L_1, L_2 \in \mathsf{dom}(hm).\ L_1 \cap L_2 = \emptyset$. This is ensured by the logic because creating a history over l consumes the *full* $\mathsf{PointsTo}$ predicate.

Fig. 2 shows the operational semantics for the commands in our language. For a thread pool $tp = \{t_1, ...t_n\}$, where $t_i = (s_i, c_i)$, we write $(t_1, s_1, c_1). \ldots .(t_n, s_n, c_n)$.

$$
\begin{array}{lrcl}
[Dcl] & (h, tp.(t, f \cdot s, T\ j; c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, f[j \mapsto \mathsf{defaultVal}(T)] \cdot s, c), lt, h_i, hm) \\
[FinDcl] & (h, tp.(t, s, T\ i = j; c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c[s(j)/i]), lt, h_i, hm) \\
[VarSet] & (h, tp.(t, f \cdot s, j = v; c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, f[j \mapsto v] \cdot s, c), lt, h_i, hm) \\
[Op] & (h, tp.(t, f \cdot s, j = op(\overline{v}); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, f[j \mapsto [\![op]\!]^h_s(\overline{v})] \cdot s, c), lt, h_i, hm) \\
[If] & (h, tp.(t, s, \mathsf{if}(b)\{c_1\}\mathsf{else}\{c_2\}; c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c'; c), lt, h_i, hm), \text{where} \\
 & & & b \Rightarrow c' = c_1; \neg b \Rightarrow c' = c_2 \\
[Return] & (h, tp.(t, f \cdot s, j = \mathsf{return}(v); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, j = v; c), lt, h_i, hm) \\
[Call] & (h, tp.(t, s, o.m(\overline{v}); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, \emptyset \cdot s, c_m[o/x_0, \overline{v}/\overline{x}]), lt, h_i, hm), \\
 & & & \text{where } \mathsf{body}(o.m) = c_m(x_0, \overline{x}); \\
[New] & (h, tp.(t, f \cdot s, j = \mathsf{new}\ C\ \langle v \rangle; c), lt, h_i, hm) & \rightsquigarrow & (h' tp.(t, f[j \mapsto o] \cdot s, c), lt[o \mapsto \mathsf{free}], h_i, hm), \\
 & & & \text{where } h' = h[o \mapsto \mathsf{initStore})], o \notin \mathsf{dom}(h) \\
[Get] & (h, tp.(t, f \cdot s, j = o.f; c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, f[j \mapsto h_i(o.f)] \cdot s, c), lt, h_i, hm) \\
[Set] & (h, tp.(t, s, o.f = v; c), lt, h_i, hm) & \rightsquigarrow & (h[o.f \mapsto v], tp.(t, s, c), lt, h_i, hm) \\
[Lock] & (h, tp.(t, s, o.\mathsf{lock}(); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c), lt[o \mapsto p], h_i, hm) \\
[Unlock] & (h, tp.(t, s, o.\mathsf{unlock}(); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c), lt[o \mapsto \mathsf{free}], h_i, hm) \\
[Fork] & (h, tp.(t, s, j = o.\mathsf{fork}(); c), lt, h_i, hm) & \rightsquigarrow & (h, tp(t, s, j = \mathsf{null}; c).(o, \emptyset, c_r[o/x_0]), lt, h_i, hm) \\
 & & & \text{where } o \notin (\mathsf{dom}(tp) \cup \{t\}), \mathsf{body}(o.run) = c_r(x_0); \\
[Join] & (h, tp.(t, s, o.\mathsf{join}(); c).(o, s', v), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c), lt, h_i, hm) \\
[Create] & (h, tp.(t, s, \mathsf{crhist}(L, R); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c), lt, h_i[l \mapsto h(l)]_{\forall l \in L}, hm[L \mapsto \mathsf{nil}]) \\
[Destr] & (h, tp.(t, s, \mathsf{dsthist}(L); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c), lt, h_i[l \mapsto \bot]_{\forall l \in L}, hm[L \mapsto \bot]) \\
[Reinit] & (h, tp.(t, s, \mathsf{reinit}(L, R); c), lt, h_i, hm) & \rightsquigarrow & (h, tp.(t, s, c), lt, h_i[l \mapsto h(l)]_{\forall l \in L}, hm[L \mapsto \mathsf{nil}])
\end{array}
$$

$$
[Action]\ \frac{(h, tp.(t, s, sc), lt, h_i, hm) \rightsquigarrow^\star (h', tp'.(t, s', \mathsf{null}), lt', h'_i, hm')}{(h, tp.(t, s, \mathsf{action}\ o.a(\overline{v})\{sc\}; c), lt, h_i, hm) \rightsquigarrow^\star (h', tp'.(t, s', c), lt', h'_i, hm'')}
$$
$$
\text{where } hm'' = hm'[L \mapsto A{+\!\!+}hm'(L)] \quad A = (o.a, \overline{v})
$$

Fig. 2. Operational semantics, $\sigma \rightsquigarrow \sigma'$.

A stack with a top frame f is denoted as $f \cdot s$. With $[\![e]\!]^h_s$ we denote the semantics of an expression e, given a heap h and a stack s. With nil we denote an empty sequence, while $A{+\!\!+}S$ appends the element A to a sequence S. The function $\mathsf{defaultVal}$ maps types to their default value, $\mathsf{initStore}$ maps objects to their initial stores. With $\mathsf{body}(o.m) = c_m(x_0, \overline{x})$ we define that c_m is the body of the method m, where x_0 is the method receiver, and $\overline{x}$ are the method parameters.

The $\mathsf{crhist}(L, R)$ command copies the value of each $l \in L$ from the Heap to the $\mathsf{InitHeap}$, and extends the domain of $\mathsf{HistMap}$ with the set L, while $\mathsf{dsthist}(L)$ is the opposite: it removes the related entries from $\mathsf{HistMap}$ and $\mathsf{InitHeap}$. The command $\mathsf{action}\ o.a(\overline{v})\{sc\}$ extends the related history with a new action $A = (o.a, \overline{v})$. Finally, with the $\mathsf{reinit}(L, R)$ command, the related history sequence in $\mathsf{HistMap}$ is emptied, and the values of $l \in L$ are copied from Heap to $\mathsf{InitHeap}$. There is no rule for the command $v.\mathsf{commit}()$; operationally this is a no-op.

Resources Our reasoning system is based on the concept of *resources* [1]. This means that we do not reason directly over the global state, but over a partial abstraction of the state, i.e., a resource. Intuitively, a resource describes how the thread that we reason about views the program state.

A resource $\mathcal{R}$ is a tuple $(h, h_i, \mathcal{P}, \mathcal{P}_h, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{H}, \mathcal{A})$, where each component abstracts part of the state: *i)* h represents the (*partial*) heap, containing only locations for which $\mathcal{R}$ has a positive permission; *ii)* h_i is the (*partial*) initial heap, contains only locations for which $\mathcal{R}$ has a positive history fraction; *iii)* $\mathcal{P} \in \mathsf{Loc} \mapsto [0, 1]$ is a *permission table* that defines the permission that

$$
\begin{array}{ll}
\mathcal{R}; s \models \mathsf{Perm}(e.f, \pi, e') & \iff [\![e]\!]^h_s = o, \mathcal{P}(o,f) \geq \pi, h(o.f) = [\![e']\!]^h_s, \\
& \quad (o,f) \in \mathsf{dom}(h_i),\ \exists L \in \mathsf{dom}(\mathcal{H}).\ (o,f) \in L \\
\mathcal{R}; s \models \mathsf{PointsTo}(e.f, \pi, e') & \iff [\![e]\!]^h_s = o, P(o,f) \geq \pi, h(o.f) = [\![e']\!]^h_s, \\
& \quad h_i(o,f) = \bot,\ \forall L \in \mathsf{dom}(\mathcal{H}).\ (o,f) \notin L \\
\mathcal{R}; s \models F * G & \iff \exists \mathcal{R}_1, \mathcal{R}_2. \mathcal{R} = \mathcal{R}_1 * \mathcal{R}_2, \mathcal{R}_1; s \models F \wedge \mathcal{R}_2; s \models G \\
\mathcal{R}; s \models \mathsf{Hist}(L, \pi, R, H) & \iff \forall (e.f) \in L\ [\![e]\!]^h_s = o, \mathcal{P}_h(o,f) \geq \pi,\ h_i(o.f) = v, \\
& \quad R[v/e.f]_{\forall (e.f) \in L} = \mathsf{true},\ \mathsf{filter}(\mathcal{H}(o,f)) \in CT_G(H) \\
\mathcal{R}; s \models \mathsf{APerm}(e.f, \pi, e') & \iff \mathcal{R}; s \models \mathsf{Perm}(e.f, \pi, e') \wedge o.f \in \mathcal{A},\ [\![e]\!]^h_s = o \\
\mathcal{R}; s \models e & \iff [\![e]\!]^h_s = \mathsf{true} \\
\mathcal{R}; s \models e.P & \iff \mathcal{R}; \emptyset \models F\ \mathit{pred\_body}(o.P) = F \quad o = [\![e]\!]^h_s \\
\mathcal{R}; s \models F \wedge G & \iff \mathcal{R}; s \models F \wedge \mathcal{R}; s \models G \\
\mathcal{R}; s \models F \vee G & \iff \mathcal{R}; s \models F \vee \mathcal{R}; s \models G \\
\mathcal{R}; s \models \forall T x F & \iff \forall \Gamma' \supseteq \Gamma,\ \mathcal{R}' \geq \mathcal{R}, \Gamma' \vdash v : T \Rightarrow \Gamma \vdash \mathcal{R}'; s \models F[v/x] \\
\mathcal{R}; s \models \exists T x F & \iff \exists v. \Gamma \vdash v : T \wedge \Gamma \vdash \mathcal{R}; s \models F[v/x] \\
\mathcal{R}; s \models e.\mathsf{fresh}() & \iff [\![e]\!]^h_s \in \mathcal{F} \\
\mathcal{R}; s \models e.\mathsf{initialized}() & \iff [\![e]\!]^h_s \in \mathcal{I}
\end{array}
$$

Fig. 3. Semantics of formulas $\mathcal{R} = (h, h_i, \mathcal{P}, \mathcal{P}_h, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{H}, \mathcal{A})$

$\mathcal{R}$ has for a given location; *iv)* $\mathcal{P}_h \in \mathsf{Loc} \mapsto [0,1]$ is a *history fraction table* that for a location l defines the fraction owned by $\mathcal{R}$ for the history predicate referring to l; *v)* $\mathcal{J} \subseteq \mathsf{ObjId}$ keeps the set of threads that can be joined; *vi)* $\mathcal{L} \in \mathsf{ObjId} \mapsto \mathsf{Set}(\mathsf{ObjId})$ abstracts the lock table, mapping each thread to the set of locks that it holds; *vii)* $\mathcal{F} \subseteq \mathsf{ObjId}$ keeps a set of fresh locks; *viii)* $\mathcal{I} \subseteq \mathsf{ObjId}$ keeps a set of initialised locks; *ix)* $\mathcal{H}$: $\mathsf{Set}(\mathsf{Loc}) \mapsto \overline{\mathsf{Action} \times \mathsf{bool}}$ abstracts the history map, marking every action with a boolean flag to indicate whether it is owned by $\mathcal{R}$; and *x)* $\mathcal{A} \subseteq \mathsf{Loc}$ stores locations referred by an action in progress.

Resources owned by different threads should be *compatible*, written $\mathcal{R}_1 \# \mathcal{R}_2$. For example, $\mathcal{R}_1 \# \mathcal{R}_2$ ensures that the sum of permissions to the same location in $\mathcal{R}_1$ and $\mathcal{R}_2$ does not exceed 1, or the same action from the history map is not owned by both $\mathcal{R}_1$ and $\mathcal{R}_2$. When threads join, their associated resources are *joined* into a resource $\mathcal{R}_1 * \mathcal{R}_2$. For the definition of both $\#$ and $*$ we refer to [3].

Semantics of formulas Fig. 3 presents the semantics of formulas. With $\mathcal{R}; s \models F$ we denote that the formula F is valid with respect to a resource $\mathcal{R}$ and a stack s. The predicate $\mathsf{Hist}(L, \pi, R, H)$ is valid when: the resource $\mathcal{R}$ contains at least a fraction π of the related history; R holds over the values from the initial heap, and $\mathsf{filter}(\mathcal{H}(o,f))$ belongs to $\mathsf{Traces}(H)$. The function $\mathsf{filter}(\mathcal{H}(o,f))$ returns the subsequence of the sequence $\mathcal{H}(o,f)$ with only those actions owned by $\mathcal{R}$, i.e., the actions marked with the flag true. The predicate $\mathsf{APerm}(e.f, \pi, e')$ states that $\mathcal{R}$ contains at least permission π for the location $e.f$, and that there exists an action in progress that refers to $e.f$.

Proof rules Fig. 4 presents the most relevant proof rules. We use $\circledast_i F_i$ to abbreviate a separation conjunction over all formulas F_i. Rules $[\mathit{ReadH}]$ and $[\mathit{WriteH}]$ state that accessing a location is allowed if an action is in progress, while $[\mathit{Read}]$ and $[\mathit{Write}]$ can only be used when there is no history maintained for the accessed location. The $[\mathit{Action}]$ rule describes that if the action implementation satisfies the action's contract, the action will be recorded in the history. The

$$[Read] \quad \{\mathsf{PointsTo}(v.f, \pi, w)\}\ j = v.f\ \{\mathsf{PointsTo}(v.f, \pi, w) * j == w\}$$

$$[Write] \quad \{\mathsf{PointsTo}(v.f, 1, \_)\}\ v.f = w\ \{\mathsf{PointsTo}(v.f, 1, w)\}$$

$$[ReadH] \quad \{\mathsf{APerm}(v.f, \pi, w)\}\ j = v.f\ \{\mathsf{APerm}(v.f, \pi, w) * j == w\}$$

$$[WriteH] \quad \{\mathsf{APerm}(v.f, 1, \_)\}\ v.f = w\ \{\mathsf{APerm}(v.f, 1, w)\}$$

$$[Create] \quad \{\circledast_{\forall v.f \in L} \mathsf{PointsTo}(v.f, 1, w) * R\}\mathsf{crhist}(L, R)\{\circledast_{\forall v.f \in L} \mathsf{Perm}(v.f, 1, w) * \mathsf{Hist}(L, 1, R, \epsilon)\}$$

$$[Destr] \quad \{\circledast_{\forall v.f \in L} \mathsf{Perm}(v.f, 1, w) * \mathsf{Hist}(L, 1, R, \epsilon)\}\mathsf{dsthist}(L)\{\circledast_{\forall v.f \in L} \mathsf{PointsTo}(v.f, 1, w)\}$$

$$[Action] \quad \frac{\begin{array}{c} act ::= \ \mathsf{requires}\ F\ \mathsf{ensures}\ F'\mathsf{accessible}\ L_a\ a(\overline{i}); \quad L_a \in L; \quad \sigma = \overline{w}/\overline{i} \\ \{\circledast_{\forall l \in L_a} \mathsf{APerm}(l, \pi_l, u) * F[\sigma]\} c \{\circledast_{\forall l \in L_a} \mathsf{APerm}(l, \pi_l, v) * F'[\sigma]\} \end{array}}{\begin{array}{c} \{\circledast_{\forall l \in L_a} \mathsf{Perm}(l, \pi_l, u) * \mathsf{Hist}(L, \pi, R, H) * F[\sigma]\} \\ \mathsf{action}\ v.a(\overline{w})\{sc\}; \\ \{\circledast_{\forall l \in L_a} \mathsf{Perm}(l, \pi_l, v) * \mathsf{Hist}(L, \pi, R, H \cdot v.a(\overline{w})) * F'[\sigma]\} \end{array}}$$

$$[Reinit] \quad \frac{\forall w \in \mathsf{Traces}(H).\{R\}w\{R'\}}{\{\mathsf{Hist}(L, 1, R, H)\} \quad \mathsf{reinit}(L, R') \quad \{\mathsf{Hist}(L, 1, R', \epsilon)\}}$$

$$[SplitMergeHist] \quad \frac{H = H_1 \parallel H_2, \pi = \pi_1 + \pi_2}{\mathsf{Hist}(L, \pi, R, H) \mathrel{-\!\!*} \mathsf{Hist}(L, \pi_1, R, H_1) * \mathsf{Hist}(L, \pi_2, R, H_2)}$$

$$[Sync] \quad \frac{\gamma(s, \overline{s}) = \tau}{\mathsf{Hist}(L, \pi_1, R, H_1) * \mathsf{Hist}(L, \pi_2, R, H_2) \mathrel{-\!\!*} \mathsf{Hist}(L, \pi_1, R, H_1 \cdot s) * \mathsf{Hist}(L, \pi_2, R, H_2 \cdot \overline{s})}$$

Fig. 4. Selected set of proof rules

premise in the [*Reinit*] rule requires that the Hoare triple $\{R\}w\{R'\}$ holds for every trace $w \in \mathsf{Traces}(H)$, where w is a sequential program. [*SplitMergeHist*] and [*Sync*] define how history predicates can be exchanged for each other.

Soundness We define correctness of our system (see [3] for the proof sketch):

Theorem 1. *Let $\{F\}c\{G\}$ be derivable, and let $\sigma \rightsquigarrow^\star \sigma'$ If $\mathcal{R}$ is a resource that abstracts the program state σ and $\mathcal{R}, s \models F$, then for any $\mathcal{R}'$ such that abstracts σ', $\mathcal{R}', s' \models G$.*

Tool support We have integrated our technique in VerCors [2], a tool for verifiying concurrent programs written in languages such as Java and C annotated with separation logic-based specifications. To verify programs with histories, the tool checks: i) whether each action segment satisfies the contract of the action; ii) whether every trace of a history H satisfies its contract (see the [Reinit] rule, Fig. 4). For this step we use a linearisation-based technique [8] that requires unfolding H only until it is in a *guarded* form from which (with the help of user specification) the contract of H can be proved. We give more detail in [3].

5 Conclusions and Related Work

This paper extends permission-based separation logic with *histories*, i.e., a mechanism that allows one to reason about functional behaviour of coarse-grained

concurrent programs, while providing simple and intuitive method specifications. We have added support for the approach to the VerCors tool set [2].

Related work Jacobs and Piessens extend the Owicki-Gries technique to allow modular reasoning about functional properties [11]. Their logic allows one to augment the client program with auxiliary code that is passed as an argument to methods. Additionally, a concrete invariant property should be specified that remains stable under the updates of all threads; however, defining such an invariant is often difficult. Another similar approach are *Concurrent Abstract Predicates (CAP)* [6], which extend 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 client threads 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 CAP) [19], where a CAP may be parametrised by a protocol defined by the client. Compared to these approaches, we believe that histories allow more natural specifications, where there is no need of specifying complex invariants or protocols.

Strongly related to our work is the recently proposed prototype logic of Ley-Wild and Nanevski [14], the *Subjective Concurrent Separation Logic (SCSL)*. They extend PBSL with the *subjective separating conjunction* operator, $\circledast$, which splits and merges a heap such that the contents of a given location may also be split: $l \mapsto a \oplus b$ is equivalent to $l \mapsto a \circledast l \mapsto b$. The user specifies a *partial commutative monoid (PCM)*, $(\mathbb{U}, \oplus, \mathbb{0})$, with a commutative and associative operator $\oplus$ that combines the effect of two threads. To solve the Owicki-Gries example, a PCM $(\mathbb{N}, +, 0)$ is chosen: local contributions are combined with the $+$ operator. However, if we extend this example with a third parallel thread that for example multiplies the shared variable by 2, we expect that the choice of the PCM will become troublesome. With our approach, in a way we use a PCM where contributions of threads are expressed via histories, and these threads effects are combined by the process algebra operator $\|$. This makes our approach easily applicable to various examples (including the one described above). Moreover, our method is also suited to reason about programs with dynamic thread creation.

Closely related to our approach is the work on *linearisability* [20], where linearisation points roughly correspond to our action specifications. Using linearisation points 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 coarse-grained program. The idea behind the code transformation is that consecutive actions are merged to increase atomicity up to the desired level. Recently, a more powerful form of linearisation has been proposed, where multiple synchronisation commands can be abstracted into one single linearisation action [10]. It might be worth investigating if these ideas carry over to our approach, by adding different synchronisation actions to the histories.

Recently, some promising parameterisable logics have been introduced [5, 12] to reason about multithreaded programs. The concepts that they introduce are

very close to our proof logic. Reusing such a framework will simplify the formalisation and justify soundness of our system, as well as show that the concept of histories is applicable in other variations of separation logic. However, to the best of our knowledge, in their current form, these frameworks are not directly applicable to our language as they do not support dynamic thread creation.

References

1. A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded java programs. *CoRR*, abs/1411.0851, 2014.
2. S. Blom and M. Huisman. The VerCors Tool for verification of concurrent programs. In *Formal Methods*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.
3. S. C. C. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. Technical report, Enschede, 2015. Available at http://eprints.eemcs.utwente.nl/25866/01/tech_report.pdf.
4. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.
5. T. Dinsdale-Young, L. Birkedal, P. Gardner, M. J. Parkinson, and H. Yang. Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300, 2013.
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*, 2009.
8. J. Groote, A. Ponse, and Y. Usenko. Linearization in parallel pCRL. *The Journal of Logic and Algebraic Programming*, 48(12):39 – 70, 2001.
9. J. F. Groote and M. A. Reniers. Algebraic process verification. In *Handbook of Process Algebra, chapter 17*, pages 1151–1208. Elsevier.
10. N. Hemed and N. Rinetzky. Brief announcement: Contention-aware linearizability. In *PODC 2014*, 2014.
11. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.
12. R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. Accepted for publication at POPL 2015.
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. R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, 2013.
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*, pages 55–74. IEEE Computer Society, 2002.
18. J. Smans, B. Jacobs, and F. Piessens. VeriFast for Java: A tutorial. In *Aliasing in Object-Oriented Programming*, pages 407–442. Springer, 2013.
19. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.
20. V. Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464, 2010.
21. M. Zaharieva-Stojanovski and M. Huisman. Verifying class invariants in concurrent programs. In *FASE*, pages 230–245, 2014.