History-based Verification of Functional Behaviour of Concurrent Programs

Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski

University of Twente, the Netherlands

Abstract. Modular verification of the functional behaviour of a concurrent program remains a challenge. We propose a new way to achieve this, using histories, modelled as process algebra terms, to keep track of local changes. When threads terminate or synchronise in some other way, local histories are combined into global histories, and by resolving the global histories, the reachable state properties can be determined. Our logic is an extension of permission-based separation logic, which supports expressive and intuitive specifications. We discuss soundness of the approach, and illustrate it on several examples.

1 Introduction

Verification of *functional properties* of concurrent programs remains a major challenge. First of all, all possible interleavings between the parallel threads have to be considered. Moreover, to make verification scale, a modular approach is needed, which requires that the behaviour of each program component (such as methods and threads) is specified. However, due to interference between parallel threads, it is non-trivial to specify the behaviour of a thread or method. In particular, all local specifications should be *stable*, i.e., it should not be possible for any other thread to invalidate them.

Currently, several successful modular techniques exist to reason about *data-race freedom* of concurrent programs [6, 16, 5]. However, ever since Owicki and Gries [24, 23] proposed the first (non-modular) verification technique for concurrent programs, complicated extensions have been necessary to reason also about functional properties. Before we explain our solution to this problem, we will give a brief overview of the Owicki-Gries's approach.

Owicki-Gries The example in Lst. 1 originates from Owicki and Gries's seminal paper [24]: 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 resource r. If the value of x initially was 0, we would like to be able to prove that at the end, after both threads have finished their updates, the value of x equals 2.

Owicki and Gries's solution to verify this program uses *auxiliary (specification only) variables*. Each thread has its own auxiliary variable (a and b, respectively) to keep track of the state of each individual thread. Lst. 2 shows the full proof outline of this program. A *resource invariant* $I(r)$ specifies the *invariant property*

```
resource r(x): cobegin
  with r when true do
       x:=x+1
||
  with r when true do
       x:=x+1
coend
```

Lst. 1. A shared Counter data structure

```
a:=0; b:=0; I(r) = {x=a+b}
/*a=0 & b=0 & I(r)*/
resource r(x,a,b): cobegin
  /*a = 0*/ with r when true do
        begin x:=x+1; a:=1; end
  /*a = 1*/
||
  /*b = 0*/ with r when true do
        begin x:=x+1; b:=1; end
  /*b = 1*/
coend
/*a=1 & b=1 & I(r)*/
/*x=2*/
```

Lst. 2. Counter-proof outline

that relates the value of x to the auxiliary variables a and b. Thus, all local changes have to be tracked explicitly by auxiliary variables that are specified by the programmer. However, as observed by Jacobs and Piessens [15], this approach does not generalise to method calls: if the code for acquiring resource r and updating the value of x is inside a method *incr*, which is called by both threads, then each thread requires an update on a different auxiliary variable.

Our Approach This paper proposes an alternative approach for reasoning about coarse-grained data structures, based on using *histories*. A history is a *process algebra* term (we use μCRL [12] as an expressive process algebra with data) that abstracts *part of the program*, by capturing the relevant program executions in the form of *abstract actions*. Therefore, reasoning about the functional behaviour of the original program is done by reasoning directly about the abstracted model.

A history traces the behaviour of a chosen set of shared locations L. The protocol for using histories is the following. The client has some initial knowledge (a predicate R) about the values of the locations in L. When these locations become shared by multiple threads, the client creates an empty history ($H = \epsilon$) over L. Thereafter, all updates of locations from L must be recorded in H.

To allow building the history in a modular way, the history is represented by a *splittable* predicate $\mathsf{Hist}(L, 1, R, H)$. A fraction π of the predicate is denoted by $\mathsf{Hist}(L, \pi, R, H)$ ($0 < \pi \leq 1$). If $\pi = 1$, the history is *global* (complete), while for $\pi < 1$ we say that the history is *local* (incomplete). When threads run in parallel to operate on the history over L, each thread obtains a local history to record its actions to. When threads are finished, their local histories are merged, and a global history $\mathsf{Hist}(L, 1, R, H)$ is obtained, from which the possible new values of the shared locations can be derived. The global history is an abstraction of the behavour of the locations in L, between the *initial state* of the history, i.e., the state when the history was empty, and the current state.

The approach is based on a variant of *permission-based separation logic* [5, 2]. As a novelty, we extend the definition of the *separating conjunction* $*$ to

histories. In particular, histories are *merged* using the following rule (where the operator *-* is read as "splitting" (from left to right) or "merging" (from right to left), and $\|$ is the "merge" process algebra operator):

$$\mathsf{Hist}(L, \pi_1 + \pi_2, R, H_1 \parallel H_2) \mathrel{*\!\!-\!\!*} \mathsf{Hist}(L, \pi_1, R, H_1) * \mathsf{Hist}(L, \pi_2, R, H_2)$$

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 (discussed above), we first specify an action a, describing the update of the location x. The behaviour of the method *incr* is specified as an extension of a local history with the action $a(1)$. Importantly, local histories are used only by the current thread and therefore, are invariant to the executions from the environmental threads. This makes history-based specifications *stable*.

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

```
//@requires Hist(L, π, R, H) * x ∈ L);
//@ensures  Hist(L, π, R, H · a(1)),
  void incr(){};
```

As stated above, when threads are joined, we obtain the $\mathsf{Hist}(L, 1, R, H)$ predicate. Based on the collected information in the history H and the knowledge R in the initial state, we can prove properties about the current state. Concretely, to prove that a property R' holds, we analyse all traces of the history and prove that R' holds after the execution of any of these traces: $\forall w \in \mathsf{Traces}(H).\{R\}w\{R'\}$. Note that such a trace is a sequence of actions, each with a pre- and postcondition; thus this boils down to reasoning about a sequential program. In the example above, we obtain a history $H = a(1) \parallel a(1)$. From H and the initial knowledge $x == 0$, we can deduce that $x == 2$.

The history is built modularly, allowing modular and *intuitive* specifications. Reasoning about the history H, however, involves reasoning about all traces in H; this is done in a non-modular way. However, we do not consider this as a serious weekness 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; 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 the behaviour of coarse-grained concurrent programs that only requires intuitive specifications from the programmer. We provide a formalisation and soundness proof of the approach. Our technique has been integrated in the VerCors tool set [4, 1]. Moreover, it has been experimentally added on top of the VeriFast logic [26].

Outline Sec. 2 reviews some background on process algebra and permission-based separation logic. Sec. 3 gives full details of our approach, which is then formalised

and proven sound in Sec. 4. Sec. 5 presents the encoding of our technique in our tool and finally, Sec. 6 concludes, and discusses related and future work.

2 Background

The µCRL Language To model histories, we use µCRL [12], i.e., a process algebra with data. µCRL allows reasoning about the behaviour of concurrent systems by describing them in terms of algebraic *process expressions*. µCRL is based on ACP (Algebra of Communicating Processes) [3].

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 without behaviour). *Processes* $\{p_1, p_2, ...\}$ are defined by combining actions and recursion variables, which (with the exception of the special actions δ and τ) may be parameterised by data.

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. In addition to these operators (which are part of ACP), µCRL includes 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. With ϵ we denote the *empty process*.

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

Separation Logic and Permissions Permission-based separation logic (PBSL) [25, 22, 2] extends Hoare Logic [14] to reason about multithreaded programs. Separation logic contains the *separating conjunction* operator ($*$): $P{*}Q$ holds when P and Q describe two *disjoint* resources.

To allow simultaneously read accesses to the same location, PBSL associates a fractional permission π to every heap location. A permission is modelled as a value in the domain $(0, 1]$ [5]. The logic depends on whether a thread holds a permission to access a location. To change a location x, a thread must hold a *write* permission to x, i.e., $\pi = 1$; while for reading a location, any *read* permission is required, i.e., $\pi > 0$. Soundness of the logic ensures that the sum of all threads' permissions to access a certain location never exceeds 1, which guarantees that a verified program is free of data-races.

```
class Counter {
  int x;
//@pred inv = Perm(x,1,v);
 Lock lock = 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(){
  lock.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)}*/
  lock.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. 3. The Counter example

A permission π to a location x is expressed by the predicate $\mathsf{PointsTo}(x, \pi, v)$, where v denotes the value stored on the location x. This predicate is splittable, and thus, parts of the predicate may be distributed and used by parallel threads.

Locks To reason about locks, we use the protocol described by Haack et al. [2]. Following Owicki and Gries and O'Hearn [24, 22], for each lock, they associate a special predicate *inv*, called a *resource invariant*, describing which locations are protected by the lock. A newly created lock is still *fresh* and not ready to be acquired. The thread must first execute a (specification-only) *commit* command on the lock that transfers the permissions from the thread to the lock and changes the lock's state to *initialized*. Any thread then may acquire the initialised lock to obtain the resource invariant. Upon release of the lock, the thread returns the resource invariant back to the lock.

3 Modular History-Based Reasoning

This section gives an informal but detailed description of our methodology. To illustrate our approach, we use a Java-like variant of the classical Owicki-Gries example, presented in Lst. 3. Class *Counter* defines a shared counter, where location x can be accessed only by a thread holding the lock.

To specify this program, the classical approach is to associate a predicate to the lock, defined as $inv = \mathsf{PointsTo}(x, 1, v)$ [22, 2]. However, the $\mathsf{PointsTo}$ predicate stores not only the access permission to x, but also information about the value of x. As the method *incr* uses internal synchronisation, after the lock

is released, the PointsTo predicate is transferred to the lock, and therewith, all information about the value of x is lost. This makes describing the method's functional behaviour in the postcondition problematic.

With our technique, a resource invariant can be used to store permissions to access a location, while information about the value stored at this location is treated separately. In particular, in the method's post-state of the example, we can not specify the *complete knowledge* of the value of x, but we can express some *partial knowledge*, i.e., the contribution of the current thread within the method. This knowledge is expressed via a history over x, a process algebra expression built of actions that represent changes to x. Partial histories can be used later by the client: by joining all threads, the client combines the partial knowledge to reconstitute complete knowledge of the behaviour of x.

Histories A history refers to a set of locations L and is called *a history over* L. It records all updates made to any of the locations in L. The same location can not appear in more than one existing history simultaneously.

We use a predicate $\mathsf{Hist}(L, 1, R, H)$ to capture a history over locations L. This contains complete knowledge about the changes to the locations in L. In particular, the predicate R captures the knowledge about the values of the locations in L in an *initial state* σ, i.e., a state when no action has been recorded in the history. More precisely, R is a predicate over L, such that $R[\sigma(l)/l]_{\forall l \in L}$ holds, where $\sigma(l)$ denotes the value of l in state σ. Further, history H is an µCRL process [12], which records the *behaviour of* L, i.e., the *history of updates* over locations in L. The second parameter π in the Hist predicate is used to make it a splittable predicate. Each part of a split predicate contains only *partial* knowledge about the behaviour of L.

Creating a History A history over L is created by the specification command $\mathsf{crhist}(L, R)$. It requires a *full* PointsTo predicate for each location $l \in L$ as a precondition. Every $\mathsf{PointsTo}(l, 1, v)$ predicate is exchanged for a new $\mathsf{Perm}(l, 1, v)$ predicate, which essentially has the same meaning as PointsTo: a splittable predicate that keeps the access permission for the location l and its current *local* value v. However, having a $\mathsf{Perm}(l, \pi, v)$ predicate indicates that there also exists a history that refers to l, and every change of l must be recorded in this history. Consuming the PointsTo predicate when creating the history ensures that the same location l can be traced by at most one history at the time. Additionally, the $\mathsf{crhist}(L, R)$ also returns a Hist predicate with an empty history, $H = \epsilon$, where R is a predicate that characterises the initial values for the variables in L.

In the example in Lst. 3, the lock's resource invariant is defined as the Perm predicate, instead of PointsTo (line 3). This means that while the permission to update x is stored in the lock, independently there exists a history that refers to x and records all updates to x. The client creates the *Counter* object, obtaining the full PointsTo predicate (line 32). It then creates a history over a single location x (line 36) and exchanges the PointsTo predicate for the predicates Perm and Hist. After the lock is committed (line 38) and the permissions are transferred to the

lock, the client still keeps the full Hist predicate. This guarantees that no other thread may update the location x until the Hist predicate is split; the value is stable even without holding any access permission to x.

Splitting and Merging of Histories The history may be redistributed among parallel threads by splitting the predicate $\mathsf{Hist}(L, \pi, R, H)$ into two separate predicates, with histories H_1 and H_2, such that $H = H_1 \parallel H_2$. Each predicate is used by one parallel thread, and each thread records its own updates in its own partial history. The basic idea is to split the history H such that $H_1 = H$ and $H_2 = \epsilon$. However, this should be done in such a way that if we later merge the two histories, we know at which point H was split. More specifically, if we split H, and then one thread does an action a, and the other thread an action b, and then the histories are merged, this should result in 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 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* (for convenience, we usually denote two synchronisation actions with s and $\overline{s}$). It is safe to add such a synchronisation barrier, because we know that all actions in the history so far must happen before this synchronisation. When the threads are joined, all partial histories over the same set of locations L are *merged* together. To allow merging histories, we require that each thread is joined at most once in the program.

In Lst. 3 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.

Recording Updates in a History

Action Definition To record updates of locations in the history, we extend the specification language with *actions*. Each action is defined by an *action name* and a list of parameters. An action is equipped with an *action specification*: 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:

```
/*@ accessible footprint
  @ assignable modified_locations
  @ requires precondition
  @ ensures postcondition
  @ action actName (parameters); */
```

Lst. 3 shows a definition of an action *inc* (lines 6 - 10), which represents an increment of the location x by one. Note that the action contract is written in a pure JML language [19], without the need to explicitly specify permissions, as they are treated separately. In particular, action contracts are used to reason about a trace of a history, which (as discussed above) is a sequential program.

Action Implementation An action may be associated with a program segment that implements the action specification. For this purpose, we introduce a specification command $\mathsf{action}\ a(\overline{v})\{sc\}$, which marks the program block sc as an implementation of the action a with parameters $\overline{v}$. We call sc an *action segment*. In Lst. 3, we specify an action segment of the action *inc* in lines 17 - 21.

Recording Actions In the prestate of the action segment, a history predicate $\mathsf{Hist}(L, \pi, R, H)$ is required, which captures the behaviour of the footprint locations of the action a. i.e., $\forall l \in \mathsf{footprint}(a).l \in L$. At the end of the action segment, the action is recorded in the history. For this, it is necessary that the action segment implements the specification of the action a. For example, in Lst. 3 the history H is extended with an action $inc(1)$, line 22.

Restrictions within an Action As discussed above, 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. Moreover, a modified location should not be visible by other threads until the action finishes. Furthermore, the same thread must not record the same update more than once in the history. Thus, a thread can not have started more than one action over the same location simultaneously.

To ensure this, we impose several restrictions on what is allowed in the action segment (a formal definition is given in Sec. 4.1). In the prestate of the action a, we require that the current thread has a positive permission to every footprint location of a. Within the action segment we forbid the running thread to release permissions and to make them accessible to other threads. 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, or starting another action.

In this way, we allow two actions to interleave only if they refer to disjoint sets of locations, or if their common locations are only readable by both threads. We also allow a single thread to have at most one started action at a time. 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 an example where these restrictions become problematic.

Updates within an Action As discussed in Sec. 2, in standard PBSL, accessing a heap location l requires a positive permission, i.e., the $\mathsf{PointsTo}(l, \pi, v)$ predicate. With our approach, if a history H over l exists, the access permission to l is provided by the $\mathsf{Perm}(l, \pi, v)$ predicate. Every update to l must then be a part of an action that will be recorded in H. Thus, the permission that the $\mathsf{Perm}(l, \pi, v)$ predicate provides is *valid* only within an action segment with a footprint that refers to l. Thus, within the action segment, the $\mathsf{Perm}(l, \pi, v)$ predicates are exchanged for predicates $\mathsf{APerm}(l, \pi, v)$, which give right to the thread to access the location l. Therefore, our logic allows accessing a shared location when the running thread holds an appropriate fraction of either the $\mathsf{PointsTo}$ or the APerm predicate. The example in Lst. 3 illustrates this on lines 17 - 21.

History Reinitialisation 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. The state of these locations is then *stable* and no other thread can update them. 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 on the current state. Thus, reasoning about the continuation of the program will be done with an initial empty history.

We add a $\mathsf{reinit}(L, R')$ specification command, which converts the full predicate $\mathsf{Hist}(L, 1, R, H)$ 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 the process H from a state satisfying R. As discussed above, this requires that R' holds after the execution of any of the traces of H: $\forall w \in \mathsf{Traces}(H).\{R\}w\{R'\}$.

In Lst. 3, 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 any permission to access x. However, holding the full Hist predicate is enough to reason about the current value of x.

Destroying a history It is possible to obtain the PointsTo predicates back for the locations that are traced in a history. This is done by *destroying the history*, by 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, in particular, this will allow the client to create a history predicate over a different set of locations.

An Example with Recursion and Multiple Locks We illustrate our approach on a more involved example, which includes recursive method calls and a shared location that is protected by two different locks. Consider an extended class *ComplexCounter* (Lst. 4) with three fields: *data*, x and y. It has two locks: *lockx* protects write access to x and read access to *data*, while *locky* protects write access to y and read access to *data*. If a thread holds both *lockx* and *locky*, it has write access to *data*.

Methods $addX()$ and $addY()$ increase respectively x and y by *data*, while the $incr(n)$ is a recursive method that increments *data* by n. The synchronised code in the methods $addX()$, $addY()$ and $incr(n)$ is associated with an appropriate action (lines 36, 45, 55). To specify the $incr(n)$ method, we additionally 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. 5 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 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. 4. Complex Counter example

$c.y$ (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.

Obviously, the values of $c.x$ and $c.y$ at the end depend on the moment when $c.data$ has been updated. Thus, the history should trace the updates of all three locations, $c.x, c.y$ and $c.data$. Each thread then 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 the behaviour of the program, 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. 5), we could prove that the property $R' = 10 \leq c.x + c.y \leq 40$ holds in the current state. The history predicate is then reinitialised to $\mathsf{Hist}(L, 1, R', \epsilon)$.

The example shows that our technique also allows reasoning about more complicated scenarios in which the same location is protected by different locks. By using a technique based on the Owicki-Gries method, providing a concrete resource invariant for every lock that describes certain behaviour would be rather difficult. With our approach, we make a clear separation between permissions and behaviour of locations. Thus, while the lock stores the permissions, the behaviour is captured independently by the history. As a result, the specification of this example remains equally intuitive and simple as the *Counter* example.

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

Lst. 5. Complex Counter example - the Client class

Reasoning about Concurrent Data Structures Finally, we illustrate how to use histories to reason about functional properties of more complex coarse-grained concurrent data structures. Lst. 6 presents a *Set* data structure that represents a set of integers. The structure is implemented as a linked list with unique elements. The client thread creates an empty set and adds the element 2 to the set. The set is then shared between three parallel threads: thread $t1$ adds the element 4 to the set (if it is not there), thread $t2$ removes the element 6 (if it is in the set) and thread $t3$ adds the element 6 (if it is not there). At the end when threads are joined, we prove that the elements 2 and 4 exist in the set.

The example includes details that are not discussed in the paper (as they are orthogonal to the main ideas of our history-based reasoning). Concretely, we use *ghost* (specification-only) fields and *specification data types*. Therefore, we just shortly discuss how we reason about this example. We associate the *Set* data structure with a representative *ghost* field *ss* (line 4) (which has a sequential data type *sset*). Additionally, the resource invariant ensures that the sequential ghost field is always *compatible* with the actual data structure (lines 61, 62). Furthermore, we define a history over the ghost field. Therefore, method contracts are expressed in terms of local changes to this history. After threads are joined, we use the history to reason about the structure of the sequential set *ss*, while the resource invariant is used to guarantee that it has the same content as the actual data structure.

```
class Set{

  Node first;
//@ ghost sset ss;

/*@ pred state(sset ss) =
  @ PointsTo(first,1,u) *
  @ first == null ⇒ ss==∅ *
  @ first ≠ null ⇒ first.state(ss);
  @ pred pinv = Perm(ss, 1, v) * state(ss);
@*/
 Lock lock = new Lock/*@<pinv>@*/();

/*@ accessible {ss};
  @ assignable {ss};
  @ ensures ss=\old(ss) ∪{k}
  @ action a(int k)

  @ accessible {ss};
  @ assignable {ss};
  @ ensures ss=\old(ss) \{k}
  @ action r(int k)
@*/

//@ requires PointsTo(ss, 1, v) * state(ss);
//@ ensures Hist({ss}, 1, ss==v, ϵ);
  void init(){
//@ crHist({ss}, ss==v);
/* Hist({ss}, 1, ss==v, ϵ) *
   Perm(ss, 1, v) * state(ss) */
//@ lock.commit();
  }

//@ requires Hist{{ss}, 1, R, H));
//@ ensures Hist{{ss}, 1, R, H·a(data)));
  void add(int data){
    lock.lock();
//... add data if it is not already in the set
  //@ action a(data){
  //@ ss = ss ∪{data};
  //@ }
   lock.unlock();
  }

//@requires Hist{{ss}, 1, R, H));
//@ensures Hist{{ss}, 1, R, H · r(data)));
  void remove(int data){
    lock.lock();
// ... remove data if it is in the set
  //@ action r(data){
  //@ ss = ss \{data};
  //@ }
    lock.unlock();
  }
}

class Node {
  int data; Node next;
//@ pred state(sset ss) = PointsTo(data, 1, v) *
  @ PointsTo(next, 1, u) *
  @ next = null ⇒ ss ={data} *
  @ next ≠ null ⇒ data ∈ ss * next.state(ss\{data})
//...
}

class Client{
 Thread1 t1; Thread2 t2; Thread3 t3;
 void main(){
  Set s = new Set();
/* PointsTo(s.ss, 1, ∅) * s.state(ss) */
 s.init();
/* Hist({s.ss}, 1, s.ss==∅, ϵ) */
  set.add(2);
/* Hist({s.ss}, 1, s.ss==∅, ss.a(2)) */
  t1 = new Thread1(s);
  t2 = new Thread2(s);
  t3 = new Thread3(s);
  t1.fork();  //ty calls s.add(4)
  t2.fork();  //ty calls s.remove(6)
  t3.fork();  //ty calls s.add(6)
  t1.join(); t2.join(); t3.join();
/*Hist({s.ss}, 1, s.ss==∅,
ss.a(2)·(ss.a(4) ‖ ss.r(6) ‖ ss.a(6))) */
//@ reinit({s.ss}, {2,4} ⊆ {s.ss})
/*Hist({s.ss}, 1, {2,4} ⊆ {s.ss}, ϵ) */
 }
}
```

Lst. 6. A Set data structure example

4 Formalisation

To formalise our approach, we use a Java-like language, to show the applicability of our technique in an environment with creation of dynamic threads. 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 [2] to reason about Java-like programs.

4.1 Language

Figure 1 presents the syntax of our language. For convenience, we distinguish between *read-only* and *read-write* variables. Apart from the special actions (δ, τ), two kinds of actions are allowed: synchronisation actions $s \in \mathsf{SAct}$ and update

$n \in \mathsf{int} \quad b \in \mathsf{bool} \quad o,t \in \mathsf{ObjId} \quad \pi \in (0,1] \quad i \in \mathsf{RdVar} \quad j \in \mathsf{RdWrVar}$
$x \in \mathsf{Var} = \mathsf{RdVar} \cup \mathsf{RdWrVar} \quad a(\overline{v}) \in \mathsf{UAct} \quad s \in \mathsf{SAct}$ (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) $fd ::= T f$
(method) $md ::= \mathsf{requires}\ F\ \mathsf{ensures}\ F\ T\ m(\overline{V i})\{c\}$
(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.) $hc ::= j = v; \mid j = op(\overline{v}); \mid j = v.f; \mid j = \mathsf{new}\ C\ \langle v \rangle; \mid j = v.m(\overline{v});$
$\mid v.f = v; \mid \mathsf{if}\ v\ \mathsf{then}\ c\ \mathsf{else}\ c; \mathsf{assert}\ F$
$\mid v.\mathsf{lock}(); \mid v.\mathsf{commit}(); \mid v.\mathsf{unlock}(); \mid v.\mathsf{fork}(); \mid v.\mathsf{join}();$
$\mid \mathsf{crhist}(L, R) \mid \mathsf{action}\ v.a(\overline{v})\{sc\} \mid \mathsf{reinit}(L, R) \mid \mathsf{dsthist}(L)$
$sc ::= j = v \mid j = v.f \mid j = \mathsf{new}\ C\ \langle v \rangle \mid v.f = v \mid T\ j; sc \mid T\ i = j; sc$
$\mid \mathsf{if}\ v\ \mathsf{then}\ sc'\ \mathsf{else}\ sc'' \mid sc'; sc'' \mid j = v.m(\overline{v})$

Fig. 1. Language syntax

actions $a(\overline{v}) \in \mathsf{UAct}$. The definition of classes, fields, methods etc. are standard. For simplicity, 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, containing 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) and processes ($proc$). Actions only have a specification, and no body. Processes have a specification and a body, which must be defined as a proper process expression.

Most of the formulas and commands in the language are standard. To reason about histories, we use the predicates Hist and APerm, and the specification commands for *creating* ($\mathsf{crhist}(L, R)$), *destroying* ($\mathsf{dsthist}(L)$), *reinitialising* a history ($\mathsf{reinit}(L, R)$), and starting an action ($\mathsf{action}\ v.a(\overline{v})\{sc\}$), where sc is a special subcategory of commands allowed within an action segment. Note that 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.

In Sec. 2 we already discussed the protocol for reasoning about locks. Therefore, the language includes the predicates $e.\mathsf{fresh}()$ and $e.\mathsf{initialized}()$, as well as the $v.\mathsf{commit}()$ command. Every object (except threads) 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.

4.2 Semantics of Histories

Histories are modelled as µCRL proces algebra terms. The set of actions is defined as: $\mathcal{A} = \mathsf{UAct} \cup \mathsf{SAct} \cup \{\tau, \delta\}$, while the communication function γ 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 a history term is defined in terms of its traces. In particular, we use the standard single step semantics $H \xrightarrow{a} H'$ for H moving in one step to H'. We extend this to:

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

Furthermore, we define the set of finished actions:

$$\mathsf{FAct} = \{a \in \mathsf{SAct} \mid \forall b \in \mathcal{A}.\gamma(a,b) = \bot\}$$

Now the *global completed trace semantics* of a process H is defined as:

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

4.3 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.

$$
\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 \perp]_{\forall l \in L}, hm[L \mapsto \perp]) \\
[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}
$$

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

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

– $hm \in \mathsf{HistMap} = \mathsf{Set}(\mathsf{Loc}) \rightharpoonup \overline{\mathsf{Action}}$ 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* 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 $(t_1, s_1, c_1)| \ ... \ |(t_n, s_n, c_n)$. 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 empty sequence, while $S +\!\!+ A$ appends the element A to a sequence S. The function defaultVal maps types to their default value, 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 InitHeap, and extends the domain of HistMap with the set L, while $\mathsf{dsthist}(L)$ is reversal: it removes the related entries from HistMap and 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 HistMap is made empty, and the values of $l \in L$ are copied from Heap to InitHeap. Note that there is no rule for the command $v.\mathsf{commit}()$; operationally this is a no-op.

Semantics of Expressions We write $[\![e]\!]^h_s$ to denote the semantics of an expression e, given a heap h and a stack s.

$$[\![v]\!]^h_s = v \text{ for } v : T, T \neq \mathsf{perm} \quad [\![1]\!]^h_s = 1 \quad [\![\mathsf{split}\,(\pi)]\!]^h_s = \frac{[\![\pi]\!]^h_s}{2}$$
$$[\![j]\!]^h_s = s(j) \quad [\![op(v_1, ..., v_n)]\!]^h_s = [\![op]\!]^h_s([\![v_1]\!]^h_s, ..., [\![v_n]\!]^h_s)$$

To express the semantics of a formula F (shown later in Fig. 4), we use the forcing relation $\Gamma \vdash \mathcal{R}; s \models F$ where: *i)* Γ is the *type environment* ($\Gamma = \mathsf{ObjId} \cup \mathsf{Var} \rightharpoonup \mathsf{Type}$), *ii)* $\mathcal{R}$ is a *resource*, and *iii)* s represents the *stack*. Before defining this forcing relation, we first explain what we call a resource.

Resources A resource $\mathcal{R}$ is an abstraction of the program state. Intuitively, we consider that each thread owns a resource, which contains *partial* information of the global state, describing the thread's local view of the program state (cf. [2]). Resources are defined as 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 abstractly describes part of the state: *i)* h represents the (*partial*) heap; *ii)* h_i represents the (*partial*) initial heap; *iii)* $\mathcal{P} \in \mathsf{Loc} \mapsto [0, 1]$ is a *permission table* that defines how much permission the resource 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 the resource 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)* $\mathcal{H}$: $\mathsf{Set}(\mathsf{Loc}) \mapsto \overline{\mathsf{Action} \times \mathsf{bool}}$ abstractly models the history map, by marking every action with a boolean flag to indicate whether it is owned by the resource; and *x)* $\mathcal{A} \subseteq \mathsf{Loc}$ stores those locations that are referred by an open action.

A resource $\mathcal{R}$ must satisfy the following *conditions*: *i)* the partial heap h contains only locations for which the resource holds a positive permission: $o \in \mathsf{dom}\,(h) \wedge f \in \mathsf{dom}\,(h(o)^2) \Leftrightarrow \mathcal{P}(o, f) > 0$; *ii)* the partial initial heap h_i contains only locations for which the resource holds a positive history fraction: $(o, f) \in \mathsf{dom}\,(h_i) \Leftrightarrow \mathcal{P}_h(o, f) > 0$ *iii)* the sets of fresh and initialised locks are disjoint: $\mathcal{F} \cap \mathcal{I} = \emptyset$; and *iv)* acquired locks are always initialised: $o \in \mathcal{L}(p) \Rightarrow o \in \mathcal{I}$.

Resources owned by different threads should be *compatible*, written $\mathcal{R}_1 \# \mathcal{R}_2$. For example, compatibility of $\mathcal{R}_1$ and $\mathcal{R}_2$ ensures that locations that exist in the partial heaps in $\mathcal{R}_1$ and $\mathcal{R}_2$ map to the same value, 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 $\mathcal{R}_2$.

Joining resources is defined by the *join* operation $\mathcal{R}_1 * \mathcal{R}_2$. Note that joining is only defined when the resources are compatible. For example, a joined resource contains the locations and permissions from both separate resources, and the actions collected from both history maps. Intuitively, if we only have a single thread, the resource should fully characterise the global program state.

Compatibility and joining of resources are formally defined *component-wise*, as shown in Fig. 3. Note that we use $x \vee \bot = \bot \vee x = x$, and $|x|$ to denote the length of the sequence x.

$$
\begin{array}{lll}
h\#h' & \Leftrightarrow & \forall o \in \mathsf{dom}(h) \cap \mathsf{dom}(h').h(o)^1 = h'(o)^1 \wedge \\
 & & \forall f \in \mathsf{dom}(h(o)^2) \cap \mathsf{dom}(h'(o)^2). \\
 & & h(o)^2(f) = h'(o)^2(f) \\
h_i\#h'_i & \Leftrightarrow & \forall (o,f) \in \mathsf{dom}(h_i) \cap \mathsf{dom}(h'_i).h_i(o,f) = h_i(o,f) \\
\mathcal{P}\#\mathcal{P}' & \Leftrightarrow & \forall (o,f).\mathcal{P}(o.f) + \mathcal{P}'(o.f) \leq 1 \\
\mathcal{P}_h\#\mathcal{P}'_h & \Leftrightarrow & \forall (o,f).\mathcal{P}_h(o.f) + \mathcal{P}'_h(o.f) \leq 1 \\
\mathcal{J}\#\mathcal{J}' & \Leftrightarrow & \mathcal{J} = \mathcal{J}' \\
\mathcal{L}\#\mathcal{L}' & \Leftrightarrow & \mathsf{dom}(\mathcal{L}) \cap \mathsf{dom}(\mathcal{L}') = \emptyset \wedge \\
 & & \forall o \in \mathsf{dom}(\mathcal{L}), \forall p \in \mathsf{dom}(\mathcal{L}').\mathcal{L}(o) \cap \mathcal{L}(p) = \emptyset \\
\mathcal{F}\#\mathcal{F}' & \Leftrightarrow & \mathcal{F} \cap \mathcal{F}' = \emptyset \\
\mathcal{I}\#\mathcal{I}' & \Leftrightarrow & \mathcal{I} = \mathcal{I}' \\
\mathcal{H}\#\mathcal{H}' & \Leftrightarrow & \mathsf{dom}(\mathcal{H}) = \mathsf{dom}(\mathcal{H}') \wedge \\
 & & \forall L \in \mathsf{dom}(\mathcal{H}).|\mathcal{H}(L)| = |\mathcal{H}'(L)| \wedge \\
 & & \forall i.\mathcal{H}(L)^1_i = \mathcal{H}'(L)^1_i \wedge \neg(\mathcal{H}(L)^2_i \wedge \mathcal{H}'(L)^2_i) \\
\mathcal{A}\#\mathcal{A}' & \Leftrightarrow & \mathsf{true}
\end{array}
$$

$$
\begin{array}{lll}
h*h' & \triangleq & \lambda o.(h(o)^1 \vee h(o)^1, h(o)^2 \vee h(o)^2) \\
h_i*h'_i & \triangleq & \lambda(o,f).h_i(o,f) \vee h'_i(o,f) \\
\mathcal{P}*\mathcal{P}' & \triangleq & \lambda(o,f).\mathcal{P}(o.f) + \mathcal{P}'(o.f) \\
\mathcal{P}_h*\mathcal{P}'_h & \triangleq & \lambda(o,f).\mathcal{P}_h(o.f) + \mathcal{P}'_h(o.f) \\
\mathcal{J}*\mathcal{J}' & \triangleq & \mathcal{J} \\
\mathcal{L}*\mathcal{L}' & \triangleq & \mathcal{L} \cup \mathcal{L}' \\
\mathcal{F}*\mathcal{F}' & \triangleq & \mathcal{F} \cup \mathcal{F}' \\
\mathcal{I}*\mathcal{I}' & \triangleq & \mathcal{I} \\
\mathcal{H}*\mathcal{H}' & \triangleq & \lambda L.\lambda i.(\mathcal{H}(L)^1_i, \mathcal{H}(L)^2_i \vee \mathcal{H}'(L)^2_i) \\
\mathcal{A}*\mathcal{A}' & \triangleq & \mathcal{A}
\end{array}
$$

Fig. 3. The compatibility (#) and the join (∗) operator

$$
\mathcal{R} = (h, h_i, \mathcal{P}, \mathcal{P}_h, \mathcal{J}, \mathcal{L}, \mathcal{F}, \mathcal{I}, \mathcal{H}, \mathcal{A})
$$

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

Fig. 4. Semantics of Formulas

Semantics of Formulas Finally, Fig. 4 presents the semantics of formulas in our language. The predicate $\mathsf{Hist}(L, \pi, R, H)$ is valid when the resource $\mathcal{R}$ contains at least a fraction π of the history associated to every $l \in L$; the formula 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 $e.f$.

4.4 Proof Rules

Fig. 5 presents the proof rules for our theory. We leave out a few standard Hoare triples (the whole list can be found in [2]). We use $\circledast_i F_i$ to abbreviate a

$$[New]\frac{\Gamma \vdash v : \mathsf{pred} \quad j : C\,\langle v\rangle}{\Gamma \vdash \{\mathsf{true}\}\ j = \mathsf{new}\ C\,\langle v\rangle\ \{\circledast_{T\,f\in\mathsf{fld}(C)}\mathsf{PointsTo}(j.f, 1, \mathsf{defaultVal}(T))\}}$$

$$[Fork]\frac{\Gamma \vdash v : CT \quad mtype(run, CT) = \mathsf{requires}\ F\ \mathsf{ensures}\ F'\mathsf{void}\ run(V_0\ i_0)\{c\}}{\Gamma \vdash \{F[v/i_0]\}v.\mathsf{fork}()\{\mathsf{Join}(v)\}}$$

$$[Join]\frac{\Gamma \vdash v : CT \quad mtype(run, CT) = \mathsf{requires}\ F\ \mathsf{ensures}\ F'\mathsf{void}\ run(V_0\ i_0)\{c\}}{\Gamma \vdash \mathsf{Join}(v)\ v.\mathsf{join}()\ \{F'[v/i_0]\}}$$

$$[Read]\frac{\Gamma \vdash v, \pi, w : V, \mathsf{perm}, W \quad W\ f \in \mathsf{fld}(V)}{\Gamma \vdash \{\mathsf{PointsTo}(v.f, \pi, w)\}\ j = v.f\ \{\mathsf{PointsTo}(v.f, \pi, w) * j == w\}}$$

$$[Write]\frac{\Gamma \vdash v, w : V, W \quad W\ f \in \mathsf{fld}(V)}{\Gamma \vdash \{\mathsf{PointsTo}(v.f, 1, -)\}\ v.f = w\ \{\mathsf{PointsTo}(v.f, 1, w)\}}$$

$$[ReadH]\frac{\Gamma \vdash v, \pi, w : V, \mathsf{perm}, W \quad W\ f \in \mathsf{fld}(V)}{\Gamma \vdash \{\mathsf{APerm}(v.f, \pi, w)\}\ j = v.f\ \{\mathsf{APerm}(v.f, \pi, w) * j == w\}}$$

$$[WriteH]\frac{\Gamma \vdash v, w : V, W \quad W\ f \in \mathsf{fld}(V)}{\Gamma \vdash \{\mathsf{APerm}(v.f, 1, -)\}\ v.f = w\ \{\mathsf{APerm}(v.f, 1, w)\}}$$

$$[Create]\frac{\forall v.f \in L \quad \Gamma \vdash v, f, w : V, W, W; f \in \mathsf{fld}(V);}{\Gamma \vdash \{\circledast_{\forall v.f\in L}\mathsf{PointsTo}(v.f, 1, w) * R\}\mathsf{crhist}(L, R)\{\circledast_{\forall v.f\in L}\mathsf{Perm}(v.f, 1, w) * \mathsf{Hist}(L, 1, R, \epsilon)\}}$$

$$[Destr]\frac{\forall v.f \in L \quad \Gamma \vdash v, f, w : V, W, W; f \in \mathsf{fld}(V);}{\Gamma \vdash \{\circledast_{\forall v.f\in L}\mathsf{Perm}(v.f, 1, w) * \mathsf{Hist}(L, 1, R, \epsilon)\}\mathsf{dsthist}(L)\{\circledast_{\forall v.f\in L}\mathsf{PointsTo}(v.f, 1, w) * R[w/v.f]_{\forall v.f\in L}\}}$$

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

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

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

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

Fig. 5. Selected set of proof rules

separation conjunction over all formulas F_i. Rules $[ReadH]$ and $[WriteH]$ state that accessing a location is allowed if an action is in progress (APerm predicates are required), while $[Read]$ and $[Write]$ can only be used when there is no history maintained for the accessed location (as they require the PointsTo predicate). The $[Action]$ rule describes that if the action implementation satisfies the action's contract, the action will be recorded in the history.

The premise in the $[Reinit]$ rule requires that the Hoare triple $\{R\}w\{R'\}$ holds for every trace $w \in \mathsf{Traces}(H)$. Importantly, w is a trace of actions, where every action can also be considered as a call to an abstract method (an action contains a specification and no implementation); thus, the trace w is also a sequential program statement.

[*SplitMergeHist*] and [*Sync*] are not proof rules about a program statement, instead they define how history predicates can be exchanged for each other.

4.5 Soundness

The soundness of our verification system is ensured by the following theorem:

Theorem 1. *Let c be a verified program with an initial state σ_0 and $\sigma_0 \rightsquigarrow^\star \sigma$ where $\sigma = (h, tp|(t, s, \mathsf{assert}\ F; c'), lt, h_i, hm)$, then there is a resource $\mathcal{R}$ that abstracts the state σ and $\mathcal{R}, s \models F$.*

Proof. The soundness result follows by induction over the commands in the language. We sketch only the proof of the [*Reinit*] rule. The proofs of the other rules basically follow directly from the semantics of the formulas and the operational semantics.

Proof sketch of the [*Reinit*] *rule*

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

Let σ and σ' be the pre- and poststate of the $\mathsf{reinit}(L, R')$ command, respectively, σ_{init} is the last initial state of the history and σ_H and σ'_H, are the prestate state of the first action and the poststate of the last action from the history. Thus, $\sigma_{\mathsf{init}} \leq \sigma_H \leq \sigma'_H \leq \sigma < \sigma'$, where "$\leq$" denotes "precedes or is equal to" (equality holds if the history is empty).

From the semantics of the Hist predicate (see Fig. 4), we need to prove that R' holds on the $\mathsf{InitHeap}$ h'_i in the state σ' (the other requirements are trivial to prove). From the precondition $\Gamma \vdash \mathcal{R} \models \mathsf{Hist}(L, 1, R, H)$ we know that R holds on the $\mathsf{InitHeap}$ h_i in the state σ, i.e., $R[h_i(l)/l]_{\forall l \in L} = \mathsf{true}$. This implies that R holds on the Heap in the state σ_{init}, when the values from the Heap have been copied to the $\mathsf{InitHeap}$. Furthermore, no update of $l \in L$ might have happened between σ_{init} and σ_H (any update must be preceded by starting an action). Therefore, all values of the locations in L from σ_{init} and σ_H are equal. We denote this $\sigma_{\mathsf{init}} =_L \sigma_H$. Thus, R holds on the Heap in σ_H.

Additionally, a full predicate $\mathsf{Hist}(L, 1, R, H)$ means that the resource $\mathcal{R}$ contains the whole global history gh over L, $gh = hm(L) = \mathcal{R}_{\mathsf{hist}}(L)$ and thus, $gh \in \mathsf{Traces}(H)$. The premise of the [*Reinit*] rule states that $\{R\}w\{R'\}$ holds for every $w \in \mathsf{Traces}(H)$. From $gh \in \mathsf{Traces}(H)$, we have $\{R\}gh\{R'\}$. This means that if R holds in a state σ_H (which we proved above), we can conclude that R' holds in a state σ'_H (this is because the program execution results in a state equivalent to the result state of an execution in which the actions happen serially, without overlapping). Moreover, $\sigma'_H =_L \sigma$ because no update of $l \in L$ might have happened between σ'_H and σ. Thus, R' holds on the Heap in σ.

Finally, the operational semantics defines that the $\mathsf{reinit}(L, R')$ command changes the $\mathsf{InitHeap}$ h_i to a heap h'_i, such that values of the locations in L are copied from the Heap to h'_i: $\forall l \in L\ h'_i(l) = h(l)$. This implies that R' holds on the $\mathsf{InitHeap}$ h'_i in the state σ', which concludes our proof.

□

5 Tool Support

We have integrated our history-based technique in the program verifier, the VerCors tool set [4]. The tool performs verification of programs written in languages such as Java and C annotated with specifications in variants of separation logic. The tool encodes the specified program into a much simpler language and then applies the Chalice [20] and Silver [17] verifiers to the simplified program.

To verify programs specified with histories, there are two verification tasks to be performed. In top down order, we have to check i) if the [*Reinit*] rule (see Sec. 4.4) is applied correctly, i.e., for every $w \in \mathsf{Traces}(H)$, the Hoare triple $\{R\}w\{R'\}$ logically follows from the contracts on the actions, and ii) if the local histories are properly maintained in the program.

Verification of the [*Reinit*] rule To verify the functional behaviour of processes, the tool requires that every action or process definition is specified with a contract. Each *action definition* is then translated to an abstract method (without implementation) with a corresponding specification. For processes there are two steps to be done: *process transformation* and *method generation*.

Process transformation Every process is first transformed to a *guarded sequential* process (it should contain no merge ($\|$) operator). This rewriting is done by applying techniques known from *linearisation* of processes (see e.g. [10]). First, the definition is expanded by applying the axioms of process algebra and unfolding defined processes until the result is a guarded process. Then, all parallel compositions are replaced by defined processes. To perform the latter step, the user has to specify all parallel compositions that might occur.

As an example, we consider a process $par(n, m) = p(n) \parallel p(m)$, where $p(n)$ is the process defined in Lst. 4, line 30. Thus, the expression describes a program where two threads are running in parallel, each of them repeatedly increasing a shared location *data*, respectively n and m times. For the tool to reason about the behaviour of this process, it will automatically perform partial linearisation of the process, i.e., derive a new process $par'(n, m)$ from $par(n, m)$ that is sequential:

$$\begin{aligned}
\mathrm{par}'(n,m) &= \mathrm{p}(n) \parallel \mathrm{p}(m) = \cdots \\
&= (\mathrm{inc}(1).(\mathrm{p}(n-1) \parallel \mathrm{p}(m))) \triangleleft n > 0 \triangleright \mathrm{p}(m) + \\
&\quad (\mathrm{inc}(1).(\mathrm{par}(m-1) \parallel \mathrm{p}(n))) \triangleleft m > 0 \triangleright \mathrm{p}(n) \\
&= (\mathrm{inc}(1).\mathrm{par}(n-1,m)) \triangleleft n > 0 \triangleright \mathrm{p}(m) + \\
&\quad (\mathrm{inc}(1).\mathrm{par}(m-1,n)) \triangleleft m > 0 \triangleright \mathrm{p}(n)
\end{aligned}$$

Processes par′ and par are equivalent and thus, verifying that the derived process par′ satisfies its contract proves that par satisfies its contract too.

For history processes that are very complex, it is possible to define a second process, prove that the processes are equivalent and show that the simple process satisfies its contract. This simplifies verification because the simple process is easier to specify and verify and the equivalence proof can be carried out by

external tools without considering functional specification of processes. For example, we can use the `lpsbisim2pbes` from the mCRL2 toolset [11]. However, this is still not integrated in the tool.

Method Generation As a second step, the transformed process is translated to a method to verify that the ensured data modifications follow logically from those specified for the actions. This translation is straightforward: all process algebra operators of sequential processes are also control flow operators in Java, except the *alternative composition* (the "+" operator). Thus, we encode this operator with an *if statement* with a randomly assigned boolean value as a condition.

For example, to verify that the process par′ (which is guarded and sequential) satisfies its contract, we check the following generated code (where if(∗) stands for non-deterministic choice and *empty*() is a predefined *empty process* (ϵ)):

```
//@ requires n >= 0 && m >= 0;
//@ ensures x == \old( x ) + n + m;
void par'(int n,int m){
  if ( * ) {
    if (n > 0) { inc(1); par'(n  1,m); } else { p(m); }
  } else {
    if (m > 0) { inc(1); par'(m  1,n); } else { p(n); }
  }
}
```

Verification of Local History Maintenance To verify compliance with histories, the proof obligations are encoded as program specifications in plain separation logic. To achieve this, for each *action implementation*, it is verified that the statements in the action segment satisfy the requirements of the action. Furthermore, the encoding uses two dedicated data types. First, a class *History* is used with a constructor that encodes the rule for creating a history, and methods that encode the other history-related rules (splitting, merging, reinitialisation or destroying a history). Second, a data type is used to replace process expressions that are not a native data type of the back end. This type is used in the specifications of the methods of the history that correspond to the history annotations.

To verify that an action is recorded properly, at the beginning of the action segment, the values of the footprint locations of the action are stored in local variables. At the end of the action segment, an assertion is set to check the validity of the postcondition of the action, in which the *old* values are replaced with the stored local variables. In addition, another assertion checks the precondition of the action, i.e., the requirements of the arguments of the action.

6 Conclusions and Related Work

This paper introduced a new history-based technique for modular verification of functional behaviour of concurrent programs. This technique allows one to provide intuitive method specifications that describe only the *local effect* of a thread, in terms of abstract (user-specified) *actions*, which reduces the need to reason about fine-grained thread interleavings. The technique is an extension of permission-based separation logic. It is particularly suited to reason about programs with internal synchronisation, and notably, when access to certain locations is protected by multiple locks. Support for the approach is added to the VerCors tool set [4], and experimentally on top of the VeriFast logic [26].

Related Work The problem of non-modularity of the Owicki-Gries approach [24] has been investigated before by Jacobs and Piessens [15]. Based on the Owicki-Gries technique, they propose a logic that allows to augment the client program with auxiliary update code (as a higher-order parameter) that is passed as an argument to methods. For example, for the *incr* method discussed in the introduction, the user has to add ghost code $a := 1$ or $b := 1$, respectively to both method calls. This results in a kind of a higher-order programming that allows reasoning about fine-grained data structures. This logic is expressive enough to support various examples; however, it requires the user to provide the concrete updates to the local state in an explicit way at *each* method call, which imposes a large overhead on verification. Moreover, the user needs to specify a concrete invariant property (as in Lst. 2) that remains stable under the updates of all threads. The choice of such an invariant is usually not trivial, especially when the access to locations requires acquiring multiple locks.

Another similar approach to reason about the functional behaviour of concurrent programs is by using *Concurrent Abstract Predicates (CAP)* [8], 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 CAP) [27], where a CAP may be parameterised by a protocol defined by the client.

Compared to these approaches, histories are in a way a ghost code that keeps track of the local contributions. We use process algebra to combine the local histories: this allows avoiding the need to specify the behaviour of the threads in an invariant. We do use invariants related to every lock, but by using histories, we intend to use these invariants for storing permissions only. Therefore, we believe histories allow more natural specifications.

Strongly related to our work is the recently proposed prototype logic of Ley-Wild and Nanevski [21], the *Subjective Concurrenct Separation Logic (SCSL)*. They allow modular reasoning about coarse-grained concurrent programs by verifying the thread's local contribution with respect to its *local view*. When views are combined, the local contributions are combined. To this end, the logic

contains the *subjective separating conjunction* operator, $\circledast$, which splits (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 and where $\mathbb{0}$ describes no effect. To solve the Owicki-Gries example, a PCM $(\mathbb{N}, +, 0)$ is chosen: threads 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 right PCM will become troublesome.

In contrast to their technique, our histories are stored as parallel processes of actions that are resolved later. 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 example described above). Moreover, our method is also suited to reason about programs with dynamic thread creation.

Furthermore, also closely related to our approach is the work on *linearisability* [28, 29]. 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. 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. [9] 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 [13]. It might be worth investigating if these ideas carry over to our approach, by adding different synchronisation actions to the histories.

Recently, some very promising parameterisable logics have been introduced [7, 18] to reason about multithreaded programs. The concepts that they introduce are very close to our proof logic. Reusing such a framework will be useful to simplify the formalisation and justify soundness of our system, as well as to show that the concept of histories is more general and applicable in other variations of separation logic. However, in their current form, they can be used as a foundation only for simplified versions of our logic. In particular, to the best of our knowledge, they are not directly applicable to our language as it contains dynamic thread creation instead of the parallel $\|$ operator.

Future Work As future work, we plan to investigate if process algebra simplifications can be applied during the construction of the history, without comprising soundness. On the longer term, we also want to analyse how this history-based approach can be used to reason about distributed software. This will require

more variations in how the global history can be derived from the local histories, but we expect that apart from this, most of the approach directly carries over.

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. A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for multithreaded java programs. *CoRR*, abs/1411.0851, 2014.
3. J. C. Baeten and W. P. Weijland. Process algebra, volume 18 of Cambridge tracts in theoretical computer science, 1990.
4. 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.
5. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.
6. C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
7. 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.
8. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.
9. T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.
10. J. Groote, A. Ponse, and Y. Usenko. Linearization in parallel pcrl. *The Journal of Logic and Algebraic Programming*, 48(12):39 – 70, 2001.
11. J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. Analysis of distributed systems with mCRL2. *Process Algebra for Parallel and Distributed Processing*, 2009.
12. J. F. Groote and M. A. Reniers. Algebraic process verification. In *Handbook of Process Algebra, chapter 17*, pages 1151–1208. Elsevier.
13. N. Hemed and N. Rinetzky. Brief announcement: Contention-aware linearizability. In *PODC 2014*, 2014.
14. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
15. B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.
16. 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.
17. U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.
18. 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.
19. 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.
20. K. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.

21. R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574, 2013.
22. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comp. Sci.*, 375(1-3):271–307, 2007.
23. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.
24. S. S. Owicki and D. Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.
25. J. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on LICS*, pages 55–74. IEEE Computer Society, 2002.
26. J. Smans, B. Jacobs, and F. Piessens. VeriFast for Java: A tutorial. In *Aliasing in Object-Oriented Programming*, pages 407–442. Springer, 2013.
27. K. Svendsen and L. Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.
28. V. Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.
29. V. Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464, 2010.