Verifying Functional Behaviour of Concurrent Programs

Marina Zaharieva-Stojanovski
University of Twente
the Netherlands

Marieke Huisman
University of Twente
the Netherlands

Stefan Blom
University of Twente
the Netherlands

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

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

1. INTRODUCTION
Verifying program correctness means proving that the program behaves as described by its formal specification. In a concurrent program, an inconsistent behaviour may occur due to thread interleavings and potential data-race conditions. Existing techniques for verifying concurrent software often focus on proving data-race freedom in a program [4, 11, 3]. Although this is an essential property for a concurrent program, it does not guarantee that the program behaves as the programmer expects. In practice, specifying the expected behaviour of a concurrent program is often quite challenging using the existing verification techniques.

We illustrate this by an example: Lst. 1 shows a class Counter, representing a simple shared data structure. The increase method is implemented correctly and is data-race free, as the shared location data is protected by a lock. However, because synchronisation happens inside the method (internal synchronisation), it is difficult to describe the behaviour via the method contract. While the postcondition expression $data = \text{old}(data) + 1$ would perfectly express the intended behaviour of the method in a sequential program, the same specification is not acceptable in a concurrent setting where, because of the unknown number of active parallel threads, the value of data is unstable after the lock release. As a result, method contracts in scenarios like this often do not fully express the method behaviour, which also limits proving properties for the client class that uses the data structure. If the Client in Lst. 1 creates a Counter object c with initial value $c.data = 0$ (line 14), and then forks two parallel threads, each of them increasing $c.data$ by 1 (line 15), we cannot prove in a modular way that after joining both threads, the value of $c.data$ is equal to 2.

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

Lst. 1: A shared Counter data structure

In this paper we develop a new method for reasoning about partial correctness of behavioral properties in concurrent programs. Our logic is an extension of permission-based separation logic [3], while the specification language is based on JML (Java Modeling Language) [12]. We target programs with internal synchronisation, as the example in Lst. 1.

The general idea of the approach is the following. We introduce actions as part of the specification language: an action over a heap location $x$ describes a (typically non-atomic) change of the value at $x$. For example, the action for incrementing an integer value by 1 may be specified as:

$$\text{action } a[x] \equiv \text{old}(x) + 1$$

When specifying the precise value of a location $x$ in the method post-state is difficult (as in Lst.1), the programmer
may specify the behaviour of the method in terms of actions over \( x \) executed within the method. Every action over \( x \) is recorded in a history of changes \( H \) associated to \( x \). In particular, every heap location \( x \) is associated with a predicate \( \text{Hist}(x, \pi, H) \), where \( H \) is a history (modelled as a process algebra term [7]) in which all actions over \( x \) are recorded.

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

The \text{Counter.increase()} method may be specified as:

```java
//@ requires Hist(data, \pi, H);
//@ ensures Hist(data, \pi, H \cdot a),
```

where \( a \) is the action specified above. The method contract describes only the local changes in the history: the actual thread has increased the value of \( \text{data} \) by 1.

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

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

2. ALGEBRA OF COMMUNICATING PROCESSES

The algebra of communicating processes (ACP) [7] is a mathematical approach for reasoning about system behaviour in terms of algebraic process expressions. The basic primitives in ACP are actions from the set \( A = \{a, b, c, \ldots\} \), each of them representing an indivisible process behaviour. To describe various processes \( \{p_1, p_2, \ldots\} \), actions are combined using algebraic operators, the most fundamental of which are the sequencing composition (\( \cdot \)) and the alternative composition (\( + \)). For example, the expression \( a + (b \cdot c) \) expresses a process composed of an action \( a \) or a sequence of actions \( b \) and \( c \). Further, two special actions are used: the deadlock action \( \delta \) and the silent action \( \tau \) (an action without behaviour). We have: \( \delta \cdot p = \delta \), \( \delta + p = p \) and \( \tau \cdot p = p \).

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

3. PERMISSIONS, FRAMING, STABILITY

Separation Logic and Permissions. Permission-based separation logic [14, 13] is a program logic (an extension of Hoare Logic [9]) used to reason about multitreaded programs. Every access to a heap location is associated with a fractional permission \( \pi \), i.e., a value in the domain \( (0,1] \) [3]. At any point in time, a thread might hold a permission to access a location. To change a location \( x \), a thread must hold a write permission for \( x \), i.e., \( \pi = 1 \); while for reading a location, any read permission is required, i.e., \( \pi > 0 \). The soundness of this logic ensures that the sum of all threads’ permissions for a certain location never exceeds 1, which guarantees that a verified program is data-race free.

The basis of this logic is the binary separating conjunction operation (\( * \)): \( P * Q \) holds when \( P \) and \( Q \) describe disjoint resources and thus, may be used by two parallel threads. Permission for a location \( x \) is expressed via the predicate \( \text{PointsTo}(x, \pi, v) \), which indicates that \( x \) points to a location for which the thread has a permission \( \pi \) and the value of \( x \) is \( v \). Proof rules for writing and reading are described by the following Hoare triples (where “ \( ?u \)” means \( \text{any value} \) and we name this value “\( u \)”):

- \text{[Write]} \quad \{ \text{PointsTo}(x, 1, ?u) \} \quad x = v; \quad \{ \text{PointsTo}(x, 1, v) \}
- \text{[Read]} \quad \{ \text{PointsTo}(x, \pi, v) \} \quad l = x; \quad \{ \text{PointsTo}(x, \pi, v) * l == v \}

The \text{PointsTo} predicate may be used as a token, i.e., it can be split and merged, and parts of the token may be distributed among parallel threads. This is shown by the \[\text{SplitPerm}\] rule, where the operator \( * * \) means “splitting” (read from left to right) or “merging” (read from right to left):

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

Framing and Stability. Permission-based separation logic is based on the concept of framing: every shared location \( x \) in a formula must be framed, i.e., the formula must express a positive permission \( \pi \) to \( x \). Holding a permission guarantees that the value of \( x \) is stable and can not be changed by any other thread. Framing is implicitly maintained with the \text{PointsTo} predicate: in general, we can reason about the value of \( x \) only via the \text{PointsTo}(x, \pi, v) predicate. This predicate in a way binds together the knowledge of the value \( v \) at a location \( x \) with an access permission to \( x \).
class Counter {
    // @ pred res_inv = PointsTo(data, 1, ?v);
    lock = new Lock/*@<res_inv>@*/;
    // @ requires //lock_not_held;
    // @ ensures //lock_not_held;
    void increase() {
        /*@ (PointsTo(data, 1, ?v)) */
        data += 1;
        lock.unlock();
        /*@ (true) @*/
    }
}

Lst. 2: The Counter class - specification with locks

Using Locks. Haack et al. [8] show how to use permission-based separation logic to reason about programs with re-entrant locks. For each lock, a special predicate is defined, called a resource invariant, describing which permissions the lock protects. For example, the resource invariant res_inv in the class Counter is associated to the lock object, expressing that a write permission to data is protected by the lock, see Lst. 2, lines 3, 4. When a thread acquires the lock, it gets the associated resource invariant (except for reentrant acquiring) (line 10). Upon final lock release, the thread returns the resource invariant back to the lock (line 14).

4. APPROACH

The specification of the Counter class (see Lst. 2) is strong enough to verify data-race freedom; however, it does not state anything about the behaviour of the increase method. Although we can not reason about the value of data in the method poststate, we would like the postcondition to express that the method functions correctly, i.e., at a certain point in the past, the value of data respected a specific property. This rises the question: How can we reason about the value of x in the past, without holding any permission to x now?

4.1 Separation of Value and Permission

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

The key of our concept is to separate i) the knowledge of the value for a given location and ii) the access permission for this location. As these two properties are tied together by the PointsTo predicate, we extend the semantics of this predicate, by adding the [Separate] rule:

\[
\text{PointsTo}(x, 1, ?v) * v \in V \leftrightarrow \text{Perm}(x, 1) * \text{Init}(x, V) * \text{Hist}(x, 1, \emptyset)\]

The [Separate] rule splits the PointsTo predicate in two separate parts: i) Perm(x, π) predicate, which keeps the access permission π for the location x and ii) Init and Hist predicates, which store information about the value of x.

The Init(x, V) predicate states that at a given moment T in the past (or possibly now), x had a value from the set V. Normally, splitting is done on the predicate PointsTo(x, 1, v); then the Init predicate stores the current unique value of x:

\[
\text{PointsTo}(x, 1, v) \leftrightarrow \text{Perm}(x, 1) * \text{Init}(x, \{v\}) * \text{Hist}(x, 1, \emptyset)\]

When x is changed, the Init(x, V) predicate is not directly updated; instead, the change is recorded in the Hist(x, π, H) predicate. In particular, the Hist(x, 1, H) predicate contains a history H of all changes of x after the moment T. The history H is modelled as an ACP process algorithm term [7], where every action is a change of x (we discuss actions more precisely later in Sec. 4.2). At the moment of splitting, there are still no changes registered in the history, and thus the Hist predicate contains an empty history, H = ∅.

The second parameter π in the Hist predicate is used to make it a splittable token. Thus, the following rule holds:

\[
[S\text{plitHist}]
\text{Hist}(x, \pi, H) * \rightarrow \text{Hist}(x, \pi, H_1) * \text{Hist}(x, \pi, H_2),
\]

where || is the standard ACP parallel composition operator. Later, in Sec. 4.2 we explain how H1 and H2 can be chosen when splitting the Hist token (when forking a new thread). When the Hist token is distributed among several parallel threads, every thread is responsible to record its own changes to x in its own part of the token. At the end, when all threads are joined and the full token is again obtained, all thread local histories are merged together (H = H1 || H2).

To reason about the current value at location x, both Hist and Hist predicates are required. Moreover, Hist must be a full token, i.e., Hist(x, 1, H). The Init(x, V) predicate stores the initial value(s) of x at a given moment T, while H (as a non-deterministic process) contains all changes done after T. Based on this information, the value V may be updated to a set of new possible values of x and the history H will be reinitialised to H = ∅ (we discuss this further in Sec. 4.5).

4.2 A History as a Communication Process

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

\[
\text{action act}_{label} \{ \text{Type } x \} \equiv f([l], \text{old}(x))\]

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

The right hand-side of the action definition is the interpretation of the action, we denote \(rs(a [x] (l)) = f(l, \text{old}(x))\). Every action over x is interpreted as a function over the list of parameters l and the value \(\text{old}(x)\), i.e., the value of x at the moment before the action starts. The function returns the value of x after the action is finished. An action is not
necessarily atomic, it is typically a sequence of operations wrapped in an abstract change.

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

Below, we show examples of three actions. The action $a$ represents a change of an $int$ value, where the value is increased by $k$; action $b$ describes adding an element to a list; while action $c$ represents an assignment to a specific value $w$.

\[
\begin{align*}
\text{action } a \{ \text{int } x \} (\text{int } k) & \equiv \{ \text{old}(x) + k \\
\text{action } b \{ \text{list } l \} (\text{int } elem) & \equiv \text{cons(elem, old(l))} \\
\text{action } c \{ \text{int } x \} (\text{int } w) & \equiv w
\end{align*}
\]

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

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

\[
\begin{align*}
\gamma(s, \bar{s}) & = \tau \\
\gamma(a, b) & = \delta \text{ if } a \not\in A_s \lor (a \in A_s \land b \not= \bar{a}) \\
\gamma(s, x) & = \delta, s \in A_s
\end{align*}
\]

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

\[
\text{Hist}(x, \pi, H) \rightarrow \text{Hist}(x, \pi/2, H \cdot s) \cdot \text{Hist}(x, \pi/2, \bar{s}), s \in L_s.
\]

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

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

### 4.3 Program Specifications

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

Having the Hist predicate that expresses changes in the past, we can easily specify the behaviour of both methods. In their prestate it is required that the actual thread holds (part of) the Hist token associated to `data` (lines 8, 19), while the postconditions guarantee that the proper change over `data` is recorded in the history $H$ (lines 9, 20). Thus, no permission is needed in the pre- or poststate of the methods. In fact, the permission to `data` is obtained inside the method via the lock object: however, the information about the value of `data` is now detached from the lock, and can be used independently.

For soundness of the approach, it is required that the program segment where a certain action occurs is explicitly specified in the program. Therefore, we introduce two specification commands: i) `start(a[x]/(l))` indicates the beginning of the action and ii) `commit(a[x]/(l))` indicates the end of the action after which the action must be recorded in the history (see Lst. 3, lines 12, 16 and 23, 25). Note that two program segments that represent an action over a same location must not overlap: this is important in order to avoid recording the same update several times in the history.

### 4.4 Verification Methodology

To check whether the program meets the specification, the verifier must: i) ensure that the `start` and `commit` specification commands are properly added when required; ii) ensure that the actions added to the history have indeed happened.
Ensuring start and commit existence. When updating the value of a certain location \( x \), we want to ensure that the change is registered somewhere. When using the PointsTo predicate, the newly assigned value is directly recorded into the predicate itself, see Hoare triple [Write], Sec. 3. With our approach, the PointsTo predicate is split into the predicates Perm, Hist and Init. Thus, in addition to the triple [Write], we need to introduce another rule for writing that should be used when the PointsTo predicate is split. In particular, we have to ensure that the assignment to \( x \) happens indeed as part of an action over \( x \) that later will be added to the history of changes of \( x \).

For this purpose, we define that when an action over \( x \) starts, a token HistPerm associated to \( x \) is produced. This token is in a way a permission obtained from the history that allows writing at location \( x \), with a guarantee that the changes will be recorded later. The start command consumes the Hist\((x, \pi, H)\) token and returns it back when the action is finished. This is described by the following Hoare triple:

\[
\begin{align*}
\text{[Start]} & \\
\{\text{Hist}(x, \pi, H)\} & \text{start a}[x]\{\tilde{0}\}; \\
\{\text{HistPerm}(x, \pi, H)\} & 
\end{align*}
\]

The new Hoare triple for assigning a location \( x \) (in addition to the triple [Write]) is defined as:

\[
\begin{align*}
\text{[WriteHist]} & \\
\{\text{Perm}(x, 1) \cdot \text{HistPerm}(x, \pi, H)\} & x = w; \\
\{\text{Perm}(x, 1) \cdot \text{HistPerm}(x, \pi, H)\} & x = w
\end{align*}
\]

The [WriteHist] rule requires writing permission for the location \( x \) (Perm\((x, 1)\)) in the prestate, as well as permission from the history (HistPerm\((x\)).

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

\[
\begin{align*}
\text{[Commit]} & \\
\{\text{HistPerm}(x, \pi, H) \cdot x = rs(a[x](\tilde{l}))\} & \\
\text{commit a}[x]\{\tilde{0}\}; \\
\{\text{Hist}(x, \pi, H \cdot a(\tilde{l}))\} & 
\end{align*}
\]

With the execution of the commit command, the action is recorded in the history under the condition that the value of \( x \) is properly changed as described by the action interpretation \((x = rs(a[x](\tilde{l}))\)). Lst. 4 shows the proof outline for the increase method.

4.5 Reasoning using a History

As discussed above, to reason about the value at a location \( x \) we need both the Init\((x, V)\) predicate and the full Hist\((x, 1, H)\) token. A full Hist token ensures that \( x \) is in a stable state and no thread can modify \( x \)’s value. The set of possible values for \( x \) can be calculated after interpreting all actions from the history. This is stated by the rule:

\[
\begin{align*}
\text{Hist}(x, 1, H) & \cdot \text{Init}(x, V) \rightarrow \text{Hist}(x, 1, \emptyset) & \cdot \text{Init}(x, [\{H_2^x\}]^V) & \\
\text{where} & \quad [\{H_2^x\}]^V & \text{returns a set of possible values for} & \text{for the evaluation of the process} & \text{of actions over} & \text{x, where the} & \text{initial value of} & \text{x was any} & \text{v \in V}. \\
\text{We define the} & \quad [\{H_2^x\}]^V & \text{operation inductively as follows (note} & \text{that the} & \text{operator can be reduced to} & \cdot \text{and +):} & \\
\text{i) & \quad [\emptyset]^V & = V & \\
\text{ii) & \quad [H_1^x + H_2^x]^V & = [H_1^x]^V \cup [H_2^x]^V & \\
\text{iii) & \quad [a(x) \cdot H_2^x]^V & = \bigcup_{v \in V}[H_2^x]^V \setminus \{v\} \cup \{v_{\text{new}}\} & \\
& \text{where} & v_{\text{new}} & = rs(a[x](\tilde{l})) \setminus \{old(x)\} \cup v_i \\
\text{Case iii) describes that after evaluation of the action} & \text{rs(a[x](\tilde{l})), every possible value} & \text{v_i from the set V is replaced by a new} & \text{value v_{new}, which is obtained by the interpretation of} & \text{the action, rs(a[x](\tilde{l})), where any occurrence of} & \text{\{old(x)\} is replaced by the value of v_i.} & \\
\text{Lst. 5 shows an example of a client that uses a Counter} & \text{object \texttt{c}. During the initialisation phase of the object \texttt{c}} & \text{the PointsTo\texttt{(c.data,1,0)} predicate is obtained from which} & \text{the permission part, Perm\texttt{(c.data,1)}, is transferred into the} & \text{lock. Thus, the client obtains both the Init and the Hist} & \text{predicates for the value \texttt{data} (line 4). The client starts a} & \text{new thread \texttt{t} and then both threads running in parallel use} & \text{the same Counter object: thread \texttt{t} increments the value} & \text{\texttt{c.data}} & \text{by 1 (line 6), while the client thread assigns \texttt{c.data}} & \text{to 4 (line 8). The Hist token is divided into two parts (line} & \text{7), so both threads record the change in their own history.} & \text{At the end, both histories are merged (line 11). The client,} & \text{holding both Init and the full Hist token can reason that the} & \text{value of \texttt{data} at the end is either 4 or 5 (line 15).}
\end{align*}
\]
Future Work

Our next goal is to reason about more complex concurrent data structures. For this, we expect that our technique can be applied, if the specifications are expressed in terms of actions over a ghost field that represents the real data structure. Next, we plan to extend the definition of an action to allow more expressive specifications. Furthermore, we plan to analyse scenarios where the order of action execution might depend on the program state (for example scenarios using the wait/notify pattern). Our initial idea is to allow specifying a partial order between actions.

Acknowledgements

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

6. REFERENCES

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