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

One of the difficult challenges in software verification is to automate its application to algorithms with an unbounded number of threads that concurrently access a dynamically allocated shared state. Such algorithms are used, e.g., to implement data structures that can be accessed concurrently by a large number of threads, and are found in widely used libraries [14], such as the Intel Threading Building Blocks, or the java.util.concurrent package. They typically employ fine-grained synchronization, avoid locking, making them difficult to get correct and verify, as witnessed by, e.g., bugs in published algorithms [9, 18]. It is therefore important to develop efficient techniques for verifying conformance to an abstract specification of overall functionality. For instance, a concurrent implementation of a familiar data type abstraction, such as a queue, should be verified to conform to a simple abstract specification of a (sequential) queue.

In this paper, we present a technique for specifying and automatically verifying that a concurrent program conforms to an abstract specification of its functionality. Our technique addresses a number of challenges:

1. the abstract specification may be infinite-state (due to unbounded size of, e.g., a queue, and an unbounded data domain),
2. the number of threads in the program may be unbounded,
3. the program uses dynamically allocated memory, and
4. may explicitly manage memory allocation (not relying on garbage collection).

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

To cope with challenge 1, we present a novel technique for specifying programs by a class of automata, called *observers*. They have a finite-state control structure and a finite set of variables that can assume values from an unbounded domain. They observe the execution of a program, and are designed to accept precisely those executions that violate the intended specification, similarly as in the automata-theoretic approach to model checking [28]. 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 employ a data-independence argument, adapted from Wolper [32], which implies that it is sufficient to consider executions in which any data value is inserted at most once. This result allows us to succinctly specify data structures such as queues and stacks, using observers with typically less than 3 variables.

To cope with challenges 2 and 4, our verification technique must employ an abstraction which is sufficiently precise for reasoning about, e.g., techniques that avoid the ABA problem in the absence of garbage collection [16]. It is then necessary to correlate the local states of different threads with each other and with the global variables and the global heap. Our analysis therefore keeps track of possible correlations between any *pair* of concurrent threads, the global variables, and the global heap. To cope with challenge 3, we adapt a variant of the transitive closure logic by Bingham and Rakamarić [4] for reasoning about heaps, to our framework. This formalism tracks reachability properties between pairs of pointer variables, and we adapt it to our analysis, where pairs of threads are correlated.

The above techniques bring the needed precision for verification, at the price of significant state-space explosion, which mainly arises from reasoning about the dynamically allocated heap when pairs of threads must be correlated. We have therefore developed a novel optimization, built on the observation that it is enough to consider pairs of pointer variables separately from each other. This optimization tracks the possible relations between each pair of pointer variables separately, vaguely analogous to the use of DBMs used in reasoning about timed automata [8]. It makes our analysis efficient without sacrificing essential information.

We have implemented our specification and verification technique, and applied it to specifying and verifying concurrent implementations of stacks and queues. Our verification in fact establishes that the implementations are linearizable [15]. Our approach is able to automatically verify linearizability of some implementations of concurrent data structures with explicit memory management that were previously beyond the reach of automatic methods, and is significantly faster on a number of hard examples that have been reported in the literature.

In summary, the paper introduces a number of contributions, including

- novel techniques for specifying that a program conforms to an infinite-state specification, and for automatically verifying conformance to such specifications,

– A technique for making automated verification of programs with an unbounded number of threads, an unbounded heap, and explicit memory management, practical, using a novel optimization for analysis that tracks correlations between threads.

Together, the new contributions of our work results in an integrated technique that addresses the four challenges introduced in the beginning of this section. Experimental evaluation shows that our technique can fully automatically verify a range of concurrent implementations of common data structure implementations, such as queues, stacks, etc.

Outline. After the comparison with related work, we present the class of programs considered in Section 2. In Section 3, we define observers, and how to use them for specification. Section 4 presents the data-independence argument that allows observers to specify queues, stacks, and other 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, where pairs of concurrent threads are correlated. We report on experimental results in Section 6. Section 7 contains conclusions and directions for future work. In the appendix, we provide observers and prove that they precisely capture standard natural specifications of data-structures.

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 [13, 21, 22]. 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 [10, 7, 11, 31].

Amit et al [2] 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 [3] 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. [3] 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 [25] 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. 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 [26] 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. [12] and Kidd et al. [17]. 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. [23] continue the work of [3] by also considering an analysis that keeps track of correlations between threads. They strive to counter the state-space explosion that [3] 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 [23].

There are several works that apply different verification techniques to programs with a bounded number of threads, including the use of TVLA [33]. Several approaches produce decidability results under limited conditions [6], or techniques based on non-exhaustive testing [5] or state-space exploration [30] for a bounded number of threads.

2 Programs

We consider concurrent systems consisting of an arbitrary number of sequential threads that access shared global variables and a shared heap using 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, 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. Methods can be invoked by concurrently executing threads at arbitrary points in time. 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. A designated initialization method, which is executed once at the beginning of program execution, initializes the global variables and the heap.

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(&S,t,x)` atomically compares the values of `S` and `t`. If equal, it assigns the value of `x` to `S` and returns `TRUE`, otherwise, it leaves `S` unchanged and returns `FALSE`.

As an example, Figure 1 shows a version of the concurrent queue by Michael and Scott [19], which uses explicit memory management (i.e., does not rely on garbage collection). It manipulates heap cells of type `node`, each consisting of a field `val`, which carries a data value, and an extended pointer field `next`. An extended pointer field (of type `pointer_t`) consists of a pointer `ptr` to a `node` and an integer counter `age`. The `age` field is necessary in order to avoid the ABA problem in the absence of automated

garbage collection. We say that an extended pointer points to a node `n` to mean that its `ptr` field points to `n`. The program represents the queue by 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.

The queue can be accessed by an arbitrary number of threads, either by an enqueue `enq(d)` method, 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 26 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`.

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

   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,
                   ⟨n,next.age+1⟩))
10           break;
11       else
12         CAS(&Tail,tail,
                ⟨next.ptr,tail.age+1⟩);
13   }
14   CAS(&Tail,tail,⟨n,tail.age+1⟩);
15 }
```

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


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

Fig. 1. Michael & Scott's non-blocking queue [19].

3 Specification by Observers

In this section, we show how to specify the dynamic behavior of programs using observers. Our technique is able to specify a rich set of safety properties that constrain the ordering between various events during program execution.

To specify a correctness property, we instrument each method to generate *abstract events* at suitable points. We first define a (finite) set of *event types*. An *abstract event* is then a term of form $l(d_1, \ldots, d_n)$ where l is an event type and $d_1, \ldots, d_n$ are data values in $\mathbb{D}$. A *parameterized event* is a term of form $l(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are formal parameters. A command of a program can be instrumented to generate an abstract

event, in which the parameters are obtained by evaluating expressions over the current program variables. The generation can be conditional on some boolean expression.

We illustrate how to instrument the program of Figure 1 in order to specify that it is a linearizable implementation of a queue. Linearizability provides the illusion that each method invocation takes effect instantaneously at some point (called the *linearization point*) between method invocation and return. This means that in any behavior, the sequence of executed linearization points conforms to the behavior of a sequential queue. To specify this correctness condition, we instrument each method to generate an abstract event whenever a linearization point is passed. The abstract event is simply the name of the method including its data parameters and return values. For instance, line 12 of the `enq` method called with data value `d` is instrumented to generate the abstract event `enq(d)` when the `CAS` command succeeds. Generation of abstract events can be conditional. For instance, line 18 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 22).

When verifying linearizability, one must check that the instrumentation reflects the actual behavior of methods, i.e., that each method invocation generates exactly one abstract event which "says" what the method does. This can be done by standard sequential verification techniques.

After instrumentation, each execution of the instrumented program will generate a sequence of abstract events called a *trace*. Using an adaptation of the automata-theoretic approach [28], we now specify that each trace conforms to a property specified by an infinite-state automaton, called an *observer*. The 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 violates the intended specification.

Formally, an *observer* is a tuple $\mathcal{A} = \langle S, s_0, F, Z, C, \longrightarrow_{\mathcal{A}} \rangle$ where S is a finite set of *observer locations*, $s_0 \in S$ is the *initial location*, F is the set of *accepting locations*, Z is a finite set of *observer variables*, C is a finite set of *observer constants* in $\mathbb{D}$, and $\longrightarrow_{\mathcal{A}}$ is a finite set of *transitions*. The observer variables assume values in $\mathbb{D}$: initially, they can assume any value in $\mathbb{D}$. Each transition is a tuple of form $s \xrightarrow{l(\overline{p});g}_{\mathcal{A}} s'$ where $s, s' \in S$ 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 Z and constants C. Note that the values of observer variables are not updated in a transition.

An *observer configuration* is a pair $\langle s, \vartheta \rangle$, where $s \in S$ is an observer location, and $\vartheta : Z \mapsto \mathbb{D}$ maps each observer variable to a value in the data domain $\mathbb{D}$. An *observer step* is a triple $\langle s, \vartheta \rangle \xrightarrow{l(\overline{d})}_{\mathcal{A}} \langle s', \vartheta \rangle$ such that there is a transition $s \xrightarrow{l(\overline{p});g}_{\mathcal{A}} s'$ 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 $\langle s_0, \vartheta \rangle \xrightarrow{l_1(\overline{d}_1)}_{\mathcal{A}} \cdots \xrightarrow{l_n(\overline{d}_n)}_{\mathcal{A}} \langle s_n, \vartheta \rangle$ where s_0 is the initial observer location. The run is *accepting* if s_n is accepting. A trace σ is *accepted* by $\mathcal{A}$ if $\mathcal{A}$ has an accepting run on σ. Thus, a program satisfies the property represented by observer $\mathcal{A}$ if no trace of the program is accepted by $\mathcal{A}$.

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

Intuitively, an observer observes a trace, and accepts whenever the trace violates the intended specification. The data variables can assume arbitrary initial values. This allows observers to check properties that are universally quantified over all data values: if a trace violates such a property for some data values, the observer can nondeterministically choose these as initial values of its variables, and thereafter detect the violation when observing the trace. Observers can thus be used to give complete specifications of a set (safety properties). For instance, the observer in Figure 2 checks that all data values d satisfy the property that a $\texttt{delete}(d)$-event can be observed only if an earlier $\texttt{insert}(d)$-event has been observed.

4 Data Independence

Since variables of the observer can initially assume any values in the data domain, observers can specify properties that are universally quantified over data values, by checking whether the property can be violated for some values of the variables. In order to specify data structures such as 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 extend a data-independence argument, originating from Wolper [32], as follows.

The argument assumes that we partition all occurrences of data values in a trace into *input occurrences* and *output occurrences*. Formally, the partition can be arbitrary, but to make the argument work, input occurrences should typically be provided as input parameters in method invocations, whereas output occurrences should be returned by method invocations. In the program of Figure 1, input occurrences are parameters of $\texttt{enq}(\texttt{d})$ events, and output occurrences are parameters of $\texttt{deq}(\texttt{d})$ events. Note that the same data value can appear both in input occurrences and output occurrences.

Definition 1. *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 set of data values. The renaming of a trace σ with some renaming f, denoted $f(\sigma)$, is obtained by replacing each data value $\texttt{d}$ in σ by $f(\texttt{d})$.

Definition 2. *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 at least one differentiated trace $\sigma_d \in \Sigma$ such that $f(\sigma_d) = \sigma$ 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 only copied, but otherwise not manipulated. 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.*

It follows from Theorem 1 that an observer for a data-independent property need only consider differentiated traces: whenever a data value is input twice in a trace, the observer can stop checking (i.e., move to a non-accepting state), since the trace will anyway be ignored.

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 appendix, we provide a standard natural specification of a queue, and prove, in Lemma 1, that its differentiated traces are precisely those that satisfy the following four properties for all data values d_1 and d_2.
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 a finite set of control locations and one or two local variables which are non-deterministically initialized to arbitrary values. If the property is violated by some specific data values d_1 and d_2, then there is some run of the observer which leads to an accepting state. Lemma 1 also provides an analogous characterization of the differentiated traces of a stack. Figure 3 shows an observer for the *fifo* property.

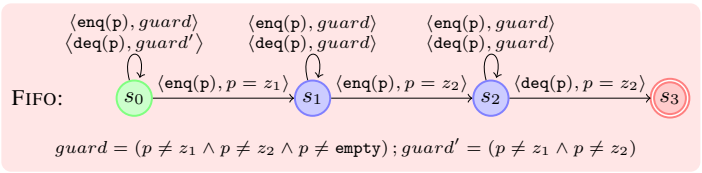

Fig. 3. An observer to check that *fifo* ordering is respected. All unmatched abstract events, for example $\langle \texttt{deq(p)}, p = z_1 \rangle$ at location s_1, send the observer to a sink state.

To verify that no trace of the program is accepted by $\mathcal{A}$, we form, as in the automata-theoretic approach [28], 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.

5 Verification by Shape Analysis

In this section, we describe our analysis for verifying that the cross-product of a program and a trace observer never reaches a configuration where the observer is in an accepting state. The analysis generates a formula that holds in the reachable configurations of this cross-product, and checks that the formula does not allow accepting observer locations.

5.1 Representation of Shape Formulas

The analysis must address the challenges of an unbounded data domain, an unbounded number of concurrently executing threads, and unbounded heap, and explicit memory management. As indicated in Section 1, the challenge of explicit memory management implies that the assertions generated by our analysis must be able to track correlations between pairs of threads. For instance, an analysis that only gives properties of single threads (i.e., a thread-modular analysis [3]) cannot express that, for the program in Figure 1, at most one thread is at any point in time at line 28 and about to deallocate the first node in the queue, and would spuriously report a memory error. Furthermore, it cannot express that at most one thread is holding some lock.

Our analysis must, thus, correlate different threads. We therefore manipulate formulas of form $\forall i_1, i_2.\ (i_1 \neq i_2 \Rightarrow \Phi[i_1, i_2])$ (abbreviated as $\forall i_1 \neq i_2.\ \Phi[i_1, i_2]$) where i_1 and i_2 range over thread identifiers, and $\Phi[i_1, i_2]$ relates the local configurations of threads i_1 and i_2 with each other, and with global variables, observer configuration, and heap. The formula states that $\Phi[i_1, i_2]$ holds for *any* two distinct executing threads i_1 and i_2.

In our analysis, we represent $\Phi[i_1, i_2]$ as a (typically large) disjunction of formulas of form $\sigma[i_1, i_2] \wedge \phi[i_1, i_2]$, where each $\sigma[i_1, i_2]$ is a so-called symbolic state, and $\phi[i_1, i_2]$ is a so-called shape formula. A *symbolic state* is a conjunction that contains

- the current control location of threads i_1 and i_2, and the observer,
- for each (live) data variable $d[i_j]$ in some thread i_j and each (live) observer variable z, either the conjunct $d[i_j] = z$ or its negation,
- for each pair of terms of form $t_1\texttt{.age}$ and $t_2\texttt{.age}$, that may be compared in a `CAS` statement, a conjunct of the form $t_1\texttt{.age} \simeq t_2\texttt{.age}$, where $\simeq \in \{<, =, >\}$. For instance, when analyzing the program in Figure 1, these conjuncts include $\texttt{head}[i_j]\texttt{.age} \simeq \texttt{Head.age}$ and $\texttt{tail}[i_j]\texttt{.ptr} \rightarrow \texttt{next.age} \simeq \texttt{next}[i_j]\texttt{.age}$ with $\simeq \in \{<, =, >\}$.

Intuitively, a symbolic state is a most precise description of the part of the state that does not concern the heap. It uses the appropriate constraints for each type of variable or field, i.e., equalities and their negations between data variables, and orderings between `age` fields of relevant pointers.

Let us now consider shape formulas. A shape formula should be an assertion about the program heap. Since the heap is unbounded, we must bound the information provided. We do this by adapting the predicate abstraction approach of [4], in the following way.

Let a *thread variable* be either i_1 or i_2. Our intention is that a shape formula records essential relationships between the pointer variables of the program. So, let a *cell term* be either a global pointer variable y or a term of form $x[i_j]$ for a local pointer variable x and a thread variable i_j. The term y denotes the cell pointed to by the global variable y, and $x[i_j]$ denotes the cell pointed to by the thread-i_j-local-copy of x. We also use the special cell terms `NULL`, `UNDEF`, and `FREE`. In addition, we need a mechanism for speaking about the heap cells that contain data that is equal to some observer variable. These heap cells might sit anywhere in the heap and need not be denoted by any pointer variable. If we do not constrain their relationship with other heap cells, we will not be able to verify, e.g., the *fifo* property of queues. We therefore introduce a set of *cell*

variables, denoted v, v_1, v_2, ..., which are also cell terms. Let $CT(i_1, i_2)$ denote the set of cell terms over threads i_1, i_2.

An *atomic heap constraint* is of one of the following forms:

- $t_1 = t_2$ means that 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 points to 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 $\{=, \mapsto, \leftarrow\!\!\shortmid, \dashrightarrow, \dashleftarrow, \bowtie\}$ of all shape relational symbols. We let $t = \texttt{NULL}$ denote that t is null, $t = \texttt{UNDEF}$ denote that t is undefined, and $t \mapsto \texttt{FREE}$ denote that t is unallocated.

In order to obtain an efficient and practical analysis, which does not lead to a severe explosion of formulas, we have found it necessary to use a compact representation in our shape formulas. We have therefore developed a novel representation, which is motivated by the observation that relationships between pairs of pointer variables are typically independent, and that it suffices to consider each pair of variables rather independently from the correlation between other pairs of variables.

So, define a *shape formula* over i_1, i_2 as a formula of form

$$\exists v_1, \ldots, v_m. \left[\psi[v_1, \ldots, v_m, z_1, \ldots, z_n] \ \wedge \bigwedge_{t_1, t_2 \in CT(i_1, i_2)} \pi[t_1, t_2] \right] ,$$

where $\psi[v_1, \ldots, v_m, z_1, \ldots, z_n]$ is a formula which states that all cell variables $v_1, \ldots, v_k$ denote mutually distinct cells, contains a conjunct of form $v_j = z_k$ or $v_j \neq z_k$ for each cell variable v_j and live observer variable z_k, and finally states that any other cell has a data value which is different from any observer variable. The second conjunct, which is called a *joined shape constraint*, contains for each pair of cell terms in $CT(i_1, i_2)$ a non-empty disjunction $\pi[t_1, t_2]$ of atomic heap predicates of form $t_1 \sim t_2$ for $\sim \in Pred$ (note that the disjunction of all $t_1 \sim t_2$ for $\sim \in Pred$ is equivalent to *true*). A joined shape constraint can be exponentially more concise than using a large disjunction of conjunctions, as outlined above, at the cost of some loss of precision.

We say that a joined shape constraint $\bigwedge_{t_1, t_2 \in CT(i_1, i_2)} \pi[t_1, t_2]$ is *closed* (under entailment) if for all terms t_1, t_2, and t_3 in $CT(i_1, i_3)$, every disjunct of $\pi[t_1, t_3]$ is consistent with a five-tuple of disjuncts chosen from $\pi[t_1, t_1]$, $\pi[t_1, t_2]$, $\pi[t_2, t_2]$, $\pi[t_2, t_3]$, and $\pi[t_3, t_3]$, respectively. Any joined shape constraint can be closed by a straightforward fixpoint procedure, analogous to that for DBMs [8]. For instance, let $\pi[x, z] = x \mapsto z$ and $\pi[y, z] = y \mapsto z \vee y \dashrightarrow z$ 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 only contain the disjuncts $x = y$, $x \bowtie y$, that are consistent with $x \mapsto z$ and $y \mapsto z$, together with $x \leftarrow\!\!\shortmid y$, $x \dashleftarrow y$, and $x \bowtie y$, that are consistent with $x \mapsto z$ and $x \dashrightarrow z$. In short, y cannot reach x, thus when closing, we remove $x \leftarrow\!\!\shortmid y$ and $x \dashleftarrow y$ from $\pi[x, y]$.

5.2 Analysis Procedure

Our analysis will generate a formula of form $\exists i_1 \neq i_2.\ \Phi[i_1, i_2]$, which is valid in all reachable states of the program. As described in previous subsection, $\Phi[i_1, i_2]$ is a (typically large) disjunction of formulas of form $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$, where $\sigma[i_1, i_2]$ is a symbolic state, and $\phi[i_1, i_2]$ is a shape formula. The formula $\Phi[i_1, i_2]$ is generated by a standard fixpoint procedure, starting from a formula that characterizes the initial states of the program.

We note that different symbolic states are mutually exclusive, and so the formula will contain (at most) one shape formula for each symbolic state. The fixpoint analysis will therefore consider each occurring symbolic state together with its corresponding shape formula, perform a postcondition computation that results in a set of possible successor combinations of symbolic states and shape formulas, and thereafter update the above formula by weakening the appropriate shape constraints if the corresponding symbolic state already occurs in $\Phi[i_1, i_2]$, otherwise a new disjunct is generated.

The postcondition computation for a formula of form $(\sigma[i_1, i_2] \wedge \phi[i_1, i_2])$ must consider two cases. Either a thread represented by one of the variables i_1 or i_2 performs a step, or some other (interfering) thread, which is not represented by either i_1 or i_2 performs a step. For the first case, we can compute the postcondition directly on the above disjunct only. For the second case, we should take into account the consequences of the possible relationships between the local state of the third thread (represented by i_3, say). For this, we must therefore consider all pairs of disjuncts of form $\sigma'[i_2, i_3] \wedge \phi'[i_2, i_3]$ and $\sigma''[i_3, i_1] \wedge \phi''[i_3, i_1]$, such that $\sigma[i_1, i_2] \wedge \sigma'[i_2, i_3] \wedge \sigma''[i_3, i_1]$ is consistent. For each such pair, we

1. generate the formula $\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3]$ where
$$\sigma[i_1, i_2, i_3] \equiv \sigma[i_1, i_2] \wedge \sigma'[i_2, i_3] \wedge \sigma''[i_3, i_1]$$
$$\phi[i_1, i_2, i_3] \equiv \phi[i_1, i_2] \wedge \phi'[i_2, i_3] \wedge \phi''[i_3, i_1]$$
whereafter we close $\phi[i_1, i_2, i_3]$ under entailment (in the same way as for joined shape formulas over two threads).
2. Then for each statement S of thread i_3 that can be executed when $\sigma[i_1, i_2, i_3]$ holds, we generate the postcondition of $\sigma[i_1, i_2, i_3] \wedge \phi[i_1, i_2, i_3]$ under S of i_3, as follows:
   - $\sigma[i_1, i_2, i_3]$ is updated to $\sigma'[i_1, i_2, i_3]$ in the standard way (by updating the possible values of control locations and data variables),
   - $\phi[i_1, i_2, i_3]$ is updated to $\phi'[i_1, i_2, i_3]$ by updating each conjunct $\pi[t_1, t_2]$ as follows:
     - remove all disjuncts that may be falsified by the step
     - add all disjuncts that may become true by the step
     - close the result under entailment

     Closing under entailment may result in more than one solution, hence we may obtain multiple variants of $\phi'[i_1, i_2, i_3]$.

   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 $t \in CT(i_1, i_2, i_3)$. First, all information about x is removed by setting every conjunct $\pi[x, t]$ and $\pi[t, x]$, $t \in CT(i_1, i_2, i_3)$, to Pred. Second, $\pi[x, y]$ is set to $y \mapsto x$ and $\pi[y, x]$ is set to $y \leftarrow\!\!\shortmid x$. Third, relationships about x that are inconsistent with the rest of the shape formula are pruned by closing the formula under entailment.

3. Finally, all the variants of the resulting formula $\sigma'[i_1, i_2, i_3] \wedge \phi'[i_1, i_2, i_3]$ are projected back onto $\sigma'[i_1, i_2] \wedge \phi'[i_1, i_2]$ by removing all information about variables of thread i_3, and subsequently joined.

Since the domain of symbolic states and the domain of shape formulae over a fixed number of cell terms are finite, the abstract domain of formulae $\forall i_1 \neq i_2 . \Phi[i_1, i_2]$ 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. Our analysis also implicitly checks for standard shape-related errors such as null/undefined pointer dereferencing (taking into account the known dangling pointers' dereferences [20]), 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 [24] 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 a violation of 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 [16].

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 [27]. However, the verification of versions that do not assume GC is, to the best of our knowledge, beyond the reach of [27] (since it does not correlate states of different threads). [23] verifies linearizability of concurrent implementations of sets, e.g., a lock-free `CAS`-based set [29] (verified in 2975s) of a comparable complexity to M&S's queue without GC (550s with our prototype). Basic

Table 1. Experimental Results.

		Conformance		Safety only	
Data-structure	**Observers**	**Time**	$\|C\|$	**Time**	$\|C\|$
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[19]	queue+	0.08s	1830	0.02s	488
Two-Locks Queue, no GC		0.73s	3460	0.13s	784
		vs 47s in [3]		*vs* 6162s/304s in [33]	
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[24]	stack+	0.23s	714	0.01s	78
		vs 0.09s in [27]			
Treiber's lock-free stack, no GC	stack+	2.28s	1535	0.10s	190
		vs 53s in [3]			
M&S's lock-free queue[19]	queue+	3.31s	3476	0.44s	594
		vs 3.36s in [27]			
M&S's lock-free queue, no GC	queue+	550s	53320	25s	6410
		vs o.o.m. in [3]		*vs* 727s/309s in [33]	

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

memory safety of M&S's queue and two-locks queue [19] without GC was also verified in [33], 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 [3], linearizability of the Treibers's stack (resp. two-locks queue [19]) is verified in 53s (resp. 47s). We achieve the same result in less than 3 seconds. A variant of M&S's queue without GC could not be successfully verified in [3] due to lack of memory.

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 numberd 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 [32] can be introduced into standard program analysis. In the current paper, we showed how to specify queues, and stacks. Other data structures, such as deques, can be specified in an analogous way.

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

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. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.

12. 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.
13. C. Flanagan and S. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Science of Computer Programming*, 71(2):89–109, 2008.
14. T. Harris, K. F. I. Pratt, and C. Purcell. Practical lock-free data structures.
15. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
16. IBM. System/370 principles of operation, 1983.
17. N. Kidd, T. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, 2011.
18. 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.
19. 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.
20. 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.
21. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proc. of PLDI'06*, pages 308–319. ACM, 2006.
22. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proc. of ICSE*, pages 386–396. IEEE, 2009.
23. 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.
24. R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Res. Ctr., 1986.
25. V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *Proc. of VMCAI*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.
26. V. Vafeiadis. Automatically proving linearizability. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 450–464. Springer, 2010.
27. V. Vafeiadis. Rgsep action inference. In *Proc. of VMCAI'10*, volume 5944 of *LNCS*, pages 345–361. Springer, 2010.
28. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS'86*, pages 332–344, June 1986.
29. M. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *Proc. of PLDI'08*, pages 125–135. ACM, 2008.
30. 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.
31. 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.
32. P. Wolper. Expressing interesting properties of programs in propositional temporal logic (extended abstract). In *Proc. of POPL'86*, pages 184–193, 1986.
33. E. Yahav and S. Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.

A Data-independence based regular specification using trace observers

We show in this section how to completely specifiy, using the observers introduced in Section 3, the sequential behaviors of queues and stacks operating over an *arbitrary (and possibly infinite) data domain* $\mathbb{D}$. Apart from the size of the data domain, queues and stacks allow counting and require therefore non-regular specifications in general. For this reason, we generalize a data-independence argument originating from Wolper [32]. We detail the approach for stacks and briefly mention how to adapt it for the case of queues. First, we recall the natural operational definition of a stack and explain how we define its behavior using the set of traces it generates. Then, we propose, using simple observers as defined in Section 3, 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 $\underline{\mathbb{D}}$ to mean $\mathbb{D} \setminus \{\texttt{empty}\}$.

A.1 Stacks and Their Functional Specification

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 $\texttt{in}(d)$ (respectively $\texttt{out}(d)$ and $\texttt{out}(\texttt{empty})$) to mean a $\texttt{push}(\texttt{d})$ (respectively $\texttt{pop}(\texttt{d})$ and $\texttt{pop}(\texttt{empty})$). The specification of a sequential stack is a strict subset of $(\Sigma_{i/o})^*$, where $\Sigma_{i/o} = \{\texttt{in}(d), \texttt{out}(d) \mid d \in \underline{\mathbb{D}}\} \cup \{\texttt{out}(\texttt{empty})\}$. We give in the following an operational and an observational characterization of the specification of a sequential stack and show their equivalence.

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 the transtions are labeled with $\Sigma_{i/o}$. More formally, T is a tuple $(\Sigma_{i/o}, (\underline{\mathbb{D}})^*, \{\epsilon\}, \rightarrow)$, where the empty word $\epsilon \in (\underline{\mathbb{D}})^*$ is the initial state, and the set of transitions $\rightarrow \subseteq (\underline{\mathbb{D}})^* \times \Sigma_{i/o} \times (\underline{\mathbb{D}})^*$ includes all transitions of the form: $\langle w, \texttt{in}(d), d \cdot w\rangle$, $\langle d \cdot w, \texttt{out}(d), w\rangle$, and $\langle \epsilon, \texttt{out}(\texttt{empty}), \epsilon\rangle$, where $d \in \underline{\mathbb{D}}$ and $w \in (\underline{\mathbb{D}})^*$. 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 $i : 0 \leq i < n$. We say that ρ is a valid stack run. A trace of T is any sequence $e_1 \cdots e_n$ such that there is a valid 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 valid 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 valid stack run on a differentiated trace that can be renamed (by forgetting the counter) into σ. The set φ^{op}_{stack} therefore satisfies definition 2 and is data-independent.

An important observation is that two data-independent sets of traces are equal iff their sets of diffirentiated traces coincide. This implies that it is enough to show, for

a data independent set of traces, that its set of differentiated traces equals the set of differentiated stack traces in order to conclude that the set coincides with the set of stack traces. We write in the following $\varphi^{op}_{diff,stack}$ to mean the set if differentiated traces in φ^{op}_{stack}.

Observational specification of a stack. We propose another specification for differentiated stack traces, written $\varphi^{obs}_{diff,stack}$ which characterizes the set of differentiated stack traces as exactly those differentiated traces that are not accepted by any of the observers ob_{crea} (Figure 4), ob_{loss} (Figure 5), ob_{dupl} (Figure 6), or ob_{lifo} (Figure 8). Intuitively, a differentiated trace that is not accepted:

- by the observer (ob_{crea}) is guatanteed to verify that every output of a data value corresponds to an earlier input of the same data value (i.e., data cannot be created);
- by the observer (ob_{dupl}) is guaranteed to verify that every input value can be output at most once (i.e., data cannot be duplicated);
- by the observer (ob_{loss}) is guaranteed to verify that $\mathtt{out}(\mathtt{empty})$ may appear only if every preceding input has been output (i.e., data cannot be lost);
- by the observer (φ_{lifo}) is guaranteed to not violate the LIFO ordering.

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 $\rightarrow$ by the smallest subset of $\left((\underline{\mathbb{D}})^* \times \Sigma_{i/o} \times (\underline{\mathbb{D}})^*\right)$ that includes, for every $d \in \underline{\mathbb{D}}$ and $w \in (\underline{\mathbb{D}})^*$, all transitions of the form: $\langle w, \mathtt{in}(d), w \cdot d\rangle$, $\langle d \cdot w, \mathtt{out}(d), w\rangle$, and $\langle \epsilon, \mathtt{out}(\mathtt{empty}), \epsilon\rangle$. $\varphi^{op}_{diff,queue}$ is the restriction of φ^{op}_{queue} to the set of differentiated traces. The observational specification $\varphi^{obs}_{diff,queue}$ is the largest set of differentiated traces that does not contain any trace accepted by ob_{crea}, ob_{loss}, ob_{dupl}, or ob_{fifo} (Figure 7).

Operational and observational functional specifications coincide. Lemma (1) shows that the differentiated operational specification of a stack or a queue, coincides with the corresponding differential observational one.

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

Proof. Recall the claim only concerns differentiated traces. We present the proof for a stack. The case of queues will be briefly discussed at the end. We will make use of two properties that hold for any valid run $\rho = w_0 e_1 w_1 e_2 \cdots e_n w_n$ of a stack.

- The *counting property* of a stack. Given a letter a in some alphabet A, and a finite word $w = a_0 \cdots a_n$ in A^*, we denote the cardinality of $\{i |\, i : 0 \leq i \leq n \wedge a = a_i\}$ by $(a)^{\#}_{w}$. We can easily show by induction on the lenght of ρ that for any $d \in \underline{\mathbb{D}}$ and $i : 0 \leq i \leq n$, $(d)^{\#}_{w_i}$ is equal to the difference $(\mathtt{in}(d))^{\#}_{(e_1 \cdots e_i)} - (\mathtt{out}(d))^{\#}_{(e_1 \cdots e_i)}$.
- The *ordering property* of a stack. Using the counting property and an induction on the length of ρ, we can show the following. For any two data values d, d' such that there are $i, j : 1 \leq i < j \leq n$ with $e_i = \mathtt{in}(d)$ and $e_j = \mathtt{in}(d')$, then

for every $k : j \leq k \leq n$ s.t. $\forall l : 1 \leq l \leq k.\ e_l \notin \{\texttt{out}(d), \texttt{out}(d')\}$ (i.e., if d is input before d' and before position k and neither of d, d' is output), then $w_k \in (\underline{\mathbb{D}} \setminus \{d, d'\})^* \cdot d' \cdot (\underline{\mathbb{D}} \setminus \{d, d'\})^* \cdot d \cdot (\underline{\mathbb{D}} \setminus \{d, d'\})^*$.

We proceed by inclusion in both directions in order to show the equality:

- $\varphi^{op}_{diff,stack} \subseteq \varphi^{obs}_{diff,stack}$. This direction is simple. Let $\rho = w_0 e_1 \cdots e_n w_n$ be a shortest run giving a trace $\sigma = e_1 \dots e_n$ in $\varphi^{op}_{diff,stack}$ accepted by one of the observers ob_{crea}, ob_{loss}, ob_{dupl}, or ob_{lifo} for some data values.
  1. σ cannot be accepted by ob_{crea}. Suppose it was the case. Since ρ is assumed to be a shortest run, e_n must be the $\texttt{out}(d)$ that labels the last transition in the observer. The fact that σ would be accepted by the observer implies there would be a data value d for which the self loop on s_0 of the observer contains no $\texttt{in}(d)$. By the counting property, $d \notin w_n$. Yet $e_n = \texttt{out}(d)$ requires w_n to be of the form $d \cdot w$.
  2. Similar to the above, σ cannot be accepted by ob_{loss} because the counting property would imply that w_{n-1} contains a d, yet $\texttt{out}(emptyEvent)$ requires $w_{n-1} = \epsilon$.
  3. Similar to the above, σ cannot be accepted by ob_{dupl} because the counting property would require w_{n-1} to contain no occurences of d yet $\texttt{out}(d)$ requires w_{n-1} to be of the form $d \cdot w$.
  4. σ cannot be accepted by ob_{lifo} because, using the ordering property of a stack on the run ρ, we can show that $w_{n-1} \in (\underline{\mathbb{D}} \setminus \{d_1, d_2\})^* \cdot d_2 \cdot (\underline{\mathbb{D}} \setminus \{d_1, d_2\})^* \cdot d_1 \cdot (\underline{\mathbb{D}} \setminus \{d_1, d_2\})^*$. But for $\texttt{out}(d_1)$ to succeed, w_{n-1} needs to be of the form $d_1 \cdot w$.
- $\varphi^{op}_{diff,stack} \supseteq \varphi^{obs}_{diff,stack}$. Suppose $\sigma = e_1 \dots e_{n+1}$ in $\varphi^{obs}_{diff,stack}$ is a shortest trace not in $\varphi^{op}_{diff,stack}$. Hence, there is a valid stack run $\rho = w_0 e_1 \cdots e_n w_n$, but there is no w_{n+1} such that $\rho' = w_0 e_1 \cdots e_{n+1} w_{n+1}$ would be a valid stack run.
  1. e_{n+1} cannot be $\texttt{in}(d)$ because then it would be enough to choose $w_{n+1} = d \cdot w_n$ to get σ in $\varphi^{op}_{diff,stack}$.
  2. if $e_{n+1} = \texttt{out}(\texttt{empty})$, then $w_n \neq \epsilon$ as otherwise σ would be in $\varphi^{op}_{diff,stack}$. 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 $i : 1 \leq i \leq n$, but $\forall j : 1 \leq j \leq n.\ e_j \neq \texttt{out}(d)$. Hence σ should have been accepted by ob_{loss}, and therefore not in $\varphi^{obs}_{diff,stack}$,
  3. if $e_{n+1} = \texttt{out}(d)$ for some data value d, there are three cases:
     (a) If $\forall i : 1 \leq i \leq n.\ e_i \neq \texttt{in}(d)$ then σ should have been accepted by ob_{crea} and therefore not in $\varphi^{obs}_{diff,stack}$
     (b) If $\exists i, j : 1 \leq i, j \leq n$ with $e_i = \texttt{in}(d)$ and and $e_j = \texttt{out}(d)$. The counting property on the valid stack run ρ implies $i < j$. The trace σ should have been accepted by ob_{dupl}
     (c) there is a $e_i = \texttt{in}(d)$ for some $i \leq n$, and there are no $e_j = \texttt{out}(d)$ for $i < j \leq n$. We have that $w_n = w \cdot d \cdot w'$ with $w = d' \cdot w''$ and $d \neq d'$ as otherwise ρ could be extended into a valid stack run. Using the counting property (the run is valid up to n), there must be a $e_k = \texttt{in}(d')$ with $k : 1 \leq k \leq n$ and without any $\texttt{out}(d')$ in the run up to n. If $i < k$ the trace should have been accepted by ob_{lifo}. If $k < i$ we

use the ordering property to deduce that w_n should be in the language $(\underline{\mathbb{D}} \setminus \{d, d'\})^* \cdot d \cdot (\underline{\mathbb{D}} \setminus \{d, d'\})^* \cdot d' \cdot (\underline{\mathbb{D}} \setminus \{d_1, d_2\})^*$ which contradicts that $w_n = d' \cdot w'' \cdot d \cdot w'$.

For the queue case, we make use of the same counting property and modify the ordering property to reflect the FIFO ordering (instead of the LIFO one for a stack). The other modifications are straightforward.

We give trace observers for each property reported in Section 6.

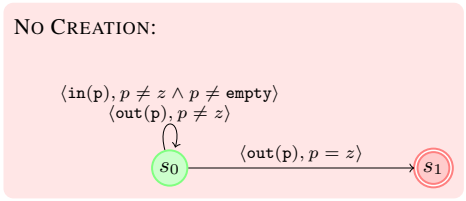

Fig. 4. A trace observer for checking that no data-value can be extracted if it has not been inserted. The variable z is an observer variable, and `empty` in an observer constant.

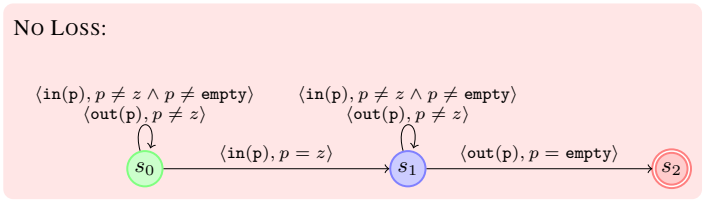

Fig. 5. A trace observer for checking that an inserted value has to be extracted before the data-structure is declared empty. The variable z is an observer variable, and `empty` in an observer constant.

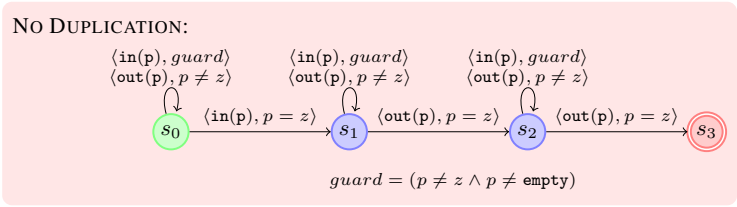

Fig. 6. A trace observer for checking that no once-inserted data value can be extracted twice. The variable z is an observer variable, and `empty` in an observer constant.

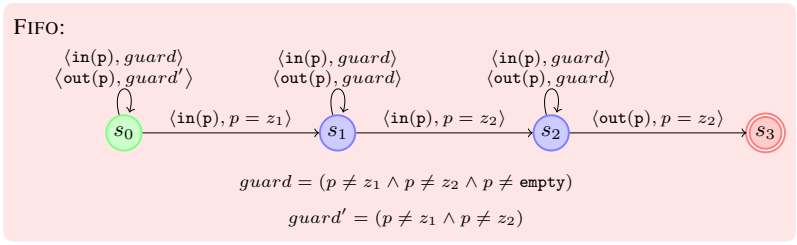

Fig. 7. An observer for detecting violations of the first inserted first extracted ordering. The initial state is s_0 and $\{s_3\}$ is the set of final states. The variables z_1, z_2 are observer variables, and `empty` in an observer constant.

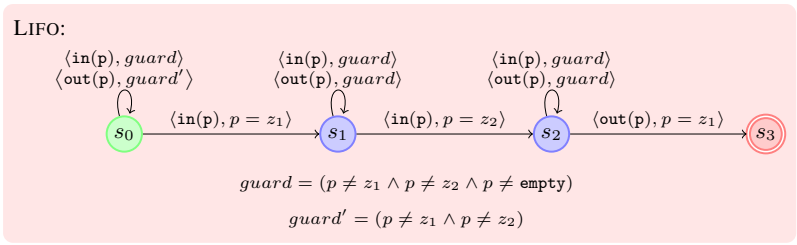

Fig. 8. An observer for detecting violations of the last inserted first extracted ordering. The initial state is s_0 and $\{s_3\}$ is the set of final states. The variables z_1, z_2 are observer variables, and `empty` in an observer constant.

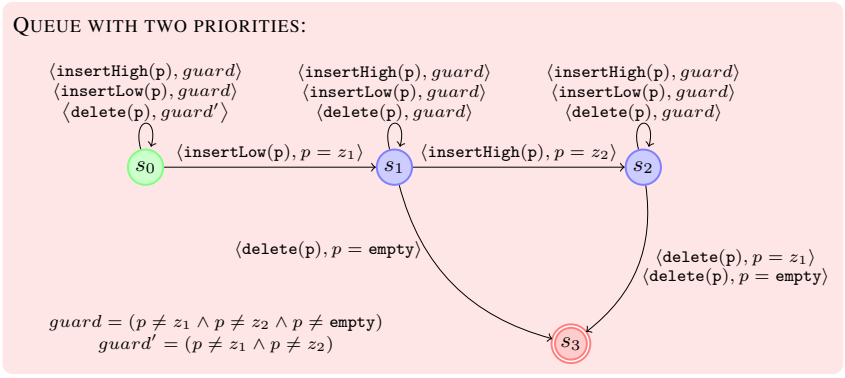

Fig. 9. A trace observer for checking that a low priority data value can not be dequeued if there is a high priority data value in the priority queue. The variables z_1, z_2 are observer variables, and `empty` in an observer constant.