2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

VERCORS: a Layered Approach to Practical Verification of Concurrent Software

Afshin Amighi Stefan Blom Marieke Huisman
Formal Methods and Tools
University of Twente
Enschede, The Netherlands
Email: {a.amighi, s.blom, m.huisman}@utwente.nl

***Abstract*—This paper discusses how several concurrent program verification techniques can be combined in a layered approach, where each layer is especially suited to verify one aspect of concurrent programs, thus making verification of concurrent programs practical. At the bottom layer, we use a combination of implicit dynamic frames and CSL-style resource invariants, to reason about data race freedom of programs. We illustrate this on the verification of a lock-free queue implementation. On top of this, layer 2 enables reasoning about resource invariants that express a relationship between thread-local and shared variables. This is illustrated by the verification of a reentrant lock implementation, where thread-locality is used to specify for a thread which locks it holds, while there is a global notion of ownership, expressing for a lock by which thread it is held. Finally, the top layer adds a notion of histories to reason about functional properties. We illustrate how this is used to prove that the lock-free queue preserves the order of elements, without having to reverify the aspects related to data race freedom.**

I. INTRODUCTION

Over the last years, the ever-growing demands on software performance has led to a significant increase in the use of concurrency. As a side-effect of this development, we also see a growing need for techniques to reason about concurrent software. Software applications are omnipresent, and their failure can have significant economic, societal, and even life-threatening consequences. Naturally, also sequential software should be without failures, however the major complexity of concurrent software makes that the absence of failures only can be guaranteed by using tool-supported formal techniques.

In earlier work, we have introduced the VERCORS tool set [1], which is an annotation based program verifier. VERCORS encodes concurrent Java programs into the intermediate language of Viper [2]) and then uses the Silicon tool to generate proof obligations, which are discharged using Z3 [3].

This paper discusses how VERCORS supports a layered approach to verification, combining different logic-based verification techniques. Each layer captures a different category of properties. In the lowest layer, we care only about data race freedom; in the middle layer we verify resource invariants that relate thread-local variables to globally shared variables, while in the top layer we verify arbitrary functional correctness properties. Strictly speaking, this separation in layers is not necessary, but it helps to keep the specification and verification tractable and to make mechanical verification feasible.

At layer 1, to reason about data race freedom of multiple threads independently modifying a shared object, we use a combination of Implicit Dynamic Frames (IDF) [4] and Concurrent Separation Logic (CSL) resource invariants [5]. In our approach, IDF is used to specify whether a thread has read or write access to a certain location, while CSL-style resource invariants are used to associate synchronizers to shared data (including atomic variables acting as a synchronizer). The combination of these two allows one to verify that a shared-memory concurrent program is free of data races in a thread-modular way: if a thread has write access to a location, then no other thread can simultaneously access this location, and thus no other thread can observe the changes made to this location; if a thread has read access to a location, then any other thread can also only have read access to this location, and thus the value stored in this location cannot be changed. If shared data is protected by a synchronizer, the synchronizer operations ensure that either one thread has exclusive access to the data protected by the resource invariant; or multiple threads have shared read-only access. This approach works for lock-based programs, as well as for programs that use atomic operations for synchronization.

However, in more complex examples, the resource invariant often needs to express a relationship between thread-local and global/shared state. Therefore, in the 2^{nd} layer, we add the notion of thread-local state. Permissions to access this local state can be split between the CSL invariant of the shared object and the threads, in such a way that when a thread holds the invariant, it can update the shared state and its own local state, while it is able to inspect the state of all other threads. Hence every thread can keep track of its own actions through its own local state, while the CSL invariant expresses consistency between the local state and the shared object.

Finally, in the top layer, we add the notion of history [6], which allows us to verify functional properties in addition to data race freedom. A history captures abstractly the updates to data. Updates to histories are traced locally, and when threads synchronize, their local histories are merged into a global history, capturing the possible interleavings between the local updates. Note that functional properties could be accounted for using resource invariants and thread-locality only. However, this would give rise to large and complex invariants, because the invariant has to take into account how everything fits

978-1-4673-8776-7/16 $31.00 © 2016 IEEE
DOI 10.1109/PDP.2016.107

together. Instead, layer 3 splits the reasoning about functional properties: at the thread level each thread accounts for the actions that it performs on a global data structure, while at the global level the abstract global history is used to show that all threads together have the required behavior.

This paper describes for each layer of the verification stack how the verification is handled in VERCORS. The key idea behind the VERCORS tool set is that it works as a transforming compiler, reducing complex verification problems into a verification problem for basic Separation Logic. As a back end, we use the Silicon verifier [2], thus all annotated programs are encoded into annotated Silver programs, which is the intermediate language used by Silicon, and has dedicated support to reason about access permissions. For each layer in the verification stack, we describe how the encoding into Silicon is defined. Because of our layered approach to verification, the transformation is easily manageable, and can be guaranteed to be correct.

We illustrate the usability of our layered approach by presenting a non-trivial verification example for each layer. At the lowest layer, we verify data race freedom of a lock-free queue implementation, derived from the standard Java API lock-free queue. At the middle layer, we show how a relation between thread-local and shared variables is used to verify an implementation of reentrant locks, again derived from the Java API implementation, ensuring that two threads never can simultaneously hold the lock. At the top layer, we use histories to prove that the lock-free queue implementation preserves the order of elements stored in the queue.

It should be noted that this approach differs from recent proposals for a large range of powerful and expressive logics to reason about concurrent software, such as CAP [7], iCAP [8] and CaReSL [9], in that we do not aim at a highly expressive logic, but instead focus on an easily manageable approach to the verification of concurrent software, by breaking down the verification problem into smaller, more manageable problems. Moreover, the focus of our work is on efficient tool support, reusing currently available technologies, while the focus of these logics is expressiveness, and the ability to capture all concurrent programming patterns.

To summarize, the main contributions of this paper are:

- a layer-based approach to the verification of concurrent software, identifying different kinds of verification problems, which all need their own level of annotations;
- for each verification layer, a discussion how the verification problem is encoded into a simpler verification problem in basic Separation Logic; and
- example verifications at each layer of the verification stack.

The paper is structured as follows: Section II briefly introduces CSL, IDF, and the VERCORS specification language. Section III discusses how the combination of IDF and CSL-style resource invariants allows to verify data race freedom. Section IV then discusses how the relation between global properties and thread-local state can be maintained as part of the resource invariant, while Section V discusses the verification of functional properties. Finally, Sections VI and VII discuss related work and conclusions.

II. BACKGROUND

A. Concurrent Separation Logic (CSL)

Concurrent Separation Logic (CSL) [5] is used to reason about concurrent programs. It employs the *points-to* predicate, along with the *separating conjunction* operator of Separation Logic [10]. The *points-to* predicate specifies the contents of a specific location of the *heap*, and the *separating conjunction* operator $\phi * \psi$ expresses that formulas ϕ and ψ hold for *disjoint* parts of the heap.

CSL's parallel composition rule expresses that each thread can be verified in isolation, provided they affect disjoint parts of the heap. Data that is shared by different threads can be specified using a *resource invariant*. Access to the data specified by the resource invariant can only be obtained (1) by using an exclusive synchronizer, *e.g.*, a lock, or (2) by executing the body of an atomic operation, *e.g.*, a compare-and-set operation; provided that the thread leaving the synchronizer or the atomic operation can re-establish the resource invariant.

Decorating the heap with permissions [11] allows one to reason about multiple threads that concurrently read the same location [12]. In Permission-based Separation Logic (PBSL) an amount of permission $\pi \in (0, 1]$ may be specified for locations: full permission $\pi = 1$ denotes a *write* permission; any fraction $\pi \in (0, 1)$ is interpreted as a *read* permission. Threads can split and combine permissions. Soundness of the logic guarantees that the sum of the permissions on a location never exceeds the full permission 1, the only operations that can transfer permissions are ordered in the happens-before relation, and as a result, any verified concurrent program is free of data races.

B. Implicit Dynamic Frames (IDF)

IDF [4] is another program logic that extends Hoare Logic with the ability to reason about access to the heap by means of access permissions to heap locations, similar to PBSL. However, IDF and PBSL differ in how value-specifications are handled: in IDF, one uses side-effect-free expressions in the underlying programming language, while in PBSL, one first relates the program variables to logical variables and then states properties about these logical variables.

To encode the behavior of language constructs that are not part of IDF, we will use two of its proof commands. The **exhale** command first asserts that a formula is true and then drops the resources specified by the formula. The **inhale** command assumes the given formula and adds the resources specified by the formula.

C. VERCORS *Specification Language*

The VERCORS specification language uses a JML-like syntax [13], combined with PBSL and IDF ingredients: resource invariants are essentially defined in PBSL, while our encoding target language Silicon uses IDF exclusively. In VERCORS, predicates have type **resource**, the separating conjunction is written $**$, a full permission is written **Perm**(x,1), and a

fractional permission is written **Perm**(x,p) where p $\in (0, 1)$. If the value of p is not known or unimportant, then one may write **Perm**(x,**read**) or **Value**(x).

Some resource formulas, known as *groups*, can be split into multiple parts that may be shared. To denote such a share of a group, we use the notation [<fraction>]<group>. The VERCORS specification language also supports ghost variables and specification commands, such as **assert**. When evaluating expressions, it may be necessary to inject ghost code just before evaluating a subexpression. This is what the **with**{ G } annotation does, where G can be arbitrary ghost code, including proof hints. Similarly, the annotation **then**{ G } inserts ghost code immediately after the evaluation of a sub-expression. Moreover, in case of a method call, the **with** block may be used to pass ghost in-parameters to the method, using assignments, and the **then** block may be used to inspect the value of ghost out-parameters.

III. LAYER 1: PERMISSIONS AND RESOURCE INVARIANTS

At the bottom layer of the VERCORS verification stack, a combination of IDF and CSL is used to reason about data race freedom. As explained, resource invariants capture access to the shared state. There are two ways to access shared state: by using locks (and other synchronizers), and by using atomics. Here we focus on atomic operations, and their encoding into Silicon, however reasoning about shared state protected by a lock is done in a similar way.

We illustrate our approach by discussing the verification of a lock-free queue, derived from ConcurrentLinkedQueue in the Java API. This example was also specified and verified by Jacobs *et al.* in richer logics [14], [15], but our version is a third shorter in length.

A. Reasoning about Atomic Blocks

In the VERCORS tool, internally an atomic operation on object o with body S is modeled as **atomic**(o){S}. The resources associated to object o are specified by defining an appropriate resource invariant csl_invariant, which has to be established when the object is initialized, thus making it an implicit postcondition of all constructors of the class that defines the resource invariant.

To reason about atomic operations, CSL uses the rule [ATOMIC] [16], which expresses the following: any thread executing an atomic operation with a body S, obtains the shared state specified by the resource invariant csl_invariant; then, while executing S it is allowed to violate csl_invariant, but after the execution of S, csl_invariant has to be re-established and released to the shared state.

$$\frac{\{\text{o.csl\_invariant()} * P\}\ \text{S}\ \{\text{o.csl\_invariant()} * Q\}}{\{P\}\ \textbf{atomic}(\text{o})\{\text{S}\}\ \{Q\}}\ [\text{ATOMIC}]$$

B. Encoding of Atomic Blocks

In Java, we do not directly write **atomic** statements. Instead, the java.util.concurrent.atomic package provides several classes, whose methods perform the atomic operations get, **set**, and compareAndSet for all Java types. Thus, the encoding of

```
resource csl_invariant() = Value(begin) **
  RPerm(head) ** ([read]reachable(begin,head.val)) **
  RPerm(tail) ** ([read]reachable(begin,tail.val)) **
  Perm(last,1)  ** ([read]reachable(begin,last)) **
  chain(head.val,last) ** RPointsTo(last.next,null);
```

Fig. 1. CSL resource invariant of the lock-free queue.

atomic methods from Java into Silicon is done in two steps: first the atomic method call is transformed into a VERCORS **atomic** instruction, and in the next step, the use of the proof rule [ATOMIC] is encoded into Silicon.

Java's atomic operations are encoded as VERCORS **atomic** instructions, using several additional ghost variables to store the results of argument evaluation. For example, if var is an AtomicInteger then res=var.compareAndSet(expect,replace) is internally transformed into:

```
int obj=var; int x=expect; int v=replace;
atomic(this){
  if (obj.val==x) { obj.val=v; res=true; }
  else { res=false; }
}
```

Note that var, expect and replace are evaluated outside of the atomic block. The call to compareAndSet might be annotated with ghost annotations **with**{ G1 } **then**{ G2 }, to maintain the resource invariant, or to provide proof hints to prove correctness of the atomic body. In that case, these ghost instructions are evaluated inside the atomic block, after lines 2 and 4, respectively.

Finally, the encoding of the [ATOMIC] proof rule in Silicon is simple; each occurrence of the statement **atomic**(o){S} is replaced by the sequence:

```
inhale o.csl_invariant();
S;
exhale o.csl_invariant();
```

This transformation, first, uses the instruction **inhale** to add the resources and knowledge of the resource invariant. Then, using the added resources the body of the atomic block S is verified, and finally, **exhale** checks that the resource invariant holds and then removes it.

C. Verification of a Non-blocking Queue

We demonstrate the usability of our approach by verifying data race freedom of the essential methods of ConcurrentLinkedQueue from the java.util.concurrent library, which implements a lock-free queue as proposed by Michael and Scott [17]. First, we briefly explain the data structure, and then we describe how the class is specified and verified.

1) Implementation: The queue consists of (1) two atomic references: head and tail, and (2) a chain of nodes, where each node contains a value field and an atomic reference field to the next node. The head points to a *sentinel* node, *i.e.*, its value does not contribute to the queue. The last node of the queue can be identified by its null-valued next field. A queue is empty when both the head and the tail point to the sentinel node with a **null**-valued next field.

2) Specification: The main part of the specification is the resource invariant, which characterizes a valid queue structure.

```
/*@ requires Value(head) ** Value(tail);
    ensures  Value(head) ** Value(tail)
    ** (\result != null ==> Perm(\result.val,1)); @*/
Integer try_deq(){
  Node n1,n2; boolean tmp; Integer res=null;
  n1=head.get();
  n2=n1.next.get()/*@ with {
    lemma_readable_or_last(this.begin,n1); } @*/;
  if (n2!=null) {
    tmp=head.compareAndSet(n1,n2)/*@ with {
      if (head.val==n1) { unfold chain(head.val,last);}} @*/;
    if(tmp){ res=new Integer(n2.val); }
  }
  return res; }
```

Fig. 2. Dequeue attempt.

The specification additionally uses two ghost fields with type Node: (1) begin, which represents the original head of the queue, *i.e.*, the head of the queue when the data structure is initialized, and (2) last, which points to the last node of the queue.

The resource invariant uses several auxiliary predicates. First, RPerm and RPointsTo, which combine permission to read an AtomicNode with **Perm** and **PointsTo** on the embedded field, respectively. Next, the reachable(n,m) predicate captures that there is a path from n to m, and chain(n,m) specifies full ownership of the data element in the nodes of the queue located between n and m. The resource invariant (see Fig. 1) states that begin can be read and is immutable. The fields head.val, tail.val, and last are writable and reachable from begin. The elements between head and last are fully owned and the last.next is writable and **null**.

3) Verification: The essential part in the verification of the lock-free queue is proving that all atomic operations preserve the resource invariant. In addition to our encoding, this uses the following lemmas (which all have inductive proofs):

(i) The reachable predicate is transitive.

(ii) Given a node from which both the last node and some other node are reachable, either the other node is the last node, or the next node of the other node is also reachable.

(iii) Appending one node to a permission chain yields a permission chain.

Fig. 2 shows a fully specified implementation of the method try_deq, which attempts to dequeue a node of the queue. First, it copies the current head of the queue to n1. Next, it copies the next of n1 to n2, which is allowed because due to lemma (ii) we have either write or read. If n2 is not **null** then the queue was not empty and compareAndSet is used to change the head to the next node. Upon success the element is returned as an Integer. In all other cases **null** is returned to signal failure. A full specification of the queue and online version of the tool is available at [18].

IV. Layer 2: Relating Thread-Local and Global Variables

To understand why in layer 2 we add the notion of thread-local state, it is important to realize that the queue specification above does not allow threads to express any property about the elements in the queue, even though the specification of the queue describes the queue's behavior in terms of all elements the queue has held and still holds. This list, however, is only available for reasoning during atomic operations, because the access permissions on the list are maintained in the invariant. Hence, it is not possible for threads to reason about the elements of the queue outside of atomic regions, and worse, because a thread does not have any permission on these variables outside of atomic regions, it is forced to forget all knowledge about them: after all, any other thread might modify them.

The simplest way to avoid this loss of information is to add thread-local state and to keep this thread-local state synchronised with the global state. The concept of thread-local state is old; it is already used in the classical example of Owicki-Gries [19] where two threads independently atomically increment a variable by one. To prove that the end result increases the initial value by two, two thread-local ghost variables are used that account for the behavior of each thread. These ghost variables are then used to state a resource invariant that precisely captures the value of the shared variable.

In our approach, this combination of thread-local and global state is achieved as follows. Full permission on the shared variable is kept in the invariant, thus it may be modified during atomic operations. For thread-local state, half the permission is held by the invariant, thus it may be read to specify the relation with the shared state. In addition, each thread holds the other half permission on its own thread-local state, which means that it can read its thread-local state at any time during execution, and moreover, it has the ability to change its own thread-local state when it holds the resource invariant, *i.e.*, during atomic operations.

A. Encoding

The concept of a thread-local variable is present in many programming languages. Typically, thread-locality is not a primitive of the language, but it is added by means of a library. For the moment, we use a manual encoding of thread-locality on top of layer 1. Our encoding assumes that an application has access to a fixed number of unique objects (or integers) identifying each of the threads, which allows to encode thread-local variables as an array, where each array element corresponds to the thread-local variable for the corresponding array index.

Additionally, a special treatment is required for reasoning about the current thread id. We have a specification construct **current_thread**, which yields the id of the current thread. Thus, as its semantics depends on the thread in which it is evaluated, **current_thread** cannot be used in invariants. In fact, it may not be used in any predicate, unless the specification modifier **thread_local** is used for the predicate. Moreover, any predicate that invokes a **thread_local** predicate has to be marked **thread_local** itself.

We encode **current_thread** by adding it as an argument to all methods, constructors, and **thread_local** predicates and all their invocations. This allows us to detect illegal use

of **current_thread**, *i.e.*, in a non-**thread_local** predicate, because every illegal use would result in a local variable that was not declared. Moreover, we check that csl_invariant is not declared **thread_local**.

B. Specification and Verification of a Reentrant Lock.

To illustrate the kind of verifications that can be done at layer 2, we discuss the specification and verification of the implementation of a reentrant lock. The specification is adapted from [12], [20], while the implementation is a simplified version of OpenJDK6 java.concurrent.locks.ReentrantLock.

The major challenge in specification and verification is the *reentrant* lock behavior. In separation logic, the specification of the behavior of non-reentrant locks is simple: when obtaining the lock, the resource invariant attached to the lock, *i.e.*, access to the shared data protected by the lock, is also obtained and upon unlocking the invariant must be released. Assuming that double locking leads to an unchecked error or a deadlock, this behavior can be specified with straightforward contracts. However, for reentrant locks more care is required: the resource invariant is only obtained upon locking for the first time and it must be yielded when unlocking for the last time only. This means that when obtaining the lock, the invariant can only be obtained if there is proof that the lock is obtained for the first time.

1) Implementation: We follow the ReentrantLock implementation in OpenJDK6 by having two fields: an atomic integer count and an integer owner. The latter variable is set to the thread id of the current owner of the lock, or -1 otherwise. If a thread already is the owner then a (re)lock is done by atomically increasing count. Otherwise, the lock must be obtained by changing count from 0 to 1 using compare-and-set. To release the lock, the count is decreased, where the owner must be cleared before the final decrease to 0.

2) Specification: The specification of a reentrant lock (see Listing 3) uses the predicate lockset(S), where S is a multi-set of locks. The predicate lockset(S) holds for a thread if the multiplicity of any lock in the lock set is the number of times the thread holds that lock. Hence, obtaining a lock adds the lock to the lock set, while releasing a lock means removing it from the lock set. Moreover, when a lock does not occur in the lock set before locking, the resource invariant is obtained, and when it is no longer present after unlocking, it has to be yielded. In our previous work [12], the lockset(S) was added to the specification language as a primitive. In this paper, we will define it in terms of the current thread and an invariant.

3) Invariant: To define the invariant for a lock (see Fig. 4), we use several ghost variables. Without loss of generality, we assume a fixed number of threads (T), which is also a ghost variable. We use a ghost array held with T entries that functions as the thread-local count for each thread. And we use a ghost variable holder that tracks the owner of the lock. Note that we cannot use the implementation field owner because it cannot change at the same time as the implementation field count changes. Ghost fields can change at the same time and thus preserve an invariant.

```
interface Lock {
//@  resource lock_invariant();
/*@
  given  bag<Lock> S;
  requires lockset(S);
  ensures  lockset(S+seq<int>{this});
  ensures  !(this \memberof S) ==> lock_invariant(); @*/
 void lock();
}
```

Fig. 3. Interface Lock.

```
resource csl_invariant()= Value(T) ** T > 0 **
  Value(held) ** Value(subject) **
  APerm(count,1/2) ** APerm(owner,1/2) **
  (count.val == 0 ==> subject.inv() **
       APerm(count,1/2) ** APerm(owner,1/2)) **
  Perm(holder,1) ** -1 <= holder < T **
  (holder == -1) == (count.val == 0) **
  (\forall* int i; 0 <= i < T ;
   Perm(held[i],1/2) ** (i!=holder ==> held[i]==0)
   ** held[i] >= 0 ** (held[i]==0 ==> owner.val!=i)
  );
  resource lockset_part()=
    Perm(held[\current_thread],1/2) **
    (held[\current_thread] > 0 ==>
      APointsTo(count,1/2,held[\current_thread]) **
      APointsTo(owner,1/2,\current_thread));
```

Fig. 4. The definition of the lock invariant in the Lock class.

Permission for the various fields of a lock are divided between the invariant and the lockset_part predicate that will be used in the lockset definition. The invariant holds permission $\frac{1}{2}$ on each element of the held array and the count and owner atomic fields. The other $\frac{1}{2}$ for held elements is held in the corresponding lockset, while the other $\frac{1}{2}$ for the atomic fields is kept in the lockset for the owning thread and in the invariant if the lock is free.

The lockset predicate is defined in Fig. 5. The most important lines are the last two: for every lock, the lockset holds the permissions defined in lockset_part and the held count for the current thread is precisely the multiplicity of the lock id in the lockset.

The full listing of the lockset specified implementation of our reentrant lock can be found online [18].

V. Layer 3: Functional Properties using Histories

Invariants, with or without thread-locals, are adequate for specifying and verifying data race freedom and basic functional properties. The verification of more complex functional properties, however, can get very tedious because all interactions between threads have to be specified in great detail. Therefore, this section discusses a different approach, adding the notion of histories [6] on top of layer 2, and uses this to prove that the order of elements is preserved in the lock-free queue.

A. Reasoning with Histories

The key idea of history-based reasoning is that functional verification is not performed on the program directly, but on

```
thread_local resource lockset(bag<int> S)=
  Value(T) ** 0 <= \current_thread < T **
  Value(L) ** L > 0 ** Value(locks) **
  (\forall* int l ; 0 <= l < L ;
    Value(locks[l]) ** Value(locks[l].subject) **
    Value(locks[l].T) ** locks[l].T==T **
    Value(locks[l].held) ** locks[l].lockset_part() **
    locks[l].held[\current_thread]==(l \memberof S)
  );
```

Fig. 5. The definition of the lockset predicate in the Thread class.

an abstract model of the program. This idea combines data abstraction [21] with process algebra [22].

An abstract model is defined in terms of variables and actions on those variables. Each action abstracts away from a concrete operation that can be carried out on the program data: an action contract specifies which concrete operations the action corresponds to. For example, Fig. 7 specifies an abstract model for queues: q specifies a queue state, while get is an abstract action on the queue. Actions can be combined into processes using standard operators, known from process algebra, such as choice, sequential composition, and parallel composition. To capture action repetition, the behavior of processes also can be described using a recursive definition, which must be paired with a contract. See for example the definition of process get_all in Fig. 7 (lines 11-14).

In the method specifications, we record the local history of the actions performed in a thread. For example, the method specification of method get in Fig. 8 expresses that this method performs an abstract get action. In the method bodies, annotations are added to mark blocks that implement actions. For example, in the method try_deq in Fig. 2 we add

{ **action** hist, p , P, hist.get(n2.val); hist.q=tail(hist.q); }

after the **unfold** at line 11, stating that we extend the history hist, for which we own a fraction p and which is P, with an action get. This is done by removing the head element from hist.q.

Typically, whenever a thread is created, it starts with an empty local history. When threads terminate and are joined, the local history of the terminated thread is merged with the history of the joining thread. Eventually, this results in one global history of abstract actions over which the desired functional property can be verified. To guide the verification, some additional annotations for the treatment of histories may be provided as proof hints: initialize a new, empty history over a set of program fields, destroy a history etc., see [6] for a full overview.

The verification of these annotations consists of two main tasks. First, the code must be verified to ensure that it implements a linearizable sequence of actions, as specified in the history annotations. Second, the history specification has to be verified to ensure that every possible trace satisfies the behavior specified in the form of the process contracts.

This has two advantages for the verification of functional properties. First, we can abstract from implementation details (e.g. the linked list in the queue becomes a sequence in the history). Second, because we have already verified data race freedom, we can verify the properties in a non-deterministic sequential setting, which makes it less complicated.

```
/*@ requires Perm( q , 1 ) ** PointsTo( q_mode , 1 , 0 );
 ensures Perm( q , 1 ) ** PointsTo( q_mode , 1 , 1 )
  ** Perm( q_init , 1/2 ) ** q == \old(q) ** q_init == \old(q)
  ** hist_passive(1,p_empty()); @*/
void create_hist();
```

Fig. 6. The method that encodes creating a history

B. Encoding

In theory, histories are defined over arbitrary sets of locations. The input language for the tool however does not use locations as first class citizens, so it defines histories over the fields of a History class. Actions and processes are also defined using an appropriate ADT in the same class. The predicate **Hist** that describes (part of) the recorded history has three arguments: a reference to a history object (instead of a set of locations), a fraction and a process expression that denotes the history accounted for. The initial state of a history is specified using the predicate HistInit, whose arguments are a reference to a history and a formula. For example, starting with an empty queue is specified as HistInit(hist,q==**seq**<**int**>{}).

To complete the first verification task *i.e.*, to ensure linearizability, modifications of the fields of the history object have to be grouped in action blocks, which must keep full permission on every field written during the action for the duration of the entire action. This is managed by having three forms of permissions on the fields: normal (**Perm**), passive (**HPerm**) and active (**APerm**). Passive permissions can only be used to read from history fields. To write a field you need full active permission. Permission changes, as described in [6], are encoded by replacing every history annotation by a method call whose contract matches the behavior of the proof rules for the annotation. For example, Fig. 6 shows the method that encodes the creation of a history. Note how the initial state is in an extra ghost field (q_init) in order to be able to reason about it. The hist_passive predicate encodes **Hist**(**this**,1,empty). Note how the empty expression is replaced by an expression in the ADT.

The verification of action blocks records the initial state of the variables to be modified in ghost fields (for checking the actions post-condition), exchanges full passive for full active permission, and assumes any pre-condition of the action. In addition, the encoding of the primitive **Hist** predicate is inhaled. At the end of the action block, it asserts the post-condition of the action and changes the permissions back. In addition, the encoding of the **Hist** predicate with an extra action appended is exhaled.

The second part of the verification is to show that every execution trace of a history satisfies its contract. To do this, we generate a method for every defined process, whose body is constructed as follows: sequential composition on processes becomes sequential composition of statements, choice on processes becomes a non-deterministic choice, and every mention

```
public class History {/*@
  seq<int> q;

  modifies q;  ensures \old(q)==seq<int>{e}+q;
  process get(int e);

  modifies q;  ensures \old(q)==es+q;
  process get_all(seq<int> es)=
   |es| == 0?empty:(get(head(es))*get_all(tail(es)));

  ensures get_all(es)*get(e)==get_all(es+seq<int>{e});
  void get_lemma(seq<int> es,int e){ if (|es|>0){
   get_lemma(tail(es),e);
  }}

  modifies q;  ensures \old(q)+input==output+q;
  process feed(seq<int> input,seq<int> output)=
   put_all(input)||get_all(output);
```

Fig. 7. Fragment of History Specification for Queues

of an action or a defined process becomes a method invocation. For example, the method generated for put_all is

```
ensures q==\old(q)+es;
void put_all(seq<int> es){
 if (|es|==0){
 } else {
  put(head(es));
  put_all(tail(es));
 }}
```

Note that we call put_all in the body. This is safe because this call is guarded by a call to the action method put, which means that by induction on the length of a trace we can assume that this call satisfies its contract. If a process expression contains unguarded calls, the laws of process algebra are used to compute an equivalent guarded form.

C. Verification of a Queue History

To illustrate reasoning about complex functional properties using histories, we prove a functional property for the lock-free queue discussed in layer 1: if one thread is given an array with elements that it puts into an empty queue, and a second thread is given an array of the same length that it fills by getting elements from the queue then, once both threads terminate, the contents of both arrays are identical. Essentially, this captures that the order of elements is preserved in the queue.

First, Fig. 7 shows part of the history specification. The data managed by the history is a single field q that contains a sequence of integers, *i.e.*, an abstraction of the queue contents. Next, the contract of the get action shows that it removes the first element of q. Then, we define process get_all, which appends a whole sequence of integers to the queue. The history specification is extended with a lemma that shows a useful property about the get_all process, namely that the sequential composition of a get_all and a get is again a get_all. Finally, we define the feed process on whose contract the whole verification hinges; it states the behavior of putting and getting two sequences in parallel: the old contents plus the new elements have to be the same as the retrieved elements plus the current state.

```
//@ History hist;
//@ boolean hist_active;
/*@ given frac p;  given process P;
    requires Value(hist) ** Hist(hist,p,P)
      ** p!=none ** PointsTo(hist_active,p/2,true);
    ensures  Value(hist) ** Hist(hist,p,P*hist.get(\result))
      ** p!=none ** PointsTo(hist_active,p/2,true);
@*/ public int get();
```

Fig. 8. History specification of the get method.

```
public void run(){
 int N=output.length;
 int i=0;
 //@ vals=seq<int>{};
 /*@ loop_invariant Value(queue) ** Value(queue.hist)
 ** Value(output) ** Perm(vals,1) ** 0 <= i <= N
 ** i==|vals| ** N==output.length
 ** PointsTo(queue.hist_active,1/4,true)
 ** (\forall* int k; 0 <= k < N ; Perm(output[k],1))
 ** (\forall int k; 0 <= k < i ; output[k]==vals[k])
 ** Hist(queue.hist,1/2,queue.hist.get_all(vals)); @*/
 while(i<N){
  output[i]=queue.get()/*@ with {
     p=1/2 ; P = queue.hist.get_all(vals);} @*/;
  //@ vals=vals+seq<int>{output[i]};
  i=i+1;
 }
}
```

Fig. 9. Specified run method of the receiver

Fig. 9 shows the run method of the receiver, annotated with a loop invariant that describes its behavior (the method contract itself is not shown as it is essentially an instance of the loop invariant). The body of the loop uses the get method specified in Fig. 8 to fill the array. Note how the ghost field vals is used to maintain the list of elements written into the array. Also note that the loop invariant only refers to the output array and the vals field: the specification does not depend on the behavior of other threads that may operate on the queue. The comparison of the orders of the input and the output occurs when the two threads are joined by the thread that forked them. This thread knows that: (i) at first q was empty, (ii) the program in parallel put all input array elements into the queue and got all output array elements from the queue. (iii) these arrays have the same length. What (ii) says is that the program performed the process feed (Fig. 7, line 16) for the contents of the two arrays. From the contract of that process, it can be inferred that those contents are the same and the current q is empty too. Thus, modular verification is achieved.

Finally, we revisit the lock free queue try_deq method in Fig. 2. To add history support we do the following:

- We add a ghost field History hist; to keep the history state.
- We add a ghost variable **boolean** hist_active=**true**; that denotes if the history is active.
- We modify the definition of chain to have a third argument that contains the contents of the chain, and we propagate this change to all uses of chain, including the lemmas.
- We change chain(head.ref,last) in the invariant to **Perm**(hist_active,1/2) ** **Value**(hist) ** (hist_active ==> **HPerm**(hist.q,1)**chain(head.ref,last,hist.q))

i.e., we can access list_active and hist, and while the history is active we hold protected permission for hist.q, whose value is precisely the contents of the queue.

- We insert an action block after the **unfold** at line 11 to keep the queue contents and hist.q equal, as discussed above:
  { **action** hist, p , P, hist.get(n2.val); hist.q=tail(hist.q); }

VI. Related Work

As already mentioned above, various program logics have recently been proposed to reason about concurrent programs, *e.g.*, CAP [7], iCAP [8], CaReSL [9] , TaDA [23] and IRIS [24]. These are all highly expressive logics, which are able to reason about similar properties as discussed in this paper, but as far as we are aware, there is no tool support for them, while our focus is to make verification of concurrent programs practical. The difference between CAP and our approach is that we use a predicate as the resource invariant, rather than an arbitrary boxed formula. As a result, we can only use explicitly declared variables to specify a relation between the local state and the invariant. This is less elegant, but also prevents the problem of stability of boxed formulas that is caused by implicitly crossing the boundary between the shared and the local state. Essentially, iCAP enriches CAP with impredicative protocols and CaReSL introduces thread-locals to modularly reason about fine-grained data-structures. However, so far there is no tool support for these program logics. Moreover, these logics share the property that functional verification happens in parallel with the verification of the data race freedom. In contract, using histories the functional verification happens separately. The logic TaDA has one feature that our logic does not (yet) have: it allows proving that a method behaves as if it atomically performs an action, while we can only axiomatize this.

Closely related to our work, Jacobs and Piessens propose a technique to verify functional properties of lock-free data-structures involving atomic operations in VeriFast [14], which has recently been extended with support for Rely-Guarantee reasoning [15]. They also study the Michael-Scott queue as an example and their work is implemented in VeriFast. As far as data race freedom is concerned, their work and ours are identical in concept, but quite different in the organization of the specification language. For example, we write glue code in **with** and **then** blocks for every atomic method invocation. They declare several ways in which an atomic method can operate up front. For the functional properties, their method does not achieve the complete separation between local and global reasoning that is enabled by histories. Also, their notion of action is less general than ours, as theirs is limited to a single atomic operation, while ours can combine multiple atomic operations into a single action.

Compared to the logics mentioned above, we use a simple form of invariant. Due to this simplicity, our invariants are easy to verify by encoding them in existing tool supported languages. Using thread local variables in a systematic way, our notion is powerful enough to prove data race freedom. Rather than designing extra features for functional verification into the invariant mechanism, we use a separate mechanism which offers a better separation of concerns.

The claim made about IRIS [24] that invariants and monoids are all that one needs to reason about concurrent software is in spirit identical to the claim we make in this paper. Invariants in our approach are similar to IRIS's invariants, but, while our process algebra is a monoid, the mechanism for executing actions has no equivalent in IRIS. Each action block consists of a number of atomic steps that are independent of any other step that it may be interleaved with.

Our logic also satisfies the requirements put forth by Da Rocha Pinto et al. [25]: Our thread-local state is auxiliary state, histories provide interference abstraction, we have resource ownership through separation logic, and we get atomicity through the use of atomic methods and blocks.

Our approach is also in line with the works of Jones et al. [26], [27], which propose a limit to the expressive power of specification formalisms in order to keep specifications analyzable and warn of not using auxiliary variables beyond the point where they are appropriate.

We have used thread-local specification patterns in our earlier work on atomic operations [28], OpenCL kernel programs [29], and Parallel Loops [30]. The encodings used to implement the latter two are similar in style to the ones introduced in this paper: they translate the proof requirements into a method that must be checked and modify code by inserting **exhale** and **inhale** instructions.

Rely-Guarantee reasoning [31] is a reasoning style that is intuitive and often elegant. We believe it is possible to encode this style into layer 2 at the price of a large amount of (automatically) added annotations. It is much easier to employ rely-guarantee reasoning at layer 3: we can put the properties on which an action relies as its pre-condition and put the properties it guarantees as its post-condition.

VII. Conclusion

In this paper, we have shown how a layered combination of CSL-based verification approaches for concurrent programs can be used to make mechanical verification feasible and practical. Each layer focuses on a particular class of properties, and reuses the results of the lower layers. The layered approach enables a compositional encoding into a simple verification language, Silicon, with appropriate tool support. Because the encoding focuses on a simple aspect of the verification, it is easier to become convinced of the correctness of the encoding. We also illustrate how verification can take advantage of the layered approach. In particular, at layer 3, we verify a functional property of a lock-free queue, for which we have already shown data race freedom at the lower layer 1. Moreover, in layer 2, we verify correctness of a standard reentrant lock implementation with respect to its contracts mainly specifying its reentrancy properties [20].

As future work, we consider several directions. It might be possible to add other layers to the stack (or have a branching structure of verification techniques) to enable verification of other classes of properties. We also see that verification at

the moment requires a large amount of annotations, and we plan to investigate if some of these can be generated automatically. Finally, we need to do large verification case studies, to show how well the mechanical verification scales. Especially, in layer 2, we are interested in case studies with richer concurrency protocols.

ACKNOWLEDGMENT

The work presented in this paper is supported by ERC grant 258405 for the VerCors project.

REFERENCES

[1] S. Blom and M. Huisman, "The VerCors tool for verification of concurrent programs," in *FM 2014: Formal Methods*, ser. LNCS, C. Jones, P. Pihlajasaari, and J. Sun, Eds. Springer International Publishing, 2014, vol. 8442, pp. 127–131, doi:10.1007/978-3-319-06410-9_9.

[2] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers, "Viper: A verification infrastructure for permission-based reasoning," ETH Zurich, Tech. Rep., 2014.

[3] L. de Moura and N. Bjrner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. LNCS, C. Ramakrishnan and J. Rehof, Eds. Springer Berlin Heidelberg, 2008, vol. 4963, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[4] J. Smans, B. Jacobs, and F. Piessens, "Implicit dynamic frames: Combining dynamic frames and separation logic," in *ECOOP 2009 Object-Oriented Programming*, ser. LNCS, S. Drossopoulou, Ed. Springer Berlin Heidelberg, 2009, vol. 5653, pp. 148–172, doi:10.1007/978-3-642-03013-0_8.

[5] P. W. OHearn, "Resources, concurrency, and local reasoning," *Theoretical Computer Science*, vol. 375, no. 13, pp. 271 – 307, 2007, festschrift for John C. Reynoldss 70th birthday.

[6] S. Blom, M. Huisman, and M. Zaharieva-Stojanovski, "History-based verification of functional behaviour of concurrent programs," in *Software Engineering and Formal Methods*, ser. LNCS, R. Calinescu and B. Rumpe, Eds. Springer International Publishing, 2015, vol. 9276, pp. 84–98, doi:10.1007/978-3-319-22969-0_6.

[7] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis, "Concurrent abstract predicates," in *ECOOP 2010 Object-Oriented Programming*, ser. LNCS, T. DHondt, Ed. Springer Berlin Heidelberg, 2010, vol. 6183, pp. 504–528, doi:10.1007/978-3-642-14107-2_24.

[8] K. Svendsen and L. Birkedal, "Impredicative concurrent abstract predicates," in *Programming Languages and Systems*, ser. LNCS, Z. Shao, Ed. Springer Berlin Heidelberg, 2014, vol. 8410, pp. 149–168, doi:10.1007/978-3-642-54833-8_9.

[9] A. Turon, D. Dreyer, and L. Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency," in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '13. New York, NY, USA: ACM, 2013, pp. 377–390, doi:10.1145/2500365.2500600.

[10] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. [Online]. Available: http://dl.acm.org/citation.cfm?id=645683.664578

[11] J. Boyland, "Checking interference with fractional permissions," in *Static Analysis*, ser. LNCS, R. Cousot, Ed. Springer Berlin Heidelberg, 2003, vol. 2694, pp. 55–72, doi:10.1007/3-540-44898-5_4.

[12] A. Amighi, C. Haack, M. Huisman, and C. Hurlin, "Permission-based separation logic for multi-threaded Java programs," *Logical Methods in Computer Science*, vol. 11, no. 1, pp. 1–66, February 2015, doi: 10.2168/LMCS-11(1:2)2015.

[13] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, 2005, doi:10.1007/s10009-004-0167-4.

[14] B. Jacobs and F. Piessens, "Modular full functional specification and verification of lock-free data structures," Department of Computer Science, K.U.Leuven, CW Reports CW551, 2009.

[15] J. Smans, D. Vanoverberghe, D. Devriese, B. Jacobs, and F. Piessens, "Shared boxes: rely-guarantee reasoning in VeriFast," Department of Computer Science, KU Leuven, CW Reports CW662, May 2014.

[16] V. Vafeiadis, "Concurrent separation logic and operational semantics," *Electronic Notes in Theoretical Computer Science*, vol. 276, pp. 335 – 351, 2011, twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII).

[17] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96. New York, NY, USA: ACM, 1996, pp. 267–275, doi:10.1145/248052.248106.

[18] "The VerCors tool online." [Online]. Available: http://www.utwente.nl/vercors/

[19] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, no. 4, pp. 319–340, 1976.

[20] A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski, "Formal specifications for Java's synchronisation classes," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, Feb 2014, pp. 725–733, doi:10.1109/PDP.2014.31.

[21] C. Hoare, "Proof of correctness of data representations," *Acta Informatica*, vol. 1, no. 4, pp. 271–281, 1972, doi:10.1007/BF00289507.

[22] J. A. Bergstra, A. Ponse, and S. A. Smolka, Eds., *Handbook of Process Algebra*. Amsterdam: Elsevier Science, 2001.

[23] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner, "TaDA: A logic for time and data abstraction," in *ECOOP 2014 Object-Oriented Programming*, ser. LNCS, R. Jones, Ed. Springer Berlin Heidelberg, 2014, vol. 8586, pp. 207–231, doi:10.1007/978-3-662-44202-9_9.

[24] 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," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 637–650, doi:10.1145/2676726.2676980.

[25] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner, "Steps in modular specifications for concurrent modules (invited tutorial paper)," in *MFPS 2015*, ser. ENTCS, 2015, in press.

[26] C. Jones, I. Hayes, and R. Colvin, "Balancing expressiveness in formal approaches to concurrency," *Formal Aspects of Computing*, vol. 27, no. 3, pp. 475–497, 2015, doi:10.1007/s00165-014-0310-2.

[27] C. Jones, "The role of auxiliary variables in the formal development of concurrent programs," in *Reflections on the Work of C.A.R. Hoare*, A. Roscoe, C. B. Jones, and K. R. Wood, Eds. Springer London, 2010, pp. 167–187, doi:10.1007/978-1-84882-912-1_8.

[28] A. Amighi, S. Blom, and M. Huisman, "Resource protection using atomics," in *Programming Languages and Systems*, ser. LNCS, J. Garrigue, Ed. Springer International Publishing, 2014, vol. 8858, pp. 255–274, doi:10.1007/978-3-319-12736-1_14.

[29] A. Amighi, S. Darabi, S. Blom, and M. Huisman, "Specification and verification of atomic operations in GPGPU programs," in *Software Engineering and Formal Methods*, ser. LNCS, R. Calinescu and B. Rumpe, Eds. Springer International Publishing, 2015, vol. 9276, pp. 69–83, doi:10.1007/978-3-319-22969-0_5.

[30] S. Blom, S. Darabi, and M. Huisman, "Verification of loop parallelisations," in *Fundamental Approaches to Software Engineering*, ser. LNCS, A. Egyed and I. Schaefer, Eds. Springer Berlin Heidelberg, 2015, vol. 9033, pp. 202–217, doi:10.1007/978-3-662-46675-9_14.

[31] C. B. Jones, "Specification and design of (parallel) programs," in *IFIP Congress*, 1983, pp. 321–332.