Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

An Integrated Specification and Verification Technique for Highly Concurrent Data Structures

Parosh Aziz Abdulla^1, **Frédéric Haziza**^{1}, **Lukáš Holík**^{12}, **Bengt Jonsson**^{1}, **Ahmed Rezine**^{3*}

^{1} Uppsala University, Sweden
^{2} Brno University of Technology, Czech Republic
^{3} Linköping University, Sweden.

Received: date / Revised version: date

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 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.

Send offprint requests to:

* supported in part by the Uppsala Programming for Multicore Architectures Research Center (UPMARC), the Czech Science Foundation (project 13-37876P), the internal projects of Brno University of Technology FIT-S-12-1 and FIT-S-14-2486, and the CENIIT research organization (project 12.04).

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 6, 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 7. Section 8 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

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

pointer_t Head, Tail;
```

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

```
ENQ
void enq(data d){
   node* n := new node();
   n→val := d;
   n→next.ptr := NULL;
   while(TRUE){
      pointer_t tail := Tail;
      pointer_t next := tail.ptr→next;
      if(tail = Tail)
         if(next.ptr = NULL)
            if(CAS(&tail.ptr→next, next, ●
                  ⟨n,next.age+1⟩))
               break;
         else
            CAS(&Tail,tail,⟨next.ptr, tail.age+1⟩);
   }
   CAS(&Tail, tail, ⟨n, tail.age+1⟩);
}
```

```
DEQ
data deq(){
   while(TRUE){
      pointer_t head := Head;
      pointer_t tail := Tail;
      pointer_t next := head.ptr→next; ●
      if(head = Head)
         if(head.ptr = tail.ptr)
            if(next.ptr = NULL)
               return empty;
            CAS(&Tail, tail, ⟨next.ptr, tail.age+1⟩);
         else
            data result := next.ptr→val;
            if(CAS(&Head, head, ●
                  ⟨next.ptr,head.age+1⟩))
               break;
   }
   free(head.ptr);
   return result;
}
```

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

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 7 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 standard 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 `c` to `a` 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

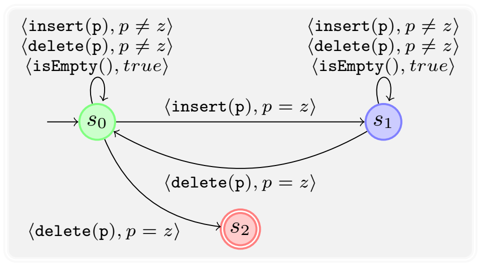


Fig. 2. An observer for deleting a non present data value.

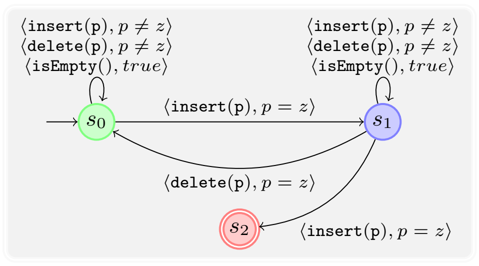


Fig. 3. An observer for inserting a data value that is already present.

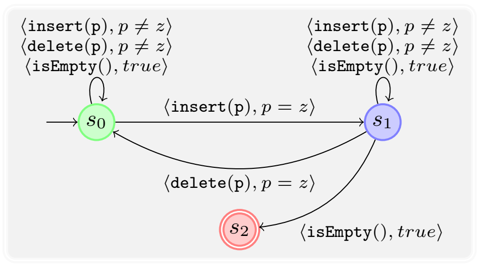


Fig. 4. An observer for losing a data value that is already present.

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 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 $\langle s_0, \vartheta \rangle \xrightarrow{l_1(\overline{d}_1)} \cdots \xrightarrow{l_n(\overline{d}_n)} \langle s_n, \vartheta \rangle$ 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 $\mathcal{A}$.

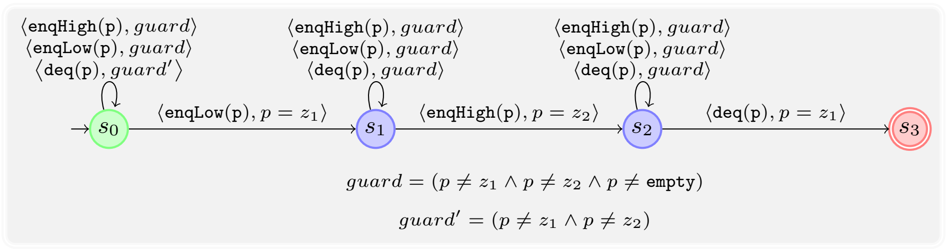


Fig. 5. A trace observer for checking that a low priority data value cannot be dequeued if there is a high priority data value that was later inserted. The variables z_1, z_2 are observer variables, and `empty` in an observer constant.

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.

3.1 Application: Sets Specification

We use three observers to exactly capture all set traces over $\cup_{d \in \mathbb{D}}$ {`insert(d)`, `deq(d)`}∪{`isEmpty()`} that violate the expected behavior of a correct set implementation. The three observers in figures 2, 3 and 4 have three states s_0, s_1 and s_2. In these observers, the initial state s_0 corresponds to positions in the runs where the non-determnistically tracked value stored in the observer variable z is not present in the set (i.e. each time it has been inserted it got deleted afterwards). The state s_1 corresponds to positions in the runs where the tracked value is present in the set (i.e. it has not been deleted since it was last inserted). The accepting state s_2 corresponds to positions in the runs where the bad behavior captured by the respective observers has been observed. For the observer depicted in figure 2, the captured bad behaviors are those where a data value is deleted although it is not present in the set. For the observer of figure 3, the captured bad behaviors are those where a data value is inserted although it is already present in the set. For the observer of figure 4, the captured behaviors are those where `isEmpty()` occurs although a data value is still present in the set.

3.2 Observers Alone Cannot "Count"

In the previous paragraph, we showed how observers can specify in a straight-forward way data structures such as sets. Registers and similar data structures (such as caches) where there is an a priori fixed bound on the number of equal data values that have been inserted but not yet retrieved can also be specified using appropriate observers. There are however data structures where observers alone are not enough to capture the specification. Queues and stacks are examples of such data structures. Here, the difference between the number of times a data value may be inserted and the number of times it is retrieved can be arbitray. In other words, the number of copies of the same data value that are present in the data structures can be arbitrary. As a result, one must be able to "count" the number of equal data values that have been inserted but not yet removed. Such data structures require therefore non-regular specifications in general. By restricting the allowed traces we can again use the observers defined in this section. For instance, assume a queue where data values are assigned a low (respectively high) priority each time they are inserted with `enqLow()` (respectively `enqHigh()`). A correct implementation of such a priority queue will not return a data value with low priority if one with high priority was later inserted. The observer of figure 3 captures all traces that violate this property and where no data value `d` is enqueued twice (whether with `enqLow(d)`, `enqHigh(d)`, or both). In the following section, we build on the idea of specifying restricted traces using observers and show, by leveraging on a data independence argument, that this is sufficient to completely specify data structures such as stacks and queues.

4 Data Independence

We adapt a data independence argument from Wolper [34]. 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 `enq(d)` events, whereas parameters of `deq(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 `d` in σ has been replaced by $f(\mathtt{d})$. 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.

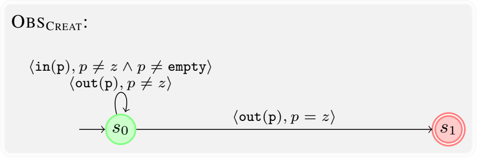

$\textsc{Obs}_{\textsc{Creat}}$:
$\langle \mathtt{in(p)}, p \neq z \wedge p \neq \mathtt{empty}\rangle$
$\langle \mathtt{out(p)}, p \neq z\rangle$
$\langle \mathtt{out(p)}, p = z\rangle$
s_0
s_1

Fig. 6. A trace observer for checking that no data-value can be extracted if it has not been inserted. The variable z is an observer variable, and `empty` in an observer constant.

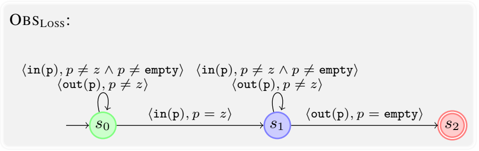

$\textsc{Obs}_{\textsc{Loss}}$:
$\langle \mathtt{in(p)}, p \neq z \wedge p \neq \mathtt{empty}\rangle$
$\langle \mathtt{out(p)}, p \neq z\rangle$
$\langle \mathtt{in(p)}, p \neq z \wedge p \neq \mathtt{empty}\rangle$
$\langle \mathtt{out(p)}, p \neq z\rangle$
$\langle \mathtt{in(p)}, p = z\rangle$
$\langle \mathtt{out(p)}, p = \mathtt{empty}\rangle$
s_0
s_1
s_2

Fig. 7. A trace observer for checking that an inserted value has to be extracted before the data-structure is declared empty. The variable z is an observer variable, and `empty` in an observer constant.

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. The following theorem states an important observation.

Theorem 1. *For any data independent sets of traces Σ and Σ', $\Sigma \subseteq \Sigma'$ iff the differentiated traces of Σ are in Σ'.*

Proof. If $\Sigma \subseteq \Sigma'$ then the differentiated traces of Σ are included in Σ'. Let σ be an arbitrary trace in Σ. We show $\sigma \in \Sigma'$. By data independence of Σ, there is a differentiated trace $\sigma_d \in \Sigma$ and a renaming f such that $f(\sigma_d) = \sigma$. By assumption, σ_d is also in Σ'. By data independence of Σ', $f(\sigma_d)$ is also in Σ', and hence $\sigma \in \Sigma'$. □

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. This means that whenever a data value is input twice in a trace, the observer can stop checking (i.e., move to a non-accepting sink state), since the trace will anyway be ignored.

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 completely and succinctly specified by observers with a small number of variables. We devote the following section to formalize and prove this fact.

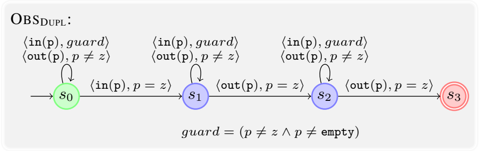

$\textsc{Obs}_{\textsc{Dupl}}$:
$\langle \mathtt{in(p)}, guard\rangle$
$\langle \mathtt{out(p)}, p \neq z\rangle$
$\langle \mathtt{in(p)}, guard\rangle$
$\langle \mathtt{out(p)}, p \neq z\rangle$
$\langle \mathtt{in(p)}, guard\rangle$
$\langle \mathtt{out(p)}, p \neq z\rangle$
$\langle \mathtt{in(p)}, p = z\rangle$
$\langle \mathtt{out(p)}, p = z\rangle$
$\langle \mathtt{out(p)}, p = z\rangle$
s_0
s_1
s_2
s_3
$guard = (p \neq z \wedge p \neq \mathtt{empty})$

Fig. 8. A trace observer for checking that no once-inserted data value can be extracted twice. The variable z is an observer variable, and `empty` in an observer constant.

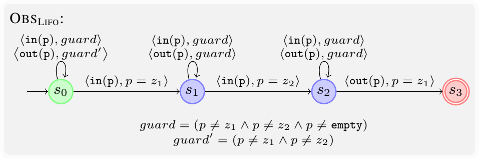

$\textsc{Obs}_{\textsc{Lifo}}$:
$\langle \mathtt{in(p)}, guard\rangle$
$\langle \mathtt{out(p)}, guard'\rangle$
$\langle \mathtt{in(p)}, guard\rangle$
$\langle \mathtt{out(p)}, guard\rangle$
$\langle \mathtt{in(p)}, guard\rangle$
$\langle \mathtt{out(p)}, guard\rangle$
$\langle \mathtt{in(p)}, p = z_1\rangle$
$\langle \mathtt{in(p)}, p = z_2\rangle$
$\langle \mathtt{out(p)}, p = z_1\rangle$
s_0
s_1
s_2
s_3
$guard = (p \neq z_1 \wedge p \neq z_2 \wedge p \neq \mathtt{empty})$
$guard' = (p \neq z_1 \wedge p \neq z_2)$

Fig. 9. An observer for detecting violations of the first inserted first extracted ordering. The initial state is s_0 and $\{s_3\}$ is the set of final states. The variables z_1, z_2 are observer variables, and `empty` in an observer constant.

5 Specifying Stacks and Queues Using Observers

We show in this section how to completely specifiy, using observers such as those introduced in section 3, and using the data independence argument introduced in section 4, the sequential behaviors of queues and stacks operating over the *arbitrary (and possibly infinite) data domain* $\mathbb{D}$. At the end of this section, we will show that the three observers of figures 6, 7 and 8, in addition to the observer of figure 9 (respectively, figure 10) are enough to specify a stack (respectively a queue) of arbitrary size[1]. We detail the approach for stacks and mention how to adapt it for the case of queues. First, we recall the natural operational specification of a stack and explain how we define its behavior using the set of traces it generates. Then, we propose, using the four simple observers mentioned above an alternative observational definition of a stack. The new definition abstracts away from the actual states and only considers properties of the generated traces. We write in the following $\mathbb{D}$ to mean $\mathbb{D} \setminus \{\mathtt{empty}\}$.

The functional specification of a sequential stack corresponds to the set of allowed finite sequences (we consider safety properties) of pushes and pops together with their arguments and return values. We use in the following $\mathtt{in}(d)$ (respectively $\mathtt{out}(d)$ and $\mathtt{out}(\mathtt{empty})$) to mean a `push(d)` (respectively `pop(d)` and `pop(empty)`). The specification of a sequential stack is a strict subset of $(\Sigma_{i/o})^*$, where $\Sigma_{i/o} = \{\mathtt{in}(d), \mathtt{out}(d) \mid d \in \mathbb{D}\} \cup \{\mathtt{out}(\mathtt{empty})\}$. We give in the following an operational and an observational characterization of the specification of a sequential stack and show their equivalence.

[1] When the observers in figures 6, 7, 8, 10 and 9 are used to specify a stack (respectively a queue), each occurence of `in(.)` should be replaced by `push(.)` (respectively `enq(.)`) and each occurence of `out(.)` should be replaced by `pop(.)` (respectively `deq(.)`)

5.1 Operational Specification of a Stack

A natural way to define the set of finite stack traces is to use a transition system T where the set of states is the set of possible stack contents, and where the transtions are labeled with $\Sigma_{i/o}$. More formally, T is a tuple $(\Sigma_{i/o}, (\mathbb{D})^*, \{\epsilon\}, \rightarrow)$, where the empty word $\epsilon \in (\mathbb{D})^*$ is the initial state, and the set of transitions $\rightarrow \subseteq (\mathbb{D})^* \times \Sigma_{i/o} \times (\mathbb{D})^*$ only includes all transitions of the form: $\langle w, \texttt{in}(d), d \cdot w\rangle$, $\langle d \cdot w, \texttt{out}(d), w\rangle$, or $\langle \epsilon, \texttt{out}(\texttt{empty}), \epsilon\rangle$, where $d \in \mathbb{D}$ and $w \in (\mathbb{D})^*$. A run of T is a finite sequence $\rho = w_0 e_1 w_1 \cdots e_n w_n$ with $w_0 = \epsilon$ and $\langle w_i, e_{i+1}, w_{i+1}\rangle \in \rightarrow$ for each $i : 0 \le i < n$. We say that ρ is a stack run. A trace of T is any sequence $e_1 \cdots e_n$ such that there is a stack run $w_0 e_1 w_1 \cdots e_n w_n$ of T. The operational sepcification of a stack, written ϕ^{op}_{stack}, is then the set of all traces of T.

Observe that the renaming of any stack trace is also a stack trace (just rename the states in the corresponding run). Also, given a trace σ resulting from a stack run ρ, one can obtain a differentiated trace whose renaming gives σ as follows. Repeat the same run but append a systematically incremented counter to the values that are input to the stack. It is easy to see that the same run as ρ, except for the appended counter values to the data, is also a stack run on a differentiated trace that can be renamed (by forgetting the counter) into σ. The set of traces φ^{op}_{stack} therefore satisfies the definition of data-independence introduced in section 4.

As a result, Theorem 1 implies that any data independent set of traces whose set of differentiated traces equals the set of differentiated stack traces does coincide with the set of stack traces. We write in the following $\varphi^{op}_{\mathit{diff},stack}$ to mean the set of differentiated traces in φ^{op}_{stack}.

5.2 Observational Specification of a Stack

We propose another specification for differentiated stack traces, written $\varphi^{obs}_{\mathit{diff},stack}$ which characterizes the set of differentiated stack traces as exactly those differentiated traces that are not accepted by any of four simple observers. Intuitively, such a differentiated trace satisfies the following four properties for all data values d_1 and d_2:

NO CREATION (OBS$_{\text{CREAT}}$, figure 6): d_1 must not be popped before it is pushed, i.e., data cannot be created,
NO LOSS (OBS$_{\text{Loss}}$, figure 7)): `empty` must not be returned if d_1 was pushed but not popped, i.e., data cannot be lost
NO DUPLICATION (OBS$_{\text{DUPL}}$, figure 8): d_1 must not be popped twice, i.e., data cannot be duplicated.
LIFO (OBS$_{\text{LIFO}}$, figure 9): d_2 must not be popped if d_1 was pushed after d_2 was pushed.

5.2.1 Differentiated Operational and Observational Specifications Coincide

Lemma 1 states that the differentiated operational specification of a stack equals the differentiated observational one.

Lemma 1. $\varphi^{op}_{\mathit{diff},stack} = \varphi^{obs}_{\mathit{diff},stack}$.

Proof. Recall the claim only concerns differentiated traces. We will make use of two properties that hold for any stack run $\rho = w_0 e_1 w_1 e_2 \cdots e_n w_n$.

– The *counting property* of a stack. We write $(a)^{\#}_{w}$ to mean the number of occurences of the letter $a \in A$ in the word $w \in A^*$, for a fixed alphabet A. Back to ρ, it is easy to show by induction that for any d in $\mathbb{D}$ and i s.t. $0 \le i \le n$, $(d)^{\#}_{w_i} = (\texttt{in}(d))^{\#}_{(e_1 \cdots e_i)} - (\texttt{out}(d))^{\#}_{(e_1 \cdots e_i)}$.
– The *ordering property* of a stack. Using the counting property and an induction on the length of ρ, one can show the following. Assume d_i and d_j are input before position k in ρ. If d_i is input before d_j, and if neither of them is output, then $w_k \in (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_j \cdot (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_i \cdot (\mathbb{D} \setminus \{d_i, d_j\})^*$.

We establish in the following inclusions in both directions in order to show the equality $\varphi^{op}_{\mathit{diff},stack} = \varphi^{obs}_{\mathit{diff},stack}$:

- $\varphi^{op}_{\mathit{diff},stack} \subseteq \varphi^{obs}_{\mathit{diff},stack}$. This direction is simple. Let $\rho = w_0 e_1 \cdots e_n w_n$ be a stack run giving a trace $\sigma = e_1 \ldots e_n$ in $\varphi^{op}_{\mathit{diff},stack}$. Suppose σ is accepted by one of the observers OBS$_{\text{CREA}}$, OBS$_{\text{LOSS}}$, OBS$_{\text{DUPL}}$, or OBS$_{\text{LIFO}}$ for some data values.
  1. σ cannot be accepted by OBS$_{\text{CREA}}$. Suppose it was the case and e_n is the $\texttt{out}(d)$ that labels the last transition in the observer. The fact that σ is accepted by the observer implies the data value d appearing in e_n does not participate in any $\texttt{in}(d)$ of the self loop on s_0. The counting property implies $d \notin w_n$. Yet $e_n = \texttt{out}(d)$ requires w_n to be of the form $d \cdot w$.
  2. σ cannot be accepted by OBS$_{\text{LOSS}}$ because the counting property implies w_{n-1} contains a d, yet $w_{n-1} = \epsilon$ since $e_n = \texttt{out}(\mathit{empty})$ appears at the end of the stack run ρ .
  3. σ cannot be accepted by OBS$_{\text{DUPL}}$ because the counting property requires w_{n-1} to contain no occurences of d. Yet $e_n = \texttt{out}(d)$ requires w_{n-1} to be of the form $d \cdot w$.
  4. σ cannot be accepted by OBS$_{\text{LIFO}}$ because that means ρ contains two events e_i, e_j with $1 \le i < j < n$ such that $e_i = \texttt{in}(d_i)$ and $e_j = \texttt{in}(d_j)$. The ordering property of a stack implies that $w_{n-1} \in (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_j \cdot (\mathbb{D} \setminus \{d_i, d_j\})^* \cdot d_i \cdot (\mathbb{D} \setminus \{d_i, d_j\})^*$. Yet for $e_n = \texttt{out}(d_i)$ to succeed, w_{n-1} needs to be of the form $d_i \cdot w$.
- $\varphi^{op}_{\mathit{diff},stack} \supseteq \varphi^{obs}_{\mathit{diff},stack}$. Suppose $\sigma = e_1 \ldots e_{n+1}$ in $\varphi^{obs}_{\mathit{diff},stack}$ is a shortest trace not in $\varphi^{op}_{\mathit{diff},stack}$. Hence, there is a stack run $\rho = w_0 e_1 \cdots e_n w_n$, but there is no w_{n+1} such that $\rho' = w_0 e_1 \cdots e_{n+1} w_{n+1}$ becomes a stack run.
  1. e_{n+1} cannot be $\texttt{in}(d)$ for some d because then it would be enough to choose $w_{n+1} = d \cdot w_n$ to get σ in $\varphi^{op}_{\mathit{diff},stack}$.
  2. if $e_{n+1} = \texttt{out}(\texttt{empty})$, then $w_n \neq \epsilon$ as otherwise choose $w_{n+1} = \epsilon$ and σ would be in $\varphi^{op}_{\mathit{diff},stack}$. Let $d \in w_n$. Using the counting property of a stack on ρ, we deduce that there is $e_i = \texttt{in}(d)$ for $i : 1 \le i \le n$,

but $\forall j : 1 \leq j \leq n.\ e_j \neq \mathtt{out}(d)$. Hence σ should have been accepted by $\mathrm{OBS}_{\mathrm{LOSS}}$, and therefore not in $\varphi^{obs}_{\mathit{diff},stack}$,

3. if $e_{n+1} = \mathtt{out}(d)$ for some data value d:
   (a) If $\mathtt{in}(d)$ does not appear in ρ, then σ should have been accepted by $\mathrm{OBS}_{\mathrm{CREA}}$ and therefore it cannot belong to $\varphi^{obs}_{\mathit{diff},stack}$
   (b) If $e_i = \mathtt{in}(d)$ and $e_j = \mathtt{out}(d)$ appear in ρ with $i, j : 1 \leq i, j \leq n$, then the counting property on the stack run ρ implies $i < j$. The trace σ should have been accepted by $\mathrm{OBS}_{\mathrm{DUPL}}$
   (c) If $e_i = \mathtt{in}(d)$ appears in ρ for some $i \leq n$ but without a $e_j = \mathtt{out}(d)$ for $i < j \leq n$, then the following holds. By the counting property, $w_n = w \cdot d \cdot w'$ with $w = d_k \cdot w''$. In addition, $d \neq d_k$ as otherwise ρ could be extended into a stack run. Using the counting property again, there must be a $e_k = \mathtt{in}(d_k)$ with $k : 1 \leq k \leq n$ and without any $\mathtt{out}(d_k)$ in the run up to n. If $i < k$, the trace should have been accepted by $\mathrm{OBS}_{\mathrm{LIFO}}$. If $k < i$, we use the ordering property to deduce that w_n should be in the language $(\mathbb{D} \setminus \{d, d_k\})^* \cdot d \cdot (\mathbb{D} \setminus \{d, d_k\})^* \cdot d_k \cdot (\mathbb{D} \setminus \{d_1, d_2\})^*$ which contradicts that $w_n = d_k \cdot w'' \cdot d \cdot w'$.

5.3 Operational and observational specification of queues.

For queues, $\mathtt{in}(d)$ (respectively $\mathtt{out}(d)$ and $\mathtt{out}(\mathtt{empty})$) stands for $\mathtt{enq}(\mathtt{d})$ (respectively $\mathtt{deq}(\mathtt{d})$ and $\mathtt{deq}(\mathtt{empty})$). The operational specification φ^{op}_{queue} is obtained by replacing $\rightarrow$ in section 5.1 by the smallest subset of $\big((\mathbb{D})^* \times \Sigma_{i/o} \times (\mathbb{D})^*\big)$ that includes, for every $d \in \mathbb{D}$ and $w \in (\mathbb{D})^*$, all transitions of the form: $\langle w, \mathtt{in}(d), w \cdot d\rangle$, $\langle d \cdot w, \mathtt{out}(d), w\rangle$, and $\langle \epsilon, \mathtt{out}(\mathtt{empty}), \epsilon\rangle$. $\varphi^{op}_{\mathit{diff},queue}$ is the restriction of φ^{op}_{queue} to the set of differentiated traces. The observational specification $\varphi^{obs}_{\mathit{diff},queue}$ contains exactly those differentiated traces that are not accepted by any of the following four observers: $\mathrm{OBS}_{\mathrm{CREA}}$, $\mathrm{OBS}_{\mathrm{LOSS}}$, $\mathrm{OBS}_{\mathrm{DUPL}}$, or $\mathrm{OBS}_{\mathrm{FIFO}}$ (figure 10). Intuitively, a differentiated trace that is not accepted by the observer $\mathrm{OBS}_{\mathrm{FIFO}}$ satisifies the following property for any data values d_1, d_2:

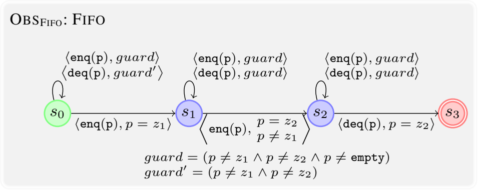
FIFO ($\mathrm{OBS}_{\mathrm{FIFO}}$, figure 10): d_2 must not be dequeued if d_1 was not dequeued since it was enqueued before d_2 was enqueued.

Lemma 2. $\varphi^{op}_{\mathit{diff},queue} = \varphi^{obs}_{\mathit{diff},queue}$.

Proof. Similar to the proof of lemma 1. We make use of the same counting property as in the stack case. We modify the ordering property to reflect the FIFO ordering (instead of the LIFO one for a stack). The other modifications are straightforward.

6 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],



$\mathrm{OBS}_{\mathrm{FIFO}}$: FIFO
$\langle \mathtt{enq(p)}, guard\rangle$
$\langle \mathtt{deq(p)}, guard'\rangle$
$\langle \mathtt{enq(p)}, guard\rangle$
$\langle \mathtt{deq(p)}, guard\rangle$
$\langle \mathtt{enq(p)}, guard\rangle$
$\langle \mathtt{deq(p)}, guard\rangle$
s_0
s_1
s_2
s_3
$\langle \mathtt{enq(p)}, p = z_1\rangle$
$\left\langle \mathtt{enq(p)}, \begin{matrix} p = z_2 \\ p \neq z_1 \end{matrix} \right\rangle$
$\langle \mathtt{deq(p)}, p = z_2\rangle$
$guard = (p \neq z_1 \wedge p \neq z_2 \wedge p \neq \mathtt{empty})$
$guard' = (p \neq z_1 \wedge p \neq z_2)$

Fig. 10. An observer to check that FIFO ordering is respected. All unmatched abstract events, for example $\langle \mathtt{deq(p)}, p = z_1\rangle$ at location s_1, send the observer to a sink state.

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.

6.1 Symbolic Encoding

The symbolic encoding is used for characterizing the set of reachable configurations of the program from the point of view of two distinct executing threads. Roughly, 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 assertions about the heap. It is a combination of several layers of conjunctions and disjunctions.

Below, we will use Figure 11 to explain the main concepts in the symbolic encoding. The left part of the figure shows a typical configuration of the heap that arises during an execution of the Michael & Scott algorithm, when run against the observer of Figure 10. The right part of the figure shows a symbolic encoding that is satisfied by the shape. Note that the symbolic encoding can represent more shapes. The heap consists of six cells operated on by two active threads Thread 1 (depicted in yellow) and Thread 2 (depicted in pink). The threads are in control states 28 and 7 respectively, and the observer is in control state s_1. The topmost cell is pointed to by the global variable H and the local variable h of Thread 1. Each cell has a *data value* field and a *next* field, the latter being a pointer to the next cell in the heap. In our example, there are three possible values that can be stored in a cell, namely *red* which means that the value is equal to the value of variable z_1 of the observer, *blue* which means that the value is equal to the value of variable z_2 of the observer, and *white* which means that it is an arbitrary value different from the above two. The topmost cell has a data value which is *white*. Finally, the figure shows the counter values (i.e. *ages*) of all the pointers (those of the pointer variables and those of the

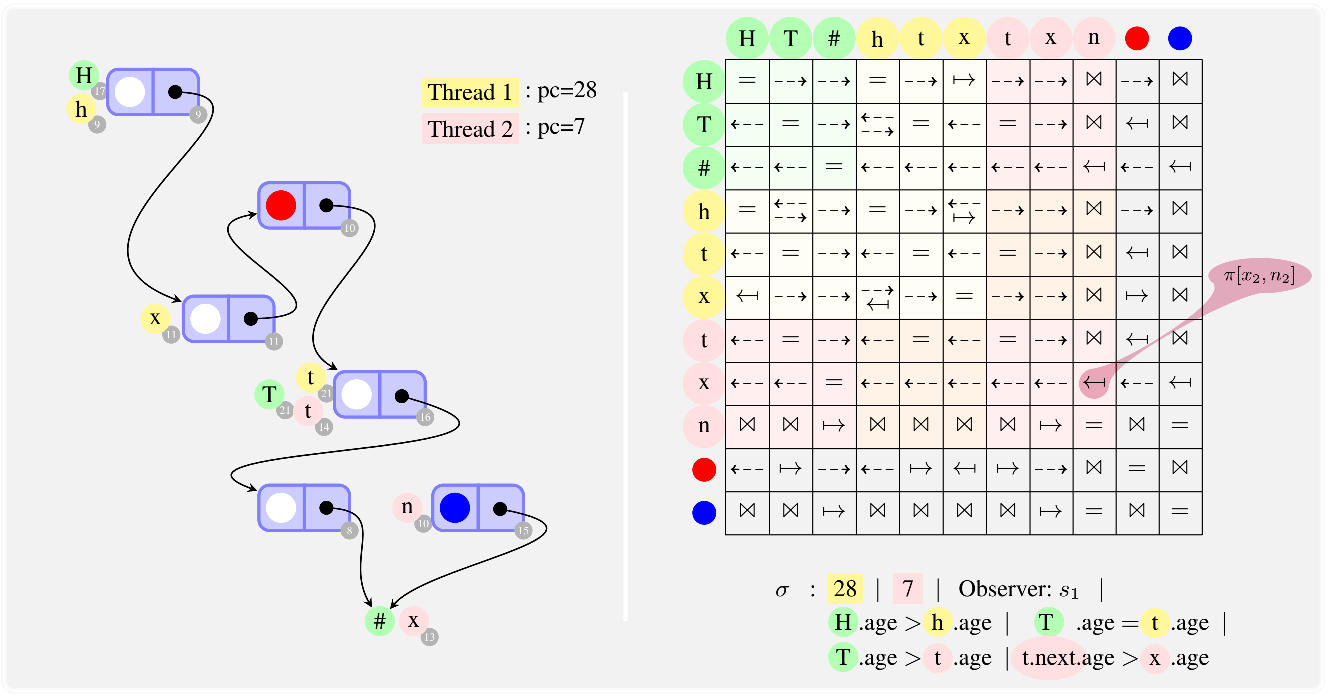

Thread 1 : pc=28
Thread 2 : pc=7
$\pi[x_2, n_2]$
σ : 28 | 7 | Observer: s_1 |
H .age > h .age | T .age = t .age |
T .age > t .age | t.next.age > x .age

Fig. 11. A concrete shape and its symbolic encoding.

next fields of the cells). For instance, the *next* pointer of the topmost cell has counter value 9, and the global variable H has counter value 17.

The right part of Figure 11 depicts a symbolic encoding that is satisfied by the given configurations. More precisely, our symbolic encoding consists of two parts, the first part, called a *joined shape constraint*, given in matrix form, describes the *shape* of the heap, while the second part, called *control formula*, denoted by σ, gives the control states of the observer and the active threads, together with the relations that hold between the pointer counters. We now introduce the needed concepts one by one, in a bottom-up manner. Let us fix two thread identifiers i_1 and i_2.

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).

Each row or column of the matrix in Figure 11 is labeled by a cell term. e.g., T, n, $\#$, etc. In particular, we use the red and blue circles, to denote the variables z_1 resp. z_2 of the observer.

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):

- $t_1 = t_2$: the cell terms t_1 and t_2 denote the same cell,
- $t_1 \mapsto t_2$: the `next` field of the cell denoted by t_1 denotes the cell denoted by t_2,
- $t_1 \dashrightarrow t_2$: 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,
- $t_1 \bowtie t_2$: 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.

Each cell in the matrix of Figure 11 contains a cell term. For instance, the cell pointed to by variable x of Thread 1 reaches in two or more steps the cell pointed to by variable variable t of Thread 2.

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. In Figure 11, the cell defined by the global variable T and the local variable h of Thread 1, indicates that *either* the cell pointed to by T is reachable from the cell pointed to by variable h of Thread 1, *or* the other way round. 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, both relations head[i].age$\simeq$Head.age and tail[i].ptr$\rightarrow$next.age$\simeq$next[i].age, for $\simeq \in \{<, =, >\}$.

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 cell variables $v_1, \ldots, v_m$ with 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

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

6.2 Verification Procedure

We compute a program invariant 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 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 $t \dashleftarrow x$ and $t \dashrightarrow x$, even if we know, via a simpler analysis, that no cycles are generated). 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. Like in [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 $\sigma[i_1, i_2, i_3] \equiv \sigma[i_1, i_2] \wedge \sigma_\bullet[i_2, i_3] \wedge \sigma_\circ[i_1, i_3]$ and $\phi[i_1, i_2, i_3] \equiv \phi[i_1, i_2] \wedge \phi_\bullet[i_2, i_3] \wedge \phi_\circ[i_1, i_3]$. 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 its postcondition $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$ in step ii. Finally, $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$ is projected back onto $\sigma'[i_1, i_2] \wedge \phi'[i_1, i_2]$ in step iii 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.

7 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 **??**). 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 **??**, 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 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.

8 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.

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

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

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.