The VerCors Project

Setting Up Basecamp

Afshin Amighi Stefan Blom (*) Marieke Huisman Marina Zaharieva-Stojanovski

University of Twente / (*) and Dundalk Institute of Technology

a.amighi/s.blom/m.huisman/m.zaharieva@utwente.nl

Abstract

This paper describes the first results and on-going work in the VerCors project. The VerCors project is about *Verification of Concurrent Data Structures*. Its goal is to develop a specification language and program logic for concurrent programs, and in particular for concurrent data structures, as these are the essential building blocks of many different concurrent programs. The program logic is based on our earlier work on permission-based separation logic for Java. This is an extension of Hoare logic that is particularly convenient to reason about concurrent programs.

The paper first describes the tool set that is currently being built to support reasoning with this logic. It supports a specification language that combines features of separation logic with JML. For the verification, the program and its annotations are encoded into Chalice, and then we reuse the Chalice translation to Boogie to generate the proof obligations.

Next, the paper describes our first results on data structure specifications. We use histories to keep track of the changes to the data structures, and we show how these histories allow us to derive other conclusions about the data structure implementations. We also discuss how we plan to reason about volatile variables, and how we will use this to verify lock-free data structures.

Throughout the paper, we discuss our plans for future work within the VerCors project.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Correctness proofs, Formal methods, Validation; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Mechanical verification, Specification techniques

General Terms languages, theory, verification

Keywords separation logic, concurrency, permissions

1. Introduction

Increasing performance demands, application complexity and explicit multi-core parallelism make concurrency omnipresent in software applications. However, due to the complex interferences between threads in an application, concurrent software is also notoriously hard to get correct. Instead of spending large amounts of money to fix incorrect software, formal techniques are needed to reason about the behavior of concurrent programs.

PLPV'12, January 24, 2012, Philadelphia, PA, USA.

Over the last years, program logics have proven themselves to be useful to reason about the correctness of sequential programs. In particular, several powerful tools, *e.g.*, Key [7], ESC/Java [15], Spec# [4], and KIV [50], have emerged and are used to prove non-trivial programs correct. The theory that underlies these tools dates back to the sixties and seventies, *e.g.*, the work of Floyd [24], Hoare [29], and Dijkstra [19]. This transition from theory to tools is due to several reasons: the increase in computing power, the emergence of languages with a well-defined semantics, such as Java, and the development of powerful automated first-order provers to prove the resulting proof obligations.

For concurrent programs, theory on how to verify them dates back to the seventies (notably the Owicki-Gries method [45]) and eighties (Jones's compositional rely-guarantee method [33]). However, these techniques are too complicated to integrate directly into the existing tools for sequential program verification. Instead, only with the emergence of separation logic [43, 44], the development of tools to verify concurrent programs has come into reach. Originally, separation logic was developed to reason about programs with pointers. The main characteristic of separation logic is that it allows one to reason explicitly about the heap, and in particular that it allows one to state explicitly that two references are pointing into disjoint parts of the memory. This makes separation logic also suitable to reason about concurrent programs [42], because it allows one to express naturally that two threads work on disjoint parts of the memory, and thus cannot interfere with each other.

In earlier work, we have used separation logic to verify concurrent Java programs [26–28]. The logic uses fractional permissions [11] to control read and write accesses. Multiple threads can simultaneously have a permission to read a location, but only at most one thread at a time can have a write permission on a location. Permissions are modeled as a fraction between 0 and 1. A write permission is a full permission: 1, and any non-zero fraction less than 1 gives a read permission. Permissions can be combined, so that threads can regain write permission on a location. Soundness of the logic ensures that if the program can be verified with the logic, it is guaranteed to be free of data races.

The logic captures the main concurrency constructs of Java. In particular, it has rules to reason about thread creation, thread termination (and in particular, other threads obtaining permissions from a terminated thread) and about reentrant locks. Every lock has an associated resource invariant that expresses which permissions a thread obtains when acquiring the lock.

The VerCors project builds on this existing work, and uses permission-based separation logic as the basis for verification. However, while in earlier work the main focus was on proving that a program had no data races, within VerCors the goal is to reason also about the functional behavior of an application. In particular, the VerCors project focuses on the specification and verification of concurrent data structures. Data structures are an essential building

block for many different applications, and thus their correctness is important. In addition, their correctness is really depending on how they handle the data: one does not only want to know that a data structure implementation does not have a data race, but also that data stored in the data structure is handled correctly. In particular, one wishes to verify properties such as: *any data that is stored in the data structure eventually is retrieved from it*, and *data is retrieved in a particular order.*

In addition, within the VerCors project, we also will target other concurrency and synchronization primitives, such as futures, barriers, and reader-writer-locks. This will make the logic applicable to a larger class of Java programs. Moreover, we believe that it will also lead to the development of a generic verification theory for concurrent programs in different programming languages. To support this idea, our intention is to find means to separate the concurrency-specific verification tasks from the functional verification tasks. This would allow, for example, to change the synchronization mechanism without redoing the functional verifications.

Within VerCors, we also plan to look at lock-free data structures and algorithms, and to reason about their correctness. In Java, two notions of lock-freeness can be identified: in the first case, all shared data is volatile, and the Java Memory Model [39] ensures that there are no data races. To support this, the logic has to be extended to reason about volatile variables. In the second case, where non-volatile variables are used, there actually might be data races in the code. For this case, we plan to study whether we can identify classes of so-called *benign data races* that may be allowed explicitly in the code, and where verification still can be done.

This paper describes the very first results and ongoing work in the VerCors project. In particular, it describes the current state of the tool set that supports reasoning within VerCors. It is planned that as a specification language, the tool set uses a combination of the Java Modeling Language (JML) [36] and assertions of permission-based separation logic. The tool set leverages existing program verification tools, in particular Chalice [37] and Boogie [3]. We explain how annotated Java programs are encoded into Chalice. To allow for easy experimentation with the tool set, we have also added a simple teaching language as extra input language. We will briefly describe the main characteristics of this language, and how the tool is set up to easily support other input languages.

To understand the kind of specifications that the tool should verify, we have also started working on the specification of several commonly used concurrent data structures and other concurrency library classes[1]. We have studied in particular the `BlockingQueue` interface, and several of its implementations, and the `AtomicInteger` class.

For each `BlockingQueue` object the specification maintains a history list that keeps track of all changes that are made to the queue. We sketch how histories can be used to specify the behavior of the `BlockingQueue` in such a way that one can conclude that data is retrieved from the queue in a particular order (*e.g.*, a FIFO queue, or retrieving the element with the highest priority first).

To specify classes such as `AtomicInteger`, we have to extend the logic and underlying semantics with volatiles, and in addition we have to specify the behavior of the native compare-and-set operation. We also discuss how the specification of atomic classes is necessary to verify implementations of lock-free data structures – using these volatile variables. We sketch in particular the properties we plan to verify for a lock-free hash table.

[1] The fact that they are commonly used is the result of a statistical analysis with the Histogram toolset [8] of a large collection of concurrent applications, found on the internet.

The work that is described in this paper is not finished yet. Instead, it presents our ideas and work in progress. Throughout the paper, we give extensive descriptions of our plans on how to continue the work; as the title says: we are only at basecamp now, and there is still much work to be done before reaching one of the peaks in the VerCors mountain region.

Overview After giving a short summary of the permission-based separation logic that is used to reason about Java in Section 2, the next two sections describe the ongoing work in VerCors. First, Section 3 describes the current state of the tool set, its architecture, the specification language, and how the annotated programs are encoded into Boogie and Chalice. Next, Section 4 describes the work on specifying commonly used concurrency classes. Then, we conclude the paper by describing related work (Section 5), and by drawing conclusions and detailing our further workplan in Section 6.

2. A Quick Summary of Separation Logic

Before discussing the first results of VerCors, this section first briefly summarizes permission-based separation logic. For more detailed information, we refer to [26–28].

Permissions π are values in the domain $(0, 1]$. At any point in time, a thread holds a collection of permissions on locations. If a thread has a full permission for a certain location, *i.e.*, the value 1, then it has permission to change this location. If a thread has a fractional permission, *i.e.*, a fraction less than 1, then it has a read permission on this location. Soundness of the logic ensures that the total number of permissions on a location never exceeds 1. Thus, at most one thread at a time can be writing a location, and whenever a thread has read permission, all other threads holding a permission on this location at the same time also can have a read permission only. Permissions can be split and combined, to change between read and write permissions. Permissions can be transferred between threads upon thread creation, and upon *joining* a terminated thread. Locks are associated with a set of permissions that can only be obtained by acquiring the lock.

Assertions in separation logic are expressed as first order logic formulas, extended with three special operators: the points-to predicate, combined with a permission, the separating conjunction (`*`) and the separating implication (or magic wand, `-*`). The syntax of formulas F is formally defined as follows:

$$lop \in \{\texttt{*}, \texttt{-*}, \texttt{\&}, \texttt{|}\} \qquad qt \in \{\texttt{ex}, \texttt{fa}\}$$
$$F ::= e \mid \texttt{PointsTo}(e.f, \pi, e) \mid F\ lop\ F \mid \texttt{(}qt\ T\ \alpha\texttt{)(}F\texttt{)}$$

In classical Hoare logic, assertions are properties over the state. In separation logic, the state is explicitly divided into the heap, where all object information is stored, and the store, containing information about the current method call. Intuitively, an assertion $\texttt{PointsTo}(e.f, \pi, v)$ (or $\texttt{x.f} \overset{\pi}{\longmapsto} v$ in traditional notation) holds for a thread t if the variable `x.f` points to a location on the heap that contains the value v, and in addition, the thread t has at least permission π on this location. A formula $\phi_1 \texttt{ * } \phi_2$ holds for a heap if the heap can be split into two *disjoint* heaps, and the first subheap satisfies ϕ_1, while the second subheap satisfies ϕ_2. A formula $\phi_1 \texttt{-*} \phi_2$ holds for any heap that has the following property: if the heap is extended with a *disjoint* heap that satisfies ϕ_1, then the combined heap satisfies ϕ_2. The separating implication is sometimes also read as a *trade* operation: the resources specified by ϕ_1 are exchanged for the resources specified by ϕ_2.

Two important characteristics of separation logic are:

- points-to assertions do not only express that a location holds a certain value, but they also express that a thread has permission to access this location; and

- whenever two expressions are separated by the separating conjunction, then implicitly they express properties about objects that are not aliased with each other.

These two characteristics are exploited to reason about concurrent Java programs. In particular, rules for field lookup and update have preconditions requiring that a thread has a permission to access or write on a location.

As mentioned above, when a new thread is created, it obtains some of the resources of the thread that creates it (and this creating thread has to give up these resources). Technically, such a specification is associated with the `run` method of the newly created thread. The resources from the new thread are separated from the resources of the creating thread, and thus the two threads can be verified in isolation. Whenever a thread terminates, other threads can `join` this thread, and obtain back resources from the thread – as specified as the postcondition of the `run` method. To ensure that no resources are created, first a special join token is created, and only threads that hold (part of) this join token can join the thread. Finally, our logic supports reentrant locks. For each lock a *resource invariant* is specified stating which resources it protects. Whenever a lock is acquired for the first time by a thread, it obtains these resources, and thus can access the data protected by the thread. Upon final release of the thread, the thread is forced to give up the specifications.

3. The VerCors Tool Set

The VerCors tool set verifies object-oriented code, in particular Java code. The input for the tool is source code that has been annotated with contracts. The output is a listing of the contracts that the tool could not prove to be correct. Each failure can optionally be accompanied by the full details provided by the underlying verification engine.

The basic technique of the tool is verification condition generation. That is, given a method with a contract and an implementation the tool will generate proof obligations in first order logic, whose validity implies that the implementation of the method satisfies the contract.

3.1 Tool Architecture

Currently, we work on supporting two input languages: Java and a toy language used for the Program Verification course taught at the University of Twente: PVL. The latter is a very simple class-based language, e.g., it has no strings and no inheritance. The specification language that is used for it is separation logic with abstract predicates, as introduced by Parkinson [46].

Rather than developing yet another verification condition generator, we have decided to reuse existing verifiers. Specifically, we have chosen to work with the existing verifiers Chalice [37] and Boogie [3]. This means that we will encode verification problems in the input language as either Chalice or Boogie programs.

The tool is built along the classical pattern of a compiler. That is, the input programs are parsed into an abstract syntax tree on which several transformations are applied before they are passed on to one of the back ends. The intermediate data structure, representing the abstract syntax tree, is called Common Object Language (COL). It encompasses the typical features of object-oriented languages, and it is set up in such a way that adding a new language is relatively straightforward.

The arrows in Fig. 1 indicate the possible paths a problem can take from input to solver. They reflect that Chalice works by translating its input into Boogie and Boogie in turn works by generating a problem for an SMT solver, such as Z3 [18]. The direct arrows from COL to Chalice and Boogie indicate that the tool will transform programs into input programs for those tools. We do not plan to generate proof obligations for an SMT solver directly.

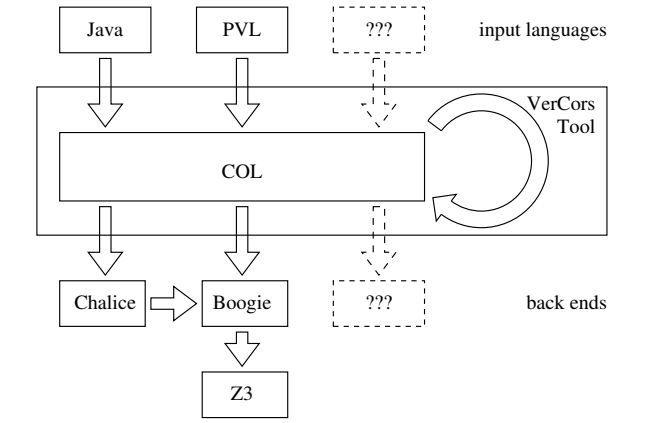

Figure 1. VerCors Tool architecture.

3.2 Supported Specification Language

An important goal of the VerCors project is to build a user-friendly tool for verifying realistic Java programs. The de facto standard for writing specifications for Java programs is JML [36]. We will follow this standard whenever possible. This will allow us to reuse much of the experience and research on specification writing, and in addition it will allow users to extend JML specifications with concurrency details, rather than having to write these specifications from scratch.

In separation logic, both the `PointsTo` predicate and the `Perm` predicate are used. The `Perm` predicate specifies fractional access to a field and is defined as follows:

$$\texttt{Perm}(\texttt{x.f}, \pi) \stackrel{\text{def}}{=} \exists v.\texttt{PointsTo}(x.f, \pi, v) \ .$$

The difference between `PointsTo` and `Perm` is that the former gives an integral specification of both access permission and value, whereas the latter specifies just the access permission. Another way of expressing the relationship between the two predicates is

$$\texttt{PointsTo}(x.f, \pi, v) \Leftrightarrow \texttt{Perm}(\texttt{x.f}, \pi) \star \texttt{x.f} = v \ . \qquad (1)$$

This equivalence has been studied by Parkinson and Summers in [47]. It allows us to explain the differences in specification styles between Chalice, VeriFast [31] and our tool. In Chalice, `PointsTo` is not supported, so the left hand side cannot be used. In VeriFast it is forbidden to write $\texttt{x.f} = v$, so the right hand side cannot be used. In contrast, our tool will support both styles of specification. To this end, we exploit the equivalence between the two specification styles, and we will extend this with translations from one style into the other.

To achieve modular verification, it is important that the specification language addresses the frame problem [40]: *when formally describing a change in a system, how do we specify what parts of the state of the system are not affected by that change?* In a sequential setting, it often suffices to specify exactly how a method modifies the state (or value) of an object. In contrast, in a concurrent setting, it is also essential to make sure that a method has appropriate permissions to access shared data. This means that one is not allowed to access a shared data structure from either code or contract unless one holds at least a read permission on it. Moreover, if one does not use the permissions to change a part of the state, this part of the state must be unchanged. As a consequence, access permission must be defined for every method.

```
public class Counter {
 int val;
 /*@ modifies val;
  *  ensures  val==\old(val)+1;
  */
 void incr(){ val=val+1; }
}
```

JML

```
public class Counter {
 int val;
 /*@ requires Perm(val,1);
  *  ensures  Perm(val,1) &&
              val==\old(val)+1; */
 void incr(){ val=val+1; }
}
```

Separation Logic, enriched syntax

```
public class Counter {
 int val;
 /*@ requires PointsTo(val,1,tmp);
  *  ensures  PointsTo(val,1,tmp+1);
  */
 void incr(){ val=val+1; }
}
```

Separation Logic, core syntax

Figure 2. Three specifications of increment.

In Fig. 2, we show three equivalent specifications of the increment method in a counter. One in JML, one in an enriched separation logic, which uses `Perm` and one in pure separation logic, which is limited to `PointsTo`. Strictly speaking, the `modifies` clause in JML does not specifiy an access permission; instead it specifies that a certain field might be modified. However, in this particular case, `modifies val;` can be read as a permission to write `val`. When considering recursive data structures and methods, deriving the access permissions that are implicitly associated to the modifies clause is less clear and a more detailed mechanism for specifiying them will be needed. This relationship will be investigated further.

To provide abstraction in the specifications, the VerCors tool set will also support abstract predicates [46]. Predicates provide the means to define properties recursively, e.g., to specify recursive data structures such as linked lists. Another advantage of abstract predicates is that it is possible to write a signature of a predicate without its definition (and later, in a subclass to add a definition). Therefore, abstract predicates provide a way of writing a complete specification of an interface without knowing any of the implementation details. Finally, it should be noted that this entire methodology also covers the requirements that inheritance imposes on contracts.

To understand how we support predicates, it is important to realise that three kinds of predicates can be distinguished:

- **value predicates** that specify properties of the values of objects only; those can only be used in the combination with permission predicates;
- **permission predicates** that specify only the permissions held on parts of the object; and
- **combined predicates** that specify both the value and permission properties at the same time.

The same distinction can be seen in equation (1): $\texttt{Perm}(\texttt{x.f}, \pi)$ is a permission statement, $\texttt{x.f} = v$ is a value statement, and $\texttt{PointsTo}(x.f, \pi, v)$ is a combined statement.

The concept of pure methods in JML is quite similar to the concept of value predicates, and therefore we intend to reuse the pure method mechanism to express value predicates. The difference is that pure methods allow arbitrary Java code, whereas the current theory of predicates only considers predicates written in a purely functional language. So in order to keep things simple, we will initially impose restrictions on the code used in pure methods, disallowing loops for example. It is future work to establish restrictions that provide a good balance between expressivity and verifiability. (In theory, everything can be translated into a functional language. However, in practice, automatic provers are unlikely to be able to handle the results of such a translation.)

Eventually, we plan to support all three kinds of predicates in our tool, but we have started with value predicates and permission predicates. Thus, we have chosen to use Chalice as a back end, rather than VeriFast [31], which works with combined predicates only. As soon as we are able to support all three kinds of predicates in the tool, we will also support VeriFast as a back end.

3.3 Encoding Verification Problems in Chalice

Chalice [37] is a verifier for concurrent programs. The input language has objects, but no inheritance or interfaces. Permissions are denoted with the `acc` predicate, which for the purpose of this paper is equivalent to the `Perm` predicate. The `PointsTo` predicate is not supported. The language has support for value predicates in the form of pure methods, called functions. It also has limited support for combined predicates using the `predicate` keyword. The restriction is that predicates cannot have arguments, which is an essential for writing predicates in pure separation logic. For example, it means that the `PointsTo` predicate cannot be defined in Chalice. Finally, it supports standard locks as opposed to the reentrant locks in Java.

The lack of inheritance is only a minor inconvenience, because the additional requirements arising from inheritance are easily added to existing contracts (cf. desugaring in JML [49]).

The lack of support for predicates with arguments is a bigger problem. We consider that it is important to be able to state that a method can perform its task on a structure, given any read permission. In separation logic this is typically specified using a predicate with a fraction as argument. Thus, we have to deal with predicates with arguments that cannot be expressed as functions because they deal with access permissions. To overcome the lack of arguments in predicates, we are developing a transformation that explicitly passes and returns permissions. That is, for every predicate in the precondition, we add an argument that encodes that the predicate holds and for every predicate in the postcondition, we return such an argument.

For example, consider a linked list, with a predicate `state(p,q)` that gives fraction `p` permission on the values in the nodes and fraction `q` on the structure of the list:

```
class List {
  int val;
  List next;
  pred state(frac p,q)
    = Perm(val,p)
    * Perm(next,q)
    * next!=null -> next.state(p,q);
}
```

We encode permissions on such a list with an explicit chain of permission objects (see Fig. 3). The class `List_state` is given in Fig. 4. This encoding is obtained as follows:

- every argument of the predicate becomes a field in the class (line 8);
- a field `about` is reserved to point to the object for which the permission is held (line 6); and

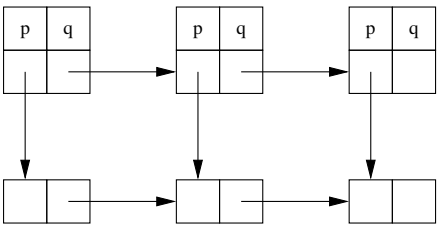

Figure 3. Example of a list with explicit permissions.

```
1  class List {
2    var val : int;
3    var next: List;
4  }
5  class List_state {
6    var about: List;
7    var about_next: List_state;
8    var p: int; var q: int;
9    predicate valid {
10     acc(about) &&
11     acc(about_next) &&
12     acc(p) && 0 < p && p <= 100 &&
13     acc(q) && 0 < q && q <= 100 &&
14     acc(about.val,p) &&
15     acc(about.next,q) &&
16     (about.next!=null ==> (
17         about_next!=null &&
18         about_next.valid &&
19         about_next.p_is(p) &&
20         about_next.q_is(q) &&
21         about_next.about_is(about.next)
22     ))
23   }
24   function p_is(frac:int):bool requires valid;
25     { unfolding valid in this.p==frac }
26   function q_is(frac:int):bool requires valid;
27     { unfolding valid in this.q==frac }
28   function about_is(l:List):bool requires valid;
29     { unfolding valid in this.about==l }
30 }
```

Figure 4. The Chalice encoding of the state predicate.

- for every field on which a recursive call is to be made, a matching permission field is added (line 7).

The predicate `state` with arguments on `List` is translated to a predicate `valid` without arguments on `List_state`. This predicate states that

- we have full access to the entire predicate object (lines 10-13);
- we have permission on the `val` and `next` fields (line 14,15); and
- we have conditional recursive access (lines 16-22).

Suppose that the class list contains a method `m` that works with any permission:

```
forall   p,q:frac;
requires state(p,q);
ensures  state(p,q);
void m(){...};
```

This method will be encoded in Chalice as follows:

```
m(pre:List_state) returns (post:List_state)
  requires pre.valid && pre.about_is(this);
  ensures post.valid && post.about_is(this);
       && post.p==pre.p && post.q==pre.q;
{...}
```

After we have finished the encoding of predicates, we will start working on the encoding of reentrant locks.

3.4 Future Work

Supporting Other Input Languages By building the tool around two input languages, we try to ensure that it remains possible to add support for other languages without major refactoring. We expect that actually adding support for other languages is beyond the scope of the VerCors project, but we have identified some languages that would be of interest to add to the VerCors tool set.

An interesting candidate is Scala [41]. To support Scala, we would need to add support for algebraic data types and pattern matching. This would also be useful to support an interactive theorem prover as a back end and to support the specification style of the VeriFast tool.

Due to the many differences between C++ [52] and Java, full support for C++ is not going to be added soon. However, we may end up supporting static dispatch and thus cover an essential part of C++. The reason is that when one is reasoning about predicates, one needs to reason about the definition of the predicate in not just the super class and the outermost subclass (similar to dynamic dispatch), but also about the definition in the current class (similar to static dispatch). Because predicates are closely related to (pure) methods, it would be convenient to treat predicates and methods in the same way. A possible way of doing this is adding support for static dispatch.

Supporting Other Back Ends We foresee that we will add the VeriFast tool as a back end. Its specification language is pure separation logic rather than the extended form we allow, but as long as the permission predicates and the value predicates follow the same recursive pattern, translating them into a combined predicate is straightforward.

We are also considering other back ends, such as a JML back end, depending on the difficulty this will give us. Currently, we believe that this might be relatively straightforward, because much of the work needed would be shared with the VeriFast back end. Moreover, the explicit permission passing transformation can also be used to encode our specifications in pure JML. This would allow us to use the different JML tools to perform all kinds of verifications (including support for run-time verification).

To satisfy the goal of verifying big chunks of the `java.util.concurrent package`, we expect that we may need to verify (initially?) certain parts with an interactive theorem prover rather than with an SMT-solver-based back end. A possibility to support this would be to add KeY [7] as a back end, but this option has not been investigated further yet.

We do not expect that we need extensive support for native methods during the VerCors project. Initially, our approach will be to manually specify their behaviour, and reason with this specification. If in a future project, we encounter more extensive native C libraries, then we might be able to support reasoning about these libraries via the VeriFast back end, or if the scope is much more extensive by using VCC for the C code.

Parallelisation of the Verification One of the things that we would like to build into the VerCors tool is the ability to exploit available computational resources better. For example, rather than passing the entire verification problem on to one of the back ends, we can split it into several chunks (say one method each) and verify

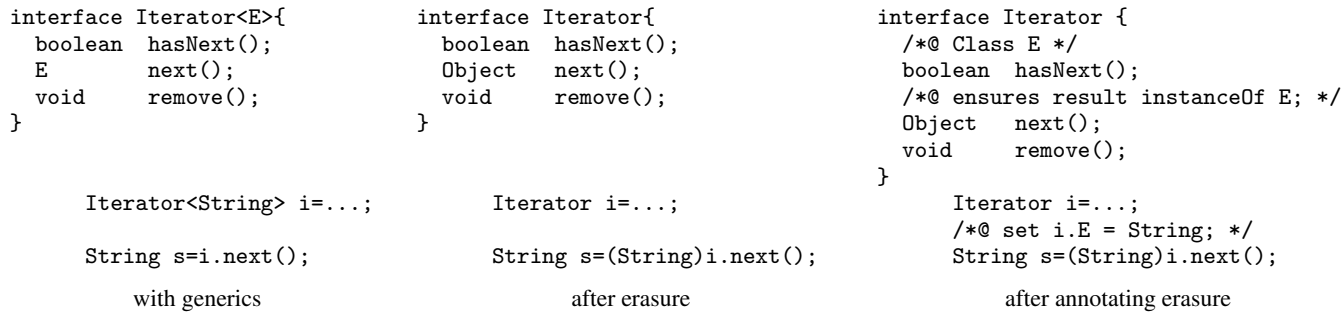

Figure 5. Two ways of treating a program with generics.

these sub-problems in parallel. This would speed up the verification task. In addition, it is also possible to apply multiple back ends to the same sub-problem. This helps because as long as for each sub-problem there exists at least one back end that can validate the problem, the overall task will have succeeded.

To enable the addition of such a feature, we structure the back end execution unit of our tool around a task queue in which tasks carry information on whether they can be split into sub-tasks and on whether there is more than one way to perform them.

Reducing the Annotation Workload Writing annotations can be very tedious. Not only is it necessary to write the contract for every method, it is also necessary to include many hints to the prover inside the code. The most common of these are the fold and unfold statements that indicate when a predicate should be replaced by its definition (and vice versa). Therefore, it is our goal to reduce the number of proof hints that must be written to a bare minimum. Again, we are thinking about reusing existing tools. Both Boogie and Chalice include several options that can automatically add annotations to simplify the process. Those will be investigated, also as part of the parallelisation effort mentioned above.

Another approach that we find interesting is the automatic inference of access permissions by Ferrara and Müller [23]. Not only would such a tool help writing method contracts, but we think that a similar constraint solving approach might be used to find out when to fold and/or unfold predicates.

Clearly, more research and more experience is needed in to find effective ways to achieve this goal.

Supporting Generics by Compilation The issue of generics is completely orthogonal to the issue of concurrency, but a practical tool has to deal with them. We are thinking about a way of dealing with generics that follows the 'verification by translation' approach that we adopted for dealing with concurrency.

Theoreticians tend to think of Java generics as a weak form of a polymorphic type system. This is however not how it is implemented in Java compilers: those use erasure instead. Effectively they remove all generics annotations, replacing the generic types by upper bounds of the possible types (often `Object`) and adding cast expressions where they are necessary. We propose to translate the generics annotations to specifications during the erasure.

For example, consider the code fragment in the left column of Fig. 5. If erasure is applied the result would be the code in the middle column. Instead we propose to translate the generics information into specifications. Whenever a generic type was introduced, this information is kept in the form of a ghost variable. Whenever the type in an argument is erased, we add a precondition. Whenever a return type is erased, we add a postcondition. The result is the code in the right column.

In order to be able to properly express the results of shifting the generics from code to specification, we will need to be able to reason about the class hierarchy in separation logic. This means adding types to the specification language, complete with a function for getting the dynamic type of a variable and rules or axioms for dealing with `instanceOf` and cast expressions.

4. Specification of Concurrent Data Structures

This section discusses the challenges of specifying concurrent data structures and possible solutions.

The first challenge is to give specifications of concurrent data structures that are as complete as possible, not just describing that the data structure implementation does not have any data races, but also how it will handle the data stored in it. The problem is that one has to ensure that a postcondition cannot be invalidated by another thread. For example, as discussed below, for a concurrent queue, one cannot simply specify that after a `put` operation the added element is the last element in the queue. This postcondition would be *unstable*, meaning that it can be invalidated by another thread, before the caller (that relies on the postcondition) can continue its execution. The solution that we discuss here for this problem is to use histories that give a serialised representation of how the data structure state has changed.

Another challenge is that specification constructs such as invariants and constraints should have a different semantics in a concurrent setting. For sequential programs, a visible state semantics is used, meaning for example that invariants have to hold at every *method border*, i.e., whenever a method is called or returns. In a concurrent setting, at any point inside a method, another thread might be at a method border for the same object, so one needs a notion of strong invariant (cf. [6]), specifying properties that have to hold at any point. We will sketch how such constructs are added to the logic, and what are the consequences for verification.

Finally, a last challenge is to consider volatile variables. These are used in lock free data structures. The Java Memory Model [39] ensures that there cannot be data races on volatile variables, but their value can be changed at any point by any other thread. When reasoning about volatile variables, the specifications have to express whether a variable can be assumed to be *stable*, i.e., whether its value can be changed by another thread or not.

This section describes our first ideas on how to develop solutions to these challenges.

4.1 Concurrent Queues

A considerable part of our recent research was focused on writing specifications for concurrent queues, in which the main target was the `BlockingQueue` interface and the classes that implement this interface from the `java.util.concurrent` package (Fig. 6). We

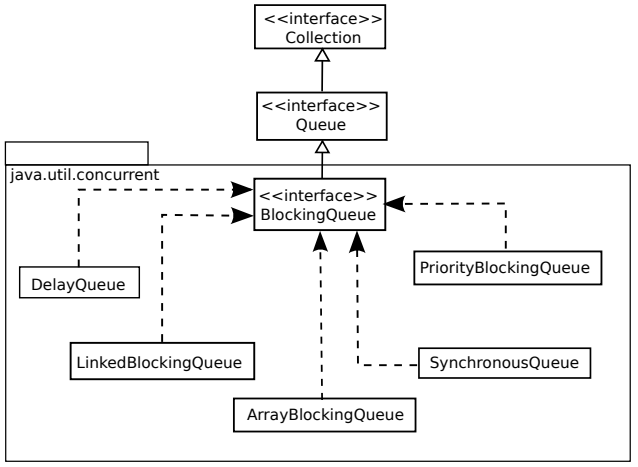

Figure 6. BlockingQueue class hierarchy

present an abstract solution for specifying the `BlockingQueue` interface and we show how this specification can be inherited by the descending classes even though they use different *orderings* to store the elements in the queue.

Thread interference leads to difficulties in defining the contracts for even the most trivial methods. Consider the method `put(E e)` from the `LinkedBlockingQueue` class.

```
public void put(E e) throws InterruptedException {
  . . .
  putLock.lockInterruptibly();
  try {
   . . . enqueue(e);
  } finally {
    putLock.unlock();
  } . . .
}
```

The method enqueues the parameter `e` at the end of the queue. Nevertheless, between the moment of the lock release and the end of the method, another thread may interfere and cause unpredictable changes to the queue, even possibly removing the newly added element `e`. This non-deterministic behavior makes it difficult to specify the functionality of the method.

We propose an approach based on logging data in a history. Two variables are defined in the `BlockingQueue` interface specification:

- `historyList` - a ghost variable that keeps track of every event that modifies the queue; and
- `actualQueue` - a model variable that is an abstract representation of the queue data structure.

Both variables are of type JMLValueSequence (a model class from the package `org.jmlspecs.model` defined as an immutable sequence of values). The `historyList` node element is a wrapper of the queue element and an additional boolean flag `exists` that indicates whether the element exists in the queue or not. The classes that implement the interface are responsible for updating the `historyList` and logging the new events, which is supported by set annotations on `historyList`. The moment when an element `e` is added to the queue, the `historyList` is extended with a new element (a wrapper of `e` with the flag `exists=true`). Removing an element from the queue is followed by setting the flag of the appropriate element in the `historyList` to false. These updates to `historyList` must be done before the lock is released, to guarantee that no other thread will interfere.

We define the contracts of the methods of the `BlockingQueue` interfaces in terms of the `historyList`. For example, for the `put(E e)` method, the postcondition expresses that the `historyList` contains the wrapper of the element `e`. However, we cannot state anything about the value of the `exists` flag, since it holds only in case that the element has not been removed after the lock release.

This pattern can be understood as a background process that observes the queue behavior and reacts appropriately by logging each action into the history. While the `actualQueue` is a shared data structure accessible by a number of threads, the `historyList` collects all events executed by different threads and serializes them in a unique sequence. Since the result of the parallel and sequentially executed code should be the same, both lists (the `historyList` and the `actualQueue`) should be *compatible* (see Fig. 7). This property guarantees that the `historyList` elements for which the `exists` flag holds, match the elements from the `actualQueue` and obey the same ordering rule. We define the compatibility property through a recursively defined predicate `compatibleOrdering (JMLValueSequence actualQueue, JMLValueSequence historyList)`. Further, the specification contains a (strong) class invariant expressing that this compatibility property always has to hold, In addition, the method's postconditions guarantee that the `historyList` is properly updated, thus our specification abstractly describes the behavior of the `actualQueue` too.

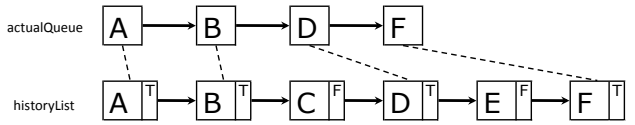

Figure 7. Compatible historyList and actualQueue

Ordering The `Queue` data structure normally, but not always, follows the FIFO ordering rule. Which rule is followed, is unknown in the `BlockingQueue` interface; it is decided by the implementing classes. In order to preserve the compatibility property, the `historyList` should follow the same ordering rule (FIFO, LIFO, etc.) as the `actualQueue`.

However for some orderings, specifying how the `historyList` should be updated might become too complicated. For example, the `PriorityBlockingQueue` class contains Comparable elements that are added in such a way that the queue remains sorted. In order to avoid the problem of writing complex `historyList` assignments, we extend the specification of the interface with an additional model boolean variable `ordered`, and a new predicate `compatibleExistence(JMLValueSequence actualQueue, JMLValueSequence historyList)`. Its purpose is to give an opportunity to the one who specifies the descending class to choose between two options: whether to update the `historyList` while obeying the same ordering rule as the `actualQueue` defines, or to add the elements in the `historyList` in an arbitrary order (typically FIFO order). If the second option is chosen, the class invariant will check the correctness of the `compatibleExistence` predicate that only guarantees that the `historyList` and the `actualQueue` are compatible regarding the existence of the elements, while the elements order is not considered. This choice is made by choosing a concrete value for the `ordered` variable in the implementation.

For instance, in the specification of the `LinkedBlockingQueue` class, the property `ordered` is set to `true`, thus the `historyList` will respect the FIFO ordering (defined by the `LinkedBlockingQueue` class), and the invariant expresses that this corresponds to

the ordering of the elements in the actual queue. In the `PriorityBlockingQueue` we set `ordered = false`, thus we do not have to obey the ordering rule when adding elements to the `historyList`. Instead, we agree that only the existence of the elements, without the ordering, will be guaranteed. As a side remark, in case of the `PriorityBlockingQueue` class, which is based on a heap data structure, the order of the elements can be easily expressed by adding an additional invariant that guarantees the correctness of the *heap rule*, i.e., the value of every parent node is smaller than that of both child nodes.

Summary of the History-Based Specification Approach We summarize the ingredients of our approach in Table 1. It gives a general overview of the specifications in the BlockingQueue interface and in the descending classes. The table illustrates the abstractness of our approach: most of the specifications are defined in the interface, and only a small effort has to be done to specify a descending class.

BlockingQueue	**Descending class**
Define a ghost variable `historyList`	Update the `historyList`
Define a model variable `actualQueue`	Define a concrete representation for the `actualQueue`
Define a model variable `ordered`	Define a concrete representation for `ordered`
Define a predicate `compatibleOrdering`	
Define a predicate `compatibleExistance`	
Define a class invariant that checks the correctness of the proper predicate	
Define methods contracts	

Table 1. Specifications overview

An important question is how a client class of the `BlockingQueue` interface can take advantage of the history-based specifications. It is currently too early to give a full answer to this question. However, we believe it is important that the proposed specifications allow one to prove *meta* properties about the queue, such as that the `BlockingQueue` object has a regular structure, holds the correct elements, and respects a certain ordering rule. The history has public visibility, thus it can be freely used outside the `BlockingQueue` class. This gives clients the possibility to reuse the history specifications and to reason about other desired properties regarding the current queue state as well as its state during the history. We intent to investigate more in this direction and to analyze the clients needs in different realistic applications, in order to make our approach also useful for client classes.

Finally, it should be remarked that currently our approach provides a poor mechanism for updating the `historyList`. It requires ghost variables assignments scattered through the code, which may be error-prone. Therefore, as future work, we plan to study how we can systematically generate these assignments.

The specifications that we provide are still not integrated in the tool that we develop, but this is planned in the next stage of the VerCors project. Nevertheless, we believe that these ideas are important ingredients that will guide the tool development to the right directions. We also plan to extend this work by specifying other concurrent data structures, for which we believe that the same pattern can be reused.

4.2 Lock-free Data Structures

Lock-free data structures form an efficient approach to parallel computing. Atomic operations like Compare-and-Swap (CAS) are the main elements of these data structures. First we look at `atomic` classes for volatile variables and then we explain our study of more complicated lock-free data structures.

Atomic Classes Java provides the `java.util.concurrent.atomic` package to support lock-free and thread-safe programming on single variables. It enables those variables to be manipulated atomically. The `atomic` package contains a set of classes that essentially provide a wrapper for `volatile` variables with appropriate atomic operations. In Java, when a variable is marked as `volatile`, the Java Memory Model guarantees that always the last written value to the variable is visible to all threads (volatile read and write actions are *synchronized actions*, just as lock, and unlock).

To understand how we can reason about volatile variables, we first need to briefly discusses the Java memory model (JMM) [39]. The JMM uses a *happens-before* relation to order memory events. If an event e_i happens-before e_j, then e_i is visible to and ordered before e_j. Two accesses to the same variable are called *conflicting* if at least one of them is a write access. If a given program contains two conflicting accesses not ordered by the happens-before relation, then the program contains a *data race*. A program is called *data race free* if in each sequentially consistent execution of the program, there is a happens-before relation between each pair of conflicting actions. To rephrase the semantics of volatile variables in terms of the JMM, we say that a write to a volatile field happens-before every subsequent read of that field. Therefore, the JMM allows volatile variables to be accessed (reading or writing) concurrently without mutual exclusion.

Most Java semantics do not have special treatment of volatile variables. To the best of our knowledge, only Boyland [12] describes an operational semantics for a Java-like language including volatile variables. Boyland proposes the notion of *write-key* to distinguish between a volatile and a normal variable read/write. For volatile variables all threads are sharing the same key knowledge. Each thread writing to a volatile variable extends the key knowledge with its write-keys, thus making the write visible to all other threads. Any thread updating a normal variable extends its local key knowledge. This ensures that a parallel thread does not read this value, because it does not have the new write-key in its local knowledge. Sharing a global write-key set permits concurrent actions on volatile variables, while inconsistencies between the local key sets model conflicting actions on normal variables.

As mentioned above, within the VerCors project, we use a logic that is based on fractional permissions. In particular, any thread that does not have a full permission is not allowed to update a normal variable. We are working on extending the logic with a distinction between volatiles and non-volatiles variables. The main idea that we are developing is that a volatile variable can be updated with any non-zero permission, but only if a thread has a full permission, it can rely upon the value in the volatile variable not being changed by another thread. We will investigate if we can use the write-key technique to describe the semantics of the volatile variables, and then prove our logic for volatiles sound w.r.t. this semantics.

An important aspect of this work is to write the specifications for atomic variables in the `atomic` package. All the atomic variable classes in this package provide a *compareAndSet* primitive, which is implemented using the fastest native construct available on the platform. We are currently working on a specification for the `AtomicInteger` class that implements all the atomic operations for an integer variable.

In order to keep track of the values written to the volatile field of an `AtomicInteger`, we are using a value history list, as proposed above for the concurrent queue specifications. However, a history list in this class does not contain the boolean existence flag. Instead, we specify that the current value of the field in the class must be the same as the last value in the history list. Again, this is expressed as a class invariant. To ensure that this class invariant is never invalidated, we have to ensure that the history list is updated as one atomic action with the write operation. Therefore, we propose an extension to the specification language: a specificataion instruction

atomic(L) that specifies an operation on a ghost variable that must be done in one atomic action with the program statement labeled L. The following code illustrates this for the specification written for the method `AtomicInteger::set(int newValue)`.

```
public class AtomicInteger extends ...{
 //@ invariant value == vList.getLast();
 private volatile int value;
 ...
 /*@ requires PointsTo(this.value,p,_) && 0<p<=1;
 @   ensures PointsTo(this.value,p,_) && 0<p<=1
 @           && vList.contains(newValue);
 @*/
 public final void set(int newValue){
   //@ set atomic(L) vList.add(newValue);
   L: value = newValue;
 }
}
```

Since the methods of the atomic classes are calling native methods exported in `sun.misc.unsafe`, writing the specifications for `AtomicInteger` also forces us to write specification for those native methods. These are mostly low level operations on Java object fields.

Verification of Lock-free Data Structures The specifications of the basic atomic classes will provide us the building blocks to specify and reason about other lock-free data structures, developed in Java, using these atomic classes. Currently, as a first case study, we are investigating how we can verify the correctness of a (Java version) of a lock-free hash table [35]. This hash table is originally designed in C as part of the LTSmin tool set [9]. It is intended to be used for state space exploration in multi-core model checkers. The data structure acts as a shared state storage to store visited states. Thus a state is either in the storage or it has to be added. Figure 8 shows the algorithm for state space exploration using a closed set V as a shared state storage.

Figure 9 presents the algorithm for `find-or-put`. A bucket is always either empty (E) or it contains a pair of a memoized hash code and the write status for the data. The algorithm probes the cache line to find the data.

- If the corresponding bucket is empty (E), an atomic write tries to insert the data (writing status is indicated with W). It then updates the bucket status (D indicates DONE), and returns `false` to indicate the absence of the data.
- If the related bucket is not empty, the data is either available (visited state, indicated as D) or being written (by another thread). In the second case, the thread waits for the data to be written completely. In both cases, the method returns `true`, indicating that the data has been found.

The crucial property for the correctness of this data structure is: *whenever a write started for a hash value, the state of the bucket can never become empty again, nor can it be used for any other hash value. So the written value can never change.* We are investigating how we can formally specify and verify this property.

```
T={S0};  V={ };
while(state==T.get()){
  for(succ in next_state(state))
      if( V.find_or_put(succ) )
         T.put(succ);
}
```

Figure 8. State space exploration

```
find_or_put(v){
   h = hash(v);
   ...
   if( Bucket[i] == E ) {
     if( CAS(Bucket[i], E, <h,W>) ) {
       Data[i] = v;
       Bucket[i] = <h,D>;
       return false;
     }
   }
   if( Bucket[i] == <h,_> ) {
     while( Bucket[i] == <_,W> ) do;
     if( Data[i] == v )
         return true;
   }
   ...
}
```

Figure 9. Lock-free find-or-put algorithm

As mentioned above, in a concurrent setting, the visible state semantics for invariants no longer applies, because multiple threads can execute code to update shared variables simultaneously. Instead, we have to rely upon *strong invariants* (cf. [6]), specifying properties that have to hold in all execution states. For the hash table, we can specify that the bucket status is always one of the values from {E,W,D}. Formally, this is expressed as the following strong class invariant for the hash table (assuming we have a bucket field b, where for simplicity we omit the hash value):

$$\texttt{this}.b \xmapsto{\pi} E \vee \texttt{this}.b \xmapsto{\pi} W \vee \texttt{this}.b \xmapsto{\pi} D \ .$$

Moreover, we also have to express that the state transitions of the hash table buckets are one-way transitions (from E to W and from W to D). In a sequential setting, constraints relate two consecutive visible states in an execution. But again, in a concurrent setting, we need to specify that the one-way transition property holds in all the execution steps of the method. Therefore, we use the notion of a *strong constraint*. A strong constraint expresses a relation between every two consecutive execution states. The following predicate defines the strong constraints for the hash table, using $\backslash pre$ to indicate the pre-state of the $bucket[i]$, denoted as b_i:

$$\begin{array}{l}(\backslash pre(\texttt{this}.b_i) = E \implies \texttt{this}.b_i = E \vee \texttt{this}.b_i = W) \wedge \\ (\backslash pre(\texttt{this}.b_i) = W \implies \texttt{this}.b_i = W \vee \texttt{this}.b_i = D) \wedge \\ (\backslash pre(\texttt{this}.b_i) = D \implies \texttt{this}.b_i = D)\end{array}$$

As mentioned above, originally, this lock-free hash table has been implemented in C. The Java version of this hash table implementation uses class `AtomicLongArray` for the `Bucket`. Once we have defined the semantics and verification techniques for volatile variables and the specifications of the classes in the `atomic` package, the Java version of the concurrent hash table will be verified.

It should be noted that so far, we have only been looking at safety properties of the hash table. As future work, it might be worth to investigate how one can also prove progress of the hash table algorithm. For this we might be able to reuse the work of Leino and Müller on deadlock-free channels [38], the work of Brotherston *et al.* on proving termination using a combination of separation logic and cyclic proofs [13], or the work of Gotsman *et al.* on proving liveness properties of non-blocking algorithms [25]. However, this topic is not on the agenda in the near future.

In the literature, several other verifications of lock-free data structures have been described (see Section 5). We are investigating if we can reuse some of this work to verify our target lock-free hash table. However, because of our focus on real Java, and our specification language that is inspired by JML, these earlier results cannot be reused directly.

5. Related Work

Tools Several other teams develop theory and tools to reason about concurrent programs. Closely related to our approach is the work on Chalice [37]. As mentioned above, our tool builds on this: an annotated Java program is encoded into Chalice. However, the language that Chalice can reason about is more limited in scope than general-purpose Java programs, and it is not clear if we can encode everything in Chalice, or whether we will need to use a direct translation into Boogie.

Appel *et al.* have developed a separation logic for a subset of C (see [2] for an overview). Their work is completely formalized within the Coq theorem prover, including the program semantics, a compiler formalization and soundness proofs of the logic. Their focus is more on having a completely formalized tool chain, whereas our focus is on having a practically usable tool set that can reason about realistic programs.

VeriFast is a separation-logic based tool to reason about concurrent C and Java programs [31]. VeriFast requires a user to write many of the intermediate assertions explicitly, which makes it mainly usable for expert users. One of the topics that we will work on within the VerCors project is the generation of annotations, so that hopefully we can provide a higher level of automation. As mentioned above, we would like to provide support to have VeriFast as a back end. This would mean that we would have to encode our specification language with JML features into the pure separation logic of VeriFast.

The Spec# programming system can verify C# [4]. It uses a permission model that organises access permissions as a forest of trees. This is similar in expressive power to what can be done with predicates and separation logic, but the details of how the permissions are managed is quite different. The same permission model is used in the VCC verifier for C code [14]. More than our own tool, the VCC verifier is aimed at large scale verification of existing code. To support the verification effort, it also encompasses several tools that provide insight in the reason why verification failed.

The KIV system is a dynamic logic-based program verifier. It has a rely-guarantee-based extension with temporal logic, to reason in a local way about concurrent programs [51].

Klebanov has proposed an extension of dynamic logic in the Key system, to reason about the behavior of programs with data races, respecting the Java Memory Model [34]. In contrast, our intention is to reason only about programs that are sequentially consistent, either because they have no data races, or because their data races can be considered *benign*.

Further, there are several tools that allow to reason about sequential Java programs. We mention in particular the separation-logic-based jStar tool [21], ESC/Java [15] and OpenJML (see `sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml`. The latter two use JML as a property specification language. We intend to combine separation logic with features of JML, and where appropriate we will reuse the verification approach implemented in these tools.

Logic Parkinson *et al.* [48] verified a non-blocking stack algorithm using separation logic. Vafeiadis and Parkinson propose a combination of rely/guarantee reasoning and separation logic, named RGSep [53], to tame the complexity of the verification of concurrent algorithms. Bornat and Amjad [10] employed RGSep to prove correctness of two inter-process buffers algorithms.

Inspired by RGSep, Dinsdale-Young *et al.* propose a compositional technique using *concurrent abstract predicates* (CAP) [20]. A given client program using a concurrent module is verified against the module's abstract predicates. This allows the client program to use different versions of the module as long as the predicates hold and thus the client's verification remains valid. Predicates are defined using an assertion language in which both states (thread-local and shared) and interferences (as actions) are specified.

As an alternative to CAP, Jacobs and Piessens propose a procedure-modular technique to specify and verify fine-grained concurrent programs [32]. The proposed solution is based on passing ghost actions as procedure arguments. Then the underlying code is able to instantiate the required operation on ghost variables. Compared to CAP, in which the designer defines the restrictions of using the module in form of abstract predicates, in this technique the client program manages the restrictions.

Abraham *et al.* [1] develop an assertional proof system for a multithreaded Java-like language. The behavior of a single instance is specified using local assertions and the connection between objects is expressed by global assertions. Histories of the local updates and the communication are recorded in separate auxiliary variables. Given an annotated program, a tool called Verger, generates verification conditions and then PVS is employed to prove the generated verification conditions. The focus of the work is on developing a sound and complete program logic for Java, not on having a practical verification tool. In particular, it is based on Owicki-Gries reasoning, which makes the number of proof obligations easily explode.

To control the complexity of correctness proofs for concurrent programs, de Boer *et al.* [16] propose a concurrency model where concurrent objects are extended with futures. They develop a language and proof system for distributed concurrent objects. This work might be of interest when verifying distributed programs, however it cannot be mapped directly onto Java's concurrency model.

Concurrent Data Structures and Synchronisation Techniques With respect to specifications of concurrent data structures and synchronization techniques, we are not aware of much work in this direction. Recently, Aquinas and Hobor presented a separation logic-based specification of a barrier [30]. However, they did not verify whether for example the Java reference implementation of a barrier respected their specification, and they just looked at this particular case. Our intention is to identify commonalities in specifications, and to verify whether the Java reference implementations respect our specifications.

Dovland *et al.* [22] use histories to model the behavior of components in a distributed environment. However, their histories keep track of communication events, i.e., a *communication history*, represented as a sequence of messages denoting events as object creation, method invocation or method completion. A prototype tool-support for run-time checking of JML annotated code extended with communication histories has been developed by De Boer *et al.* [17]. In contrast to this event-based history, our history is data-based, which allows one to reason about the data elements that are logged in it.

6. Conclusion

This paper gives an overview of the VerCors project. As the title mentions: we are only at base camp now, and there is still a long

way to go to reach the peaks in the VerCors area[2]. However, we have taken the first steps, and with every step we get a better idea of the road that is ahead of us.

The goal of the VerCors project is to develop techniques to reason about concurrent programs and concurrent data structures in particular. Permission-based separation logic for Java [26–28] is the basis of our work, as it has proven to be an adequate approach to keep verification of concurrent programs tractable.

Our intention is to express not just that a data structure implementation is free of data races, but also its functional properties. Therefore, our specification language is not purely separation logic-based, but we intend to leverage work on existing specification languages such as JML. As explained above, we use a history-based approach to express functional properties of data structures: the specifications maintain a history that serialises the behavior of the data structure, and that at any point can be related to the actual contents of the data structure. The serialised history can then be used to establish properties such as the preservation of the order in which elements are stored.

A special focus point are so-called lock-free algorithms, using volatile variables. We are currently working on extending the logic to reason about such variables, and we are exploring how we can use this to prove correctness of a lock-free hash table implementation.

We consider that all techniques should be practically usable. This means in particular that they should all be supported by the VerCors tool set. The tool set focuses on reasoning about Java programs (intending to be as complete as possible), but is set up in such a way that, given an appropriate variant of the logic, it can easily be extended to another programming language. Its internal design leverages the use of existing tools to reason about concurrent programs, in particular using Chalice and Boogie. The VerCors tool is intended to be usable for an (experienced) Java programmer, therefore particular attention will be given to support annotation generation and to automate the program verification process.

To reach these goals, we have identified the following main steps.

- An initial version of the tool set should be available soon, which can automatically reason about absence of data races in programs with fork/join parallelism and reentrant locks. To see whether this goal is reached, the tool will be applied on all examples in the papers introducing permission-based separation logic for Java [26–28]. As mentioned above, the tool set is set up to be easily extendable, both for input languages and back ends. The plan is to take advantage of this later; for example by choosing the most appropriate back end for each different verification tasks. However, to make sure there is a complete working version of the tool soon, we have chosen to use only Chalice and Boogie as back ends initially.
- The next step will be to extend the tool, so that it can parse and create proof obligations for the history-based specifications of the queue hierarchy. We expect that not all proof obligations will be proven automatically, so the next step will then be to understand how we can help the prover to do this. Various options for this will be explored, in particular generating annotations to get smaller proof obligations, or developing dedicated decision procedures. This approach will then be validated by specifying and verifying other classes from the concurrency library.
- A similar approach will be taken to extend the tool set to reason about lock free algorithms. We expect this will take longer, because we have also have to investigate what is the appropriate logic for this case. As a test case, we plan to verify several (Java versions) of lock-free data structures and algorithms developed for multi-core model checking.
- Once we know how to verify data structures in isolation, we plan to also verify a complete application, using the specified and verified data structures. As it will take some time before we have reached this state, we do not have a concrete application in mind yet.

What is important to realise is that our intention is not to build a new verification tool from scratch. Instead, we intend to build on the wide range of tools that are available and to tune them to our needs, so that they can be used to verify non-trivial properties of multithreaded Java programs. The major effort in tool development will thus be on encoding the specifications and programs into the input languages of the different verification tools around.

Finally, in the long run, we would like to also address more open questions such as: can we separate the verification of the locking policy from the functional specification; can we identify classes of data races (of non-volatile variables) where we can still reason about a program's behavior; and how can we adapt the logic to other synchronization and concurrency primitives? However, we do not have any concrete ideas on these issues yet.

[2] On `http://parc-du-vercors.fr` one can see why those peaks are worth a visit.

Acknowledgments

This work was supported by ERC grant 258405 for the VerCors project (Amighi, Huisman, and Zaharieva-Stojanovski), and Artemis grant 2008-100039 for the CHARTER project (Blom).

References

[1] E. Ábrahám, F. S. de Boer, W. P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theoretical Computer Science*, 331(2-3):251–290, 2005.

[2] A. Appel. Verified software toolchain. In *European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 2011.

[3] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In Barthe et al. [5], pages 151–171.

[5] G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors. *Proceedings, Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'04) Workshop*, volume 3362 of *Lecture Notes in Computer Science*, 2005. Springer-Verlag.

[6] B. Beckert and W. Mostowski. A program logic for handling Java Card's transaction mechanism. In *Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Computer Science*, pages 246–260. Springer-Verlag, 2003.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334 in Lecture Notes in Computer Science. Springer-Verlag, 2007.

[8] S. Blom and J. R. Kiniry. The histogram tool set, 2011. URL `http://fmt.ewi.utwente.nl/redmine/projects/vercors-charter-histogram/wiki`.

[9] S. Blom, J. van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359. Springer-Verlag, 2010.

[10] R. Bornat and H. Amjad. Inter-process buffers in separation logic with rely-guarantee. *Formal Asp. Comput.*, 22(6):735–772, 2010.

[11] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72. Springer-Verlag, 2003.

[12] J. Boyland. An operational semantics including "volatile" for safe concurrency. *Journal of Object Technology*, 8(4):33–51, 2009.

[13] J. Brotherston, R. Bornat, and C. Calcagno. Cyclic proofs of program termination in separation logic. In G. C. Necula and P. Wadler, editors, *POPL*, pages 101–112. ACM, 2008. ISBN 978-1-59593-689-9.

[14] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer-Verlag, 2009.

[15] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In Barthe et al. [5], pages 108–128.

[16] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

[17] F. S. de Boer, S. de Gouw, and J. Vinju. Prototyping a tool environment for run-time assertion checking in JML with communication histories. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 6:1–6:7. ACM, 2010.

[18] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.

[19] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[20] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In T. D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer-Verlag, 2010.

[21] D. DiStefano and M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 213–226. ACM Press, 2008.

[22] J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. *Electronic Notes in Theoretical Computer Science*, 203:19–34, 2008.

[23] P. Ferrara and P. Müller. Automatic inference of access permissions. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, Lecture Notes in Computer Science. Springer-Verlag, 2012.

[24] R. W. Floyd. Assigning meanings to programs. *Proc. Symp. Appl. Math*, 19:19–31, 1967.

[25] A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In Z. Shao and B. C. Pierce, editors, *POPL*, pages 16–28. ACM, 2009. ISBN 978-1-60558-379-2.

[26] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, 2008.

[27] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2008.

[28] C. Haack, M. Huisman, and C. Hurlin. Permission-based separation logic for Java, 201x. Submitted.

[29] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. ISSN 0001-0782.

[30] A. Hobor and C. Gherghina. Barriers in concurrent separation logic. In *20th European Symposium of Programming (ESOP 2011)*, Lecture Notes in Computer Science, pages 276–296. Springer-Verlag, 2011.

[31] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW520, Katholieke Universiteit Leuven, 2008.

[32] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In T. Ball and M. Sagiv, editors, *POPL*, pages 271–282. ACM, 2011. ISBN 978-1-4503-0490-0.

[33] C. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. ISSN 0164-0925.

[34] V. Klebanov. *Extending the Reach and Power of Deductive Program Verification*. PhD thesis, Department of Computer Science, Universität Koblenz-Landau, Germany, 2009.

[35] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In R. Bloem and N. Sharygina, editors, *FMCAD*, pages 247–255. IEEE, 2010.

[36] 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. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[37] K. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Lecture notes of FOSAD*, volume 5705 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[38] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-free channels and locks. In A. D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, 2010.

[39] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Principles of Programming Languages*, pages 378–391, 2005.

[40] J. Mccarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

[41] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[42] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[43] P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In L. Fribourg, editor, *Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001. Invited paper.

[44] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Principles of Programming Languages*, pages 268–280. ACM Press, 2004.

[45] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.

[46] M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In J. Palsberg and M. Abadi, editors, *POPL*, pages 247–258. ACM, 2005.

[47] M. J. Parkinson and A. J. Summers. The relationship between separation logic and implicit dynamic frames. In G. Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 439–458. Springer-Verlag, 2011.

[48] M. J. Parkinson, R. Bornat, and P. W. O'Hearn. Modular verification of a non-blocking stack. In M. Hofmann and M. Felleisen, editors, *POPL*, pages 297–302. ACM, 2007. ISBN 1-59593-575-4.

[49] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005.

[50] W. Reif, G. Schellhorn, K. Stenzel, and M. Balser. Structured specifications and interactive proofs with KIV. In *Automated Deducation - A Basis for Applications*. Kluwer Academic Publishing, 1998.

[51] G. Schellhorn, B. Tofan, G. Ernst, and W. Reif. Interleaved programs and rely-guarantee reasoning with ITL. In *Proc. of International Symposium on Temporal Representation and Reasoning in AI (TIME)*. IEEE Press, 2011. To appear.

[52] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 2004.

[53] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In L. Caires and V. T. Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer-Verlag, 2007.