An Integrated Specification and Verification Technique for Highly Concurrent Data Structures *

Parosh Aziz Abdulla[1], Frédéric Haziza[1], Lukáš Holík[1 2], Bengt Jonsson[1], and Ahmed Rezine[3]

[1] Uppsala University, Sweden
[2] Brno University of Technology, Czech Republic
[3] Linköping University, Sweden.

Abstract. We present a technique for automatically verifying safety properties of concurrent programs, in particular programs which rely on subtle dependencies of local states of different threads, such as lock-free implementations of stacks and queues in an environment without garbage collection. Our technique addresses the joint challenges of infinite-state specifications, an unbounded number of threads, and an unbounded heap managed by explicit memory allocation. Our technique builds on the automata-theoretic approach to model checking, in which a specification is given by an automaton that observes the execution of a program and accepts executions that violate the intended specification. We extend this approach by allowing specifications to be given by a class of infinite-state automata. We show how such automata can be used to specify queues, stacks, and other data structures, by extending a data-independence argument. For verification, we develop a shape analysis, which tracks correlations between pairs of threads, and a novel abstraction to make the analysis practical. We have implemented our method and used it to verify programs, some of which have not been verified by any other automatic method before.

1 Introduction

In this paper, we consider one of the most difficult current challenges in software verification, namely to automate its application to algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state. Such algorithms are of central importance in concurrent programs. They are widely used in libraries, such as the Intel Threading Building Blocks or the java.util.concurrent package, to provide efficient concurrent realizations of simple interface abstractions. They are notoriously difficult to get correct and verify, since they often employ fine-grained synchronization and avoid locking wherever possible. A number of bugs in published algorithms have been reported [10, 19]. It is therefore important to develop efficient techniques for verifying conformance to simple abstract specifications of overall functionality, a concurrent implementation of a common data type abstraction, such as a

* supported in part by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the Czech Science Foundation (project P103/10/0306), and the CENIIT research organization (project 12.04).

queue, should be verified to conform to a simple abstract specification of a (sequential) queue.

We present an integrated technique for specifying and automatically verifying that a concurrent program conforms to an abstract specification of its functionality. Our starting point is the automata-theoretic approach to model checking [30], in which programs are specified by automata that accept precisely those executions that violate the intended specification, and verified by showing that these automata never accept when they are composed with the program. This approach is one of the most successful approaches to automated verification of finite-state programs, but is still insufficiently developed for infinite-state programs. In order to use this approach for our purposes, we must address a number of challenges.

1. The abstract specification is infinite-state, because the implemented data structure may contain an unbounded number of data values from an infinite domain.
2. The program is infinite-state in several dimensions: it (i) consists of an unbounded number of concurrent threads, (ii) uses unbounded dynamically allocated memory, and (iii) the domain of data values is unbounded.
3. The program does not rely on automatic garbage collection, but manages memory explicitly. This requires additional mechanisms to avoid the ABA problem, i.e., that a thread mistakenly confuses an outdated pointer with a valid one.

Each of these challenges requires a significant advancement over current specification and verification techniques.

We cope with challenge 1 by combining two ideas. First, we present a novel technique for specifying programs by a class of automata, called *observers*. They extend automata, as used by [30], by being parameterized on a finite set of variables that assume values from an unbounded domain. This allows to specify properties that should hold for an infinite number of data values. In order to use our observers to specify queues, stacks, etc., where one must "count" the number of copies of a data value that have been inserted but not removed, we must extend the power of observers by a second idea. This is a data independence argument, adapted from Wolper [34], which implies that it is sufficient to consider executions in which any data value is inserted at most once. This allows us to succinctly specify data structures such as queues and stacks, using observers with typically less than 3 variables.

To cope with challenge 2(i), we would like to adapt the successful thread-modular approach [4], which verifies a concurrent program by generating an invariant that correlates the global state with the local state of an arbitrary thread. However, to cope with challenge 3, the generated invariant must be able to express that *at most* one thread accesses some cell on the global heap. Since this cannot be expressed in the thread-modular approach, we therefore extend it to generate invariants that correlate the global state with the local states of an arbitrary *pair* of threads.

To cope with challenge 2(ii) we need to use shape analysis. We adapt a variant of the transitive closure logic by Bingham and Rakamarić [5] for reasoning about heaps with single selectors, to our framework. This formalism tracks reachability properties between pairs of pointer variables, and we adapt it to our analysis, in which pairs of threads are correlated. On top of this, we have developed a novel optimization, based on the observation that it suffices to track the possible relations between each pair of

pointer variables separately, analogously to the use of DBMs used in reasoning about timed automata [9]. Finally, we cope with challenge 2(iii) by first observing that data values are compared only by equalities or inequalities, and then employing suitable standard abstractions on the concerned data domains.

We have implemented our technique, and applied it to specify and automatically verify that a number of concurrent programs are linearizable implementation of stacks and queues [16]. This shows that our new contributions result in an integrated technique that addresses the challenges 1 – 3, and can fully automatically verify a range of concurrent implementations of common data structures. In particular, our approach advances the power of automated verification in the following ways.

- We present a direct approach for verifying that a concurrent program is a linearizable implementation of, e.g., a queue, which consists in checking a few small properties of the algorithm, and is thus suitable for automated verification. Previous approaches typically verified linearizability separately from conformance to a simple abstraction, most often using simulation-based arguments, which are harder to automate than simple property-checking.
- We can automatically verify concurrent programs that use explicit memory management. This was previously beyond the reach of automatic methods.

In addition, on examples that have been verified automatically by previous approaches, our implementation is in many cases significantly faster.

Overview We give an overview of how our technique can be used to show that a concurrent program is a linearizable implementation of a data structure. As described in Section 2, we consider concurrent programs consisting of an arbitrary number of sequential threads that access shared global variables and a shared heap using a finite set of methods. Linearizability provides the illusion that each method invocation takes effect instantaneously at some point (called the linearization point) between method invocation and return [16]. In Section 3, we show how to specify this correctness condition by first instrumenting each method to generate a so-called abstract event whenever a linearization point is passed. We also introduce *observers*, and show how to use them for specifying properties of sequences of abstract events. In Section 4, we introduce the data independence argument that allows observers to specify queues, stacks, and other unbounded data structures. In Section 5, we describe our analysis for checking that the cross-product of the program and the observer cannot reach an accepting location of the observer. The analysis is based on a shape analysis, which generates an invariant that correlates the global state with the local states of an arbitrary pair of threads. We also introduce our optimization which tracks the possible relations between each pair of pointer variables separately. We report on experimental results in Section 6. Section 7 contains conclusions and directions for future work.

Related work. Much previous work on verification of concurrent programs has concerned the detection of generic concurrency problems, such as race conditions, atomicity violations, or deadlocks [14, 22, 23]. Verification of conformance to a simple abstract specification has been performed using refinement techniques, which establish simulation relations between the implementation and specification, using partly manual techniques [11, 8, 12, 33].

Amit et al [3] verify linearizability by verifying conformance to an abstract specification, which is the same as the implementation, but restricted to serialized executions. They build a specialized abstract domain that correlates the state (including the heap cells) of a concrete thread and the state of the serialized version, and a sequential reference data structure. The approach can handle a bounded number of threads. Berdine et al [4] generalize the approach to an unbounded number of threads by making the shape analysis thread-modular. In our approach, we need not keep track of heaps emanating from sequential reference executions, and so we can use a simpler shape analysis. Plain thread-modular analysis is also not powerful enough to analyze e.g. algorithms with explicit memory management. [4] thus improves the precision by correlating local states of different threads. This causes however a severe state-space explosion which limits the applicability of the method.

Vafeiadis [27] formulates the specification using an unbounded sequence of data values that represent, e.g., a queue or a stack. He verifies conformance using a specialized abstraction to track values in the queue and correlate them with values in the implementation. Like [25], our technique for handling values in queues need only consider a small number of data values (not an unbounded one), for which it is sufficient to track equalities. The approach is extended in [28] to automatically infer the position of linearization points: these have to be supplied in our approach.

Our use of data variables in observers for specifying properties that hold for all data values in some domain is related in spirit to the identification of arbitrary but fixed objects or resources by Emmi et al. [13] and Kidd et al. [18]. In the framework of regular model checking, universally quantified temporal logic properties can be compiled into automata with data variables that are assigned arbitrary initial values [1].

Segalov et al. [24] continue the work of [4] by also considering an analysis that keeps track of correlations between threads. They strive to counter the state-space explosion that [4] suffers from, and propose optimizations that are based on the assumption that inter-process relationships that need to be recorded are relatively loose, allowing a rather crude abstraction over the state of one of the correlated threads. These optimizations do not work well when thread correlations are tight. Our experimental evaluation in Section 6 shows that our optimizations of the thread correlation approach achieve significantly better analysis times than [24].

There are several works that apply different verification techniques to programs with a bounded number of threads, including the use of TVLA [35]. Several approaches produce decidability results under limited conditions [7], or techniques based on non-exhaustive testing [6] or state-space exploration [32] for a bounded number of threads.

2 Programs

We consider systems consisting of an arbitrary number of concurrently executing threads. Each thread may at any time invoke one of a finite set of methods. Each method declares local variables (including the input parameters of the method) and a method body. In this paper, we assume that variables are either pointer variables (to heap cells), or data variables (assuming values from an unbounded or infinite domain, which will be denoted by $\mathbb{D}$). The body is built in the standard way from atomic commands using stan-

dard control flow constructs (sequential composition, selection, and loop constructs). Method execution is terminated by executing a `return` command, which may return a value. The global variables can be accessed by all threads, whereas local variables can be accessed only by the thread which is invoking the corresponding method. We assume that the global variables and the heap are initialized by an initialization method, which is executed once at the beginning of program execution.

Atomic commands include assignments between data variables, pointer variables, or fields of cells pointed to by a pointer variable. The command **new** `node()` allocates a new structure of type `node` on the heap, and returns a reference to it. The cell is deallocated by the command **free**. The compare-and-swap command `CAS(&a,b,c)` atomically compares the values of `a` and `b`. If equal, it assigns the value of `a` to `c` and returns `TRUE`, otherwise, it leaves `a` unchanged and returns `FALSE`.

As an example, Figure 1 shows a version of the concurrent queue by Michael and Scott [20]. The program represents a queue as a linked list from the node pointed to by `Head` to a node that is either pointed by `Tail` or by `Tail`'s successor. The global variable `Head` always points to a dummy cell whose successor, if any, stores the head of the queue. In the absence of garbage collection, the program must handle the ABA problem where a thread mistakenly assumes that a globally accessible pointer has not been changed since it previously accessed that pointer. Each pointer is therefore equipped with an additional `age` field, which is incremented whenever the pointer is assigned a new value.

The queue can be accessed by an arbitrary number of threads, either by an enqueue method `enq(d)`, which inserts a cell containing the data value `d` at the tail, or by a dequeue method `deq(d)` which returns `empty` if the queue is empty, and otherwise advances `Head`, deallocates the previous dummy cell and returns the data value stored in the new dummy cell. The algorithm uses the atomic compare-and-swap (CAS) operation. For example, the command `CAS(&Head, head, ⟨next.ptr,head.age+1⟩)` at line 29 of the `deq` method checks whether the extended pointer `Head` equals the extended pointer `head` (meaning that both fields must agree). If not, it returns `FALSE`. Otherwise it returns `TRUE` after assigning `⟨next.ptr,head.age+1⟩` to `Head`.

3 Specification by Observers

To specify a correctness property, we instrument each method to generate abstract events. An *abstract event* is a term of the form $l(d_1, \ldots, d_n)$ where l is an event type, taken from a finite set of event types, and $d_1, \ldots, d_n$ are data values in $\mathbb{D}$. To specify linearizability, the abstract event $l(d_1, \ldots, d_n)$ generated by a method should be such that l is the name of the method, and $d_1, \ldots, d_n$ is the sequence of actual parameters and return values in the current invocation of the method. This can be established using standard sequential verification techniques.

We illustrate how to instrument the program of Figure 1 in order to specify that it is a linearizable implementation of a queue. The linearization points • are at line 9, 21 and 29. For instance, line 9 of the `enq` method called with data value `d` is instrumented to generate the abstract event `enq(d)` when the `CAS` command succeeds; no abstract event is generated when the `CAS` fails. Generation of abstract events can be conditional. For

```
void initialize() {                              INIT
  node* n := new node();
  n→next.ptr := NULL;
  Head.ptr := n;
  Tail.ptr := n;
}
```

```
struct node {data val, pointer_t next}
struct pointer_t {node* ptr, int age}

pointer_t Head, Tail;
```

```
                                                 ENQ
0  void enq(data d){
1    node* n := new node();
2    n→val := d;
3    n→next.ptr := NULL;
4    while(TRUE){
5      pointer_t tail := Tail;
6      pointer_t next := tail.ptr→next;
7      if(tail = Tail)
8        if(next.ptr = NULL)
9          if(CAS(&tail.ptr→next, next,  •
10               ⟨n,next.age+1⟩))
11           break;
12       else
13         CAS(&Tail,tail,⟨next.ptr, tail.age+1⟩);
14   }
15   CAS(&Tail, tail, ⟨n, tail.age+1⟩);
16 }
```

```
                                                 DEQ
17 data deq(){
18   while(TRUE){
19     pointer_t head := Head;
20     pointer_t tail := Tail;
21     pointer_t next := head.ptr→next;  •
22     if(head = Head)
23       if(head.ptr = tail.ptr)
24         if(next.ptr = NULL)
25           return empty;
26         CAS(&Tail, tail, ⟨next.ptr, tail.age+1⟩);
27       else
28         data result := next.ptr→val;
29         if(CAS(&Head, head,  •
30               ⟨next.ptr,head.age+1⟩))
31           break;
32   }
33   free(head.ptr);
34   return result;
35 }
```

Fig. 1. Michael & Scott's non-blocking queue [20].

instance, line 21 of the `deq` method is instrumented to generate `deq(empty)` when the value assigned to `next` satisfies `next.ptr = NULL` (i.e., it will cause the method to return `empty` at line 25).

Each execution of the instrumented program will generate a sequence of abstract events called a *trace*. A *correctness property* (or simply a *property*) is a set of traces. We say that an instrumented program *satisfies* a property if each trace of the program is in the property. In contrast to the classical (finite-state) automata-theoretic approach [30], we specify properties by *infinite-state* automata, called *observers*. An observer has a finite set of control locations, and a finite set of data variables that range over potentially infinite domains. It observes the trace and can reach an accepting control location if the trace is not in the property.

Formally, let a *parameterized event* be a term of the form $l(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are formal parameters. We will write $\overline{p}$ for $p_1, \ldots, p_n$, and $\overline{d}$ for $d_1, \ldots, d_n$. An *observer* consists of a finite set of *observer locations*, one of which is *initial* and some of which are *accepting*, a finite set of *observer variables*, and a finite set of *transitions*. Each transition is of form $s \xrightarrow{l(\overline{p});g} s'$ where s, s' are observer locations, $l(\overline{p})$ is a parameterized event, and the guard g is a Boolean combination of equalities over formal parameters $\overline{p}$, and observer variables. Intuitively, it denotes that the observer can move from location s to location s' when an abstract event of form $l(\overline{d})$ is generated such that $g[\overline{d}/\overline{p}]$ is true. Note that the values of observer variables are not updated in a transition. An *observer configuration* is a pair $\langle s, \vartheta \rangle$, where s is an observer location, and ϑ maps each observer variable to a value in the data domain $\mathbb{D}$. The configuration is initial if s is initial; thus the variables can assume any initial values. An *observer step* is a triple $\langle s, \vartheta \rangle \xrightarrow{l(\overline{d})} \langle s', \vartheta \rangle$ such that there is a transition $s \xrightarrow{l(\overline{p});g} s'$ for which $g[\overline{d}/\overline{p}]$ is true. A *run* of the observer on a trace $\sigma = l_1(\overline{d}_1) l_2(\overline{d}_2) \cdots l_n(\overline{d}_n)$ is a sequence of

observer steps $\langle s_0, \vartheta \rangle \xrightarrow{\texttt{l}_1(\overline{d}_1)} \cdots \xrightarrow{\texttt{l}_n(\overline{d}_n)} \langle s_n, \vartheta \rangle$ where s_0 is the initial observer location. The run is *accepting* if s_n is accepting. A trace σ is *accepted* by an observer $\mathcal{A}$ if $\mathcal{A}$ has an accepting run on σ. The property specified by $\mathcal{A}$ is the set of traces that are not accepted by $\mathcal{A}$.

Since the data variables can assume arbitrary initial values, observers can specify properties that are universally quantified over all data values. If a trace violates such a property for some data values, the observer can non-deterministically choose these as initial values of its variables, and thereafter detect the violation when observing the trace. Several data structures can be specified by a collection of properties, each of which is represented by an observer. For instance, a set can be specified by a collection of properties, one of which is that a data value can be deleted, only if it has been previously inserted. The observer in Figure 2 specifies this property: it accepts executions in which for some data value d, a $\texttt{delete}(d)$-event is observed even though no earlier $\texttt{insert}(d)$-event has been observed.

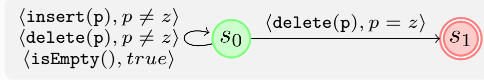

Fig. 2. An observer for checking that no data value can be deleted if it has not been first inserted. The variable z is an observer variable.

4 Data Independence

In the previous section, we showed how observers can specify some data structures, such as sets, in a straight-forward way. However, in order to specify some other data structures, including queues and stacks, for which one must be able to "count" the number of copies of a data value that have been inserted but not removed, we must additionally employ an extension of a data independence argument, originating from Wolper [34], as follows.

The argument assumes that for each trace, there is a fixed subset of all occurrences of data values in the trace, called the set of *input occurrences*. Formally, this subset can be arbitrary, but to make the argument work, input occurrences should typically be the data values that are provided as actual parameters of method invocations. Thus, in the program of Figure 1, the input occurrences are the parameters of $\texttt{enq}(\texttt{d})$ events, whereas parameters of $\texttt{deq}(\texttt{d})$ events are *not* input occurrences, since they are provided as return values.

Let us introduce some definitions. A trace is *differentiated* if all its input occurrences are pairwise different. A *renaming* is any function $f : \mathbb{D} \mapsto \mathbb{D}$ on the domain of data values. A renaming f can be applied to trace σ, resulting in the trace $f(\sigma)$, where each data value $\texttt{d}$ in σ has been replaced by $f(\texttt{d})$. A set Σ of traces is *data independent* if for any trace $\sigma \in \Sigma$ the following two conditions hold:

- $f(\sigma) \in \Sigma$ for any renaming f, and
- there exists a differentiated trace $\sigma_d \in \Sigma$ with $f(\sigma_d) = \sigma$ for some renaming f.

We say that a program is *data independent* if the set of its traces is data independent. A program, like the one in Figure 1, can typically be shown to be data independent by

a simple syntactic analysis that checks that data values are not manipulated or tested, but only copied. In a similar manner, a correctness property is *data independent* if the set of traces that it specifies is data independent. From these definitions, the following theorem follows directly.

Theorem 1. *A data independent program satisfies a data independent property iff its differentiated traces satisfy the property.* □

Thus, when checking that a data independent program satisfies a data independent property, it suffices to check that all differentiated traces of the program belong to the property. Hence, an observer for a data independent property need only accept the differentiated traces that violate the property. It should not accept any (differentiated or non-differentiated) trace that satisfies it.

Note that the set of traces of a set is *not* data independent, e.g., since it contains a trace where two different data values are inserted, but *not* its renaming which inserts the same data value twice. This is not a problem, since the set of *all* traces of a set can be specified by observers, without using a data independence argument.

The key observation is now that the differentiated traces of queues and stacks can be specified succinctly by observers with a small number of variables. In the case of a FIFO queue, its differentiated traces are precisely those that satisfy the following four properties for all data values d_1 and d_2.

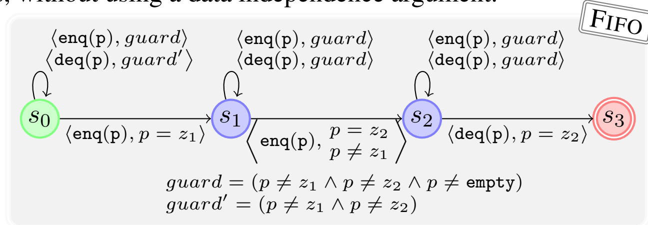

Fig. 3. An observer to check that FIFO ordering is respected. All unmatched abstract events, for example $\langle \texttt{deq(p)}, p = z_1 \rangle$ at location s_1, send the observer to a sink state.

- NO CREATION: d_1 must not be dequeued before it is enqueued
- NO DUPLICATION: d_1 must not be dequeued twice,
- NO LOSS: `empty` must not be returned if d_1 has been enqueued but not dequeued,
- FIFO: d_2 must not be dequeued before d_1 if it is enqueued after d_1 is enqueued.

Each such property can be specified by an observer with one or two variables. If the property is violated by some specific data values d_1 and d_2, then there is some run of the observer, in which the initial values of the variables are d_1 and d_2, which leads to an accepting state. Figure 3 shows an observer for the FIFO property.

We can also provide an analogous characterization of the differentiated traces of a stack.

5 Verification by Shape Analysis

To verify that no trace of the program is accepted by an observer, we form, as in the automata-theoretic approach [30], the cross-product of the program and the observer, synchronizing on abstract events, and check that this cross-product cannot reach a configuration where the observer is in an accepting state.

The analysis needs to deal with the challenges of an unbounded data domain, an unbounded number of concurrently executing threads, an unbounded heap, and an explicit memory management. As indicated in Section 1, the explicit memory management implies that the assertions generated by our analysis must be able to track correlations between pairs of threads. We present our shape analysis in two steps. We first describe a symbolic encoding of the configurations of the program and then present the verification procedure.

5.1 Representation of Symbolic Encodings

A symbolic encoding characterizes *all* the configurations of the program from the point of view of two distinct executing threads. This is done by recording the relationships of the local configurations of the two threads with each other, the relationships of the local variables of a thread with global variables, the observer configuration, and the assertions about the heap. It is a combination of several layers of conjunctions and disjunctions. In this section, let us fix two thread identifiers i_1 and i_2 and let us first introduce some necessary definitions in a bottom-up manner.

Cell terms. Let a *cell term* be one of the following: (i) a global pointer variable y, which denotes the cell pointed to by the global variable y, (ii) a term of the form $x[i_j]$ (where $j = 1$ or $j = 2$) for a local pointer variable x of thread i_j, which denotes the cell pointed to by the thread-i_j-local-copy of x, (iii) a special term `NULL`, `UNDEF`, or `FREE`, or (iv) a cell variable, which denotes a cell whose data value is equal to the current value of an observer variable. (Note that the value of an observer variable is fixed during a run of the observer). The latter allows us to keep track of the data in the heap cells, even in the case where a heap cell is not denoted by any pointer variable (in order to verify, e.g., the FIFO property of a queue). We use $CT(i_1, i_2)$ to denote the set of all cell terms (of thread i_1 and i_2).

Atomic heap constraint. In order to obtain an efficient and practical analysis, which does not lead to a severe explosion of formulas, we have developed a novel representation, adapted from the transitive closure logic of [5]. The representation is motivated by the observation that relationships between pairs of pointer variables are typically independent. The key aspect of the representation is that it is sufficient to consider only pairs of variables rather than correlating all variables. An atomic heap constraint is of one of the following forms (where t_1 and t_2 are two cell terms):

- $t_1 = t_2$ means that the cell terms t_1 and t_2 denote the same cell,
- $t_1 \mapsto t_2$ means that the `next` field of the cell denoted by t_1 denotes the cell denoted by t_2,
- $t_1 \dashrightarrow t_2$ means that the cell denoted by t_2 can be reached by following a chain of two or more `next` fields from the cell denoted by t_1,
- $t_1 \bowtie t_2$ means that none of $t_1 = t_2$, $t_1 \mapsto t_2$, $t_2 \mapsto t_1$, $t_1 \dashrightarrow t_2$, or $t_2 \dashrightarrow t_1$ is true.

We use *Pred* to denote the set $\{=, \mapsto, \leftarrowtail, \dashrightarrow, \dashleftarrow, \bowtie\}$ of all shape relational symbols. We let $t =$ `NULL` denote that t is null, $t \mapsto$ `UNDEF` denote that t is undefined, and $t \mapsto$ `FREE` denote that t is unallocated.

Joined shape constraint. A joined shape constraint, for thread i_1 and i_2, denoted as $M(i_1, i_2)$, is a (typically large) conjunction $\bigwedge_{t_1,t_2 \in CT(i_1,i_2)} \pi[t_1, t_2]$ where $\pi[t_1, t_2]$ is a non-empty disjunction of atomic heap constraints. Intuitively, it is a matrix representing the heap parts accessible by the two threads (along with the cell data). Such a representation can be (exponentially) more concise than using a large disjunction of conjunctions of atomic heap constraints, at the cost of some loss of precision.

We say that a joined shape constraint $M(i_1, i_2)$ is *saturated* if for all terms x, y, and z in $CT(i_1, i_2)$, every atomic heap constraint from the disjunction $\pi[x, z]$ implies the heap constraints that one can derive from those found in $\pi[x, x]$, $\pi[x, y]$, $\pi[y, y]$, $\pi[y, z]$, and $\pi[z, z]$. Any joined shape constraint can be saturated by a straightforward fixpoint procedure, analogous to [5] or the one for DBMs [9]. For instance, let $\pi[x, z]$ be $x \mapsto z$ and $\pi[y, z]$ be $y \mapsto z \vee y \dashrightarrow z$ and let $\pi[x, x]$ and $\pi[y, y]$ admit only equality (there is no loop involving x or y). Then $\pi[x, y]$ can contain the disjuncts $x = y$, $x \bowtie y$, which are consistent with $x \mapsto z$ and $y \mapsto z$. It can also contain $x \leftarrow\!\shortmid y$, $x \dashleftarrow y$, and $x \bowtie y$, that are consistent with $x \mapsto z$ and $y \dashrightarrow z$. In short, x cannot reach y, thus when saturating, we remove $x \mapsto y$ and $x \dashrightarrow y$ from $\pi[x, y]$.

Symbolic Encoding. We can now define formally a symbolic encoding over two threads. A symbolic encoding is a disjunction $\Theta[i_1, i_2]$ of formulas of the form $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ where $\sigma[i_1, i_2]$ is a *control formula* and $\phi[i_1, i_2]$ is a *shape formula*.

A *control formula* $\sigma[i_1, i_2]$ contains (i) the current control location of threads i_1 and i_2, and the observer, and (ii) a conjunction encompassing the relations between the `age` fields of any pair of terms. For instance, when analyzing the program in Figure 1, this conjunction includes among others, for a thread i, `head[i].age`$\simeq$`Head.age` and `tail[i].ptr`$\rightarrow$`next.age`$\simeq$`next[i].age`, where $\simeq \in \{<, =, >\}$.

A *shape formula* $\phi[i_1, i_2]$ is a joined shape constraint conjoined with a formula $\psi[v_1, \ldots, v_m, z_1, \ldots, z_n]$ which links the cell variables $v_1, \ldots, v_m$ with the observer variables $z_1, \ldots, z_n$ that are used to keep track of heap cells with values equal to the observer variables. Formally, $\phi[i_1, i_2]$ is a formula of the form

$$\exists v_1, \ldots, v_m. \, [\psi[v_1, \ldots, v_m, z_1, \ldots, z_n] \; \wedge \; M(i_1, i_2)]$$

5.2 Verification Procedure

We compute an invariant of the program of the form $\forall i_1, i_2. \, (i_1 \neq i_2 \Rightarrow \Theta[i_1, i_2])$ which characterizes the configurations of the program from the point of view of two distinct executing threads i_1 and i_2. We obtain the invariant by a standard fixpoint procedure, starting from a formula that characterizes the set of initial configurations of the program. For two distinct threads i_1 and i_2, and for each control formula $\sigma[i_1, i_2]$, our analysis will generate one shape formula $\phi[i_1, i_2]$.

The fixpoint analysis performs a postcondition computation that results in a set of possible successor combinations of control and shape formulas. The new shape formulas of which the control formula already appears in the original $\Theta[i_1, i_2]$ will be used to weaken the corresponding old shape formula. Otherwise, if the control state is new, a new disjunct is added to $\Theta[i_1, i_2]$.

For two threads i_1 and i_2, we must consider two scenarios: either thread i_1 or i_2 performs a step, or some other (interfering) thread i_3, (distinct from i_1 and i_2), performs a step.

Postcondition computation. In the first scenario, where one of the threads i_1 or i_2 performs a step, we can compute the postcondition of $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ as follows. $\sigma[i_1, i_2]$ is first updated to a new control state $\sigma'[i_1, i_2]$ in the standard way (by updating the possible values of control locations and observer state). $\phi[i_1, i_2]$ is then updated to $\phi'[i_1, i_2]$ by updating each conjunct $\pi[t_1, t_2]$ according to the particular program statement that the thread is performing. In general, we (i) remove all disjuncts that must be falsified by the step (this may require splitting the formula into several stronger formulas whenever the falsification might be ambiguous), (ii) add all disjuncts that may become true by the step, (iii) saturate the result.

Consider for instance the program statement `x:=y.next`. Since only the value of x is changing, the transformer updates only conjuncts $\pi[t, x]$ and $\pi[x, t]$ where $t \in CT(i_1, i_2)$. All assertions about x are reset by setting every conjunct $\pi[x, t]$ and $\pi[t, x]$ to $Pred$, for all $t \in CT(i_1, i_2)$. (The disjunction over all elements of $Pred$ is the assertion $true$). We then set $\pi[x, y]$ to $x \leftarrow\!\!\shortmid y$, $\pi[y, x]$ to $y \mapsto x$ and derive all predicates that may follow by transitivity. Finally, we saturate the formula. It prunes the (newly added) predicates that are inconsistent with the rest of the shape formula.

For `x.next:=y`, it is important to know the reachabilities that depend on the pointer x.`next`. The representation might potentially contain imprecision (it might for instance state that, for a term t, $\pi[t, x]$ contains $\dashleftarrow$ and $\dashrightarrow$). Hence, we first split the formula into stronger formulas in such a way that we disambiguate the part of the reachability relation involving x. On each resulting formula, we then remove reachability predicates between cell terms that depend on x.`next` (e.g., we remove $u \dashrightarrow v$ if $u \dashrightarrow x$ and $x \dashrightarrow v$). We then set $\pi[x, y]$ to $x \mapsto y$ and derive all predicates that may follow by transitivity (e.g., if $u \dashrightarrow x$ and $y \dashrightarrow v$, we add $u \dashrightarrow v$), and we saturate the result.

Interference. In the case where we need to account for possible interference on the formula $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ by another thread, (distinct from i_1 or i_2), we proceed as follows. We (i) extend the formula with the interfering thread, (ii) compute a postcondition as described in the first scenario and (iii) project away the interfering thread.

Step (i) combines a given formula $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ with the information of an extra thread i_3. In a similar manner to [2], the resulting formula is of the form $(\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3])$ such that any projection to two threads is a formula compatible with some disjunct of $\Theta[i_1, i_2]$. To generate all such formulas involving three threads, we must, besides $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ itself, consider all pairs of disjuncts $(\sigma_\bullet[i_2, i_3] \wedge \phi_\bullet[i_2, i_3])$ and $(\sigma_\circ[i_1, i_3] \wedge \phi_\circ[i_1, i_3])$, such that $\sigma[i_1, i_2] \wedge \sigma_\bullet[i_2, i_3] \wedge \sigma_\circ[i_1, i_3]$ is consistent. In this case, we generate the formula $\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3]$ where

$$\sigma[i_1, i_2, i_3] \equiv \sigma[i_1, i_2] \wedge \sigma_\bullet[i_2, i_3] \wedge \sigma_\circ[i_1, i_3]$$
$$\phi[i_1, i_2, i_3] \equiv \phi[i_1, i_2] \wedge \phi_\bullet[i_2, i_3] \wedge \phi_\circ[i_1, i_3]$$

We then saturate $\phi[i_1, i_2, i_3]$ (in the same way as for joined shape formulas over two threads). For each statement S of thread i_3 that can be executed when $\sigma[i_1, i_2, i_3]$ holds, we compute (step ii) its postcondition $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$. Finally (step iii),

$\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$ is projected back onto $\sigma'[i_1, i_2] \wedge \phi'[i_1, i_2]$ by removing all information about the variables of thread i_3.

Since the domain of control formulas and the domain of shape formulas over a fixed number of cell terms are finite, the abstract domain of formulas $\forall i_1, i_2. \ (i_1 \neq i_2 \Rightarrow \Theta[i_1, i_2])$ is finite as well. The iteration of postcondition computation is thus guaranteed to terminate.

6 Experimental results

We have implemented a prototype in OCaml and used it to automatically establish the conformance of concurrent data-structures (including lock-free and lock-based stacks, queues and priority queues) to their operational specification (implying their linearizability). Our analysis also implicitly checks for standard shape-related errors such as null/undefined pointer dereferencing (taking into account the known dangling pointers' dereferences [21]), double-free, or presence of cycles.

Some of the examples are verified in the absence of garbage collection, in particular, the lock-free versions of Treiber's [26] stack and Michael&Scott's queue (see Figure 1). We hereafter refer to them as Treiber's stack and M&S's queue, and garbage collection as GC. The verification of these examples is extensively demanding as it requires to correlate the possible states of the threads with high precision. We are not aware of any other method capable of verifying high level functionality of these benchmarks.

In addition to establishing correctness of the original versions of the benchmark programs, we also stressed our tool with few examples in which we intentionally inserted bugs (cf. Table 2). As expected, the tool did not establish correctness of these erroneous programs since the approach is sound. For example, we tested whether stacks (resp. queues) implementations can exhibit fifo (resp. lifo) traces, we tested whether values can be lost (loss observer), or memory errors can be triggered (memo observer accepts on memory errors made visible), we moved linearization points to wrong positions, and we tested a program which stores wrong values of inserted data. In all these cases, the analysis correctly reported traces that violated the concerned safety property. Finally, we ran the data structure implementations without garbage collection discarding the `age` counters and our (precise) analysis produced as expected a trace involving the ABA problem [17].

We ran the experiments on a 3.5 GHz processor with 8GB memory. We report, in Table 1, the running times (in seconds) and the final number of joined shape constraints generated ($|C|$, reduced by symmetry).

We also include a succinct comparison with related work. Although it is often unfair to compare approaches solely based on running times of different tools, we believe that such a comparison can give an idea of the efficiency of the involved approaches. Our running times on the versions of Treiber's stack and M&S's queue that assume GC are comparable with the results of [29]. However, the verification of versions that do not assume GC is, to the best of our knowledge, beyond the reach of [29] (since it does not correlate states of different threads). [24] verifies linearizability of concurrent implementations of sets, e.g., a lock-free `CAS`-based set [31] (verified in 2975s) of a comparable complexity to M&S's queue without GC (550s with our prototype). Basic memory safety of M&S's queue and two-locks queue [20] without GC was also verified

Table 1. Experimental Results.

		Conformance		Safety only	
Data-structure	**Observers**	**Time**	$\lvert C \rvert$	**Time**	$\lvert C \rvert$
Coarse Stack	stack+	0.02s	436	0.01s	102
Coarse Stack, no GC		0.07s	569	0.01s	130
Coarse Queue	queue+	0.04s	673	0.01s	196
Coarse Queue, no GC		0.48s	1819	0.10s	440
Two-Locks Queue[20]	queue+	0.08s	1830	0.02s	488
Two-Locks Queue, no GC		0.73s	3460	0.13s	784
		vs 47s in [4]		*vs* 6162s/304s in [35]	
Coarse Priority Queue (Buckets)	prio	0.24s	1242	0.07s	526
Coarse Priority Queue (List-based)		0.04s	499	0.01s	211
Bucket locks Priority Queue		0.22s	1116	0.05s	372
Treiber's lock-free stack[26]	stack+	0.23s	714	0.01s	78
		vs 0.09s in [29]			
Treiber's lock-free stack, no GC	stack+	2.28s	1535	0.10s	190
		vs 53s in [4]			
M&S's lock-free queue[20]	queue+	3.31s	3476	0.44s	594
		vs 3.36s in [29]			
M&S's lock-free queue, no GC	queue+	550s	53320	25s	6410
		vs o.o.m. in [4]		*vs* 727s/309s in [35]	

stack+ (resp. queue+) is an observer encompassing
the loss, creation, duplication and lifo (resp. fifo) observers

in [35], but only for a scenario where all threads are either dequeuing or enqueuing. The verification took 727s and 309s for M&S's queue and 6162s and 304s for the two-locks queue. Our verification analysis produced the same result significantly faster, even allowing any thread to non deterministically decide to either enqueue or dequeue. In [4], linearizability of the Treibers's stack (resp. two-locks queue [20]) is verified in 53s (resp. 47s). We achieve the same result in less than 3 seconds. Finally, a variant of M&S's queue without GC could not be successfully verified in [4] due to lack of memory.

Table 2. Introducing intentional bugs: The analysis is sound and the programs are not verified.

Data-structure	**Modification**	**Observer**	**Output**	**Time**
Treiber's stack	none	fifo	bad trace	0.07s
Treiber's stack, no GC	none	fifo	bad trace	6.19s
M&S's queue	none	lifo	bad trace	1.26s
Two-locks queue	bad commit point	fifo	bad trace	0.02s
M&S's queue	bad commit point	loss	bad trace	0.51s
Treiber's stack	omitting data	lifo	bad trace	0.02s
Treiber's stack, no GC	discard ages	loss	bad trace	0.42s
Treiber's stack, no GC	discard ages	loss	cycle creation	0.01s
M&S's queue, no GC	discard ages	loss	bad trace	272s
M&S's queue, no GC	discard ages	loss	dereferencing null	0.01s
M&S's queue	swapped assignments	memo	dereferencing null	0.01s

7 Conclusions and Future Work

We have presented a technique for automated verification of temporal properties of concurrent programs, which can handle the challenges of infinite-state specifications, an unbounded number of threads, and an unbounded heap managed by explicit memory allocation. We showed how such a technique can be based naturally on the automata-theoretic approach to verification, by nontrivial combinations and extensions that handle unbounded data domains, unbounded number of threads, and heaps of arbitrary size. The result is a simple and direct method for verifying correctness of concurrent programs. The power of our specification formalism is enhanced by showing how the data-independence argument by Wolper [34] can be introduced into standard program analysis. Our method can be parameterized by different shape analyses. Although we concentrate on heaps with single selectors in the current paper, we expect that our method can be adapted to deal with multiple selectors, by integrating recent approaches such as [15]. Morever, our experminatation deals with the specification of stacks and queues. Other data structures, such as deques, can be handled in an analogous way.

References

1. P. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular model checking for LTL(MSO). *STTT*, 14(2):223–241, 2012.
2. P. A. Abdulla, L. s Holík, and F. Haziza. All for the price of few (parameterized verification through view abstraction), 2013. Accepted at VMCAI'2013.
3. D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *Proc. of CAV'07*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.
4. J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *Proc. of CAV'08*, volume 5123 of *LNCS*, pages 399–413. Springer Verlag, 2008.
5. J. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Proc. of VMCAI'06*, volume 3855 of *LNCS*, pages 207–221. Springer, 2006.
6. S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *Proc. of PLDI'10*, pages 330–340. ACM, 2010.
7. P. Cerný, A. Radhakrishna, D. Zufferey, S. Chaudhuri, and R. Alur. Model checking of linearizability of concurrent list implementations. In *Proc. of CAV'10*, volume 6174 of *LNCS*, pages 465–479. Springer, 2010.
8. R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Proc. of CAV'06*, volume 4144 of *LNCS*, pages 475–488. Springer, 2006.
9. D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite-State Systems*, volume 407 of *LNCS*. Springer Verlag, 1989.
10. S. Doherty, D. Detlefs, L. Groves, C. Flood, V. Luchangco, P. Martin, M. Moir, N. Shavit, and G. S. Jr. Dcas is not a silver bullet for nonblocking algorithm design. In *Proc. of SPAA'04*, pages 216–224. ACM, 2004.
11. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *Proc. FORTE'04*, volume 3235 of *LNCS*, pages 97–114. Springer, 2004.

12. T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Proc. of TACAS'10*, volume 6015 of *LNCS*, pages 296–311. Springer Verlag, 2010.
13. M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. In *Proc. of TACAS'09*, volume 5505 of *LNCS*, pages 352–367. Springer Verlag, 2009.
14. C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71(2):89–109, 2008.
15. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. Forest automata for verification of heap manipulation. *Formal Methods in System Design*, pages 1–24, 2012.
16. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
17. IBM. System/370 principles of operation, 1983.
18. N. Kidd, T. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, 2011.
19. M. Michael and M. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Rochester, NY, USA, 1995.
20. M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, pages 267–275, 1996.
21. M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 21–30, New York, NY, USA, 2002. ACM.
22. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proc. of PLDI'06*, pages 308–319. ACM, 2006.
23. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. of ICSE*, pages 386–396. IEEE, 2009.
24. M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, and M. Sagiv. Abstract transformers for thread correlation analysis. In *APLAS*, LNCS, pages 30–46. Springer, 2009.
25. O. Shacham. *Verifying Atomicity of Composed Concurrent Operations*. PhD thesis, Department of Computer Science, Tel Aviv University, 2012.
26. R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.
27. V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proc. of VMCAI*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.
28. V. Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010.
29. V. Vafeiadis. Rgsep action inference. In *Proc. of VMCAI'10*, volume 5944 of *LNCS*, pages 345–361. Springer, 2010.
30. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS'86*, pages 332–344, June 1986.
31. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *Proc. of PLDI'08*, pages 125–135. ACM, 2008.
32. M. Vechev, E. Yahav, and G. Yorsh. Experience with model checking linearizability. In *Proc. of SPIN'09*, volume 5578 of *LNCS*, pages 261–278. Springer, 2009.
33. L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proc. of PPOPP'05*, pages 61–71. ACM, 2005.
34. P. Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc. of POPL'86*, pages 184–193, 1986.
35. E. Yahav and S. Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.