Electronic Notes in Theoretical Computer Science 89 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 14 pages

Automatically Verifying Concurrent Queue Algorithms

Eran Yahav Mooly Sagiv

School of Computer Science,
Tel-Aviv University, Tel-Aviv, Israel
{*yahave,msagiv*}*@post.tau.ac.il*

Abstract

Concurrent FIFO queues are a common component of concurrent systems. Using a single shared lock to prevent concurrent manipulations of queue contents reduces system concurrency. Therefore, many algorithms were suggested to increase concurrency while maintaining the correctness of queue manipulations. This paper shows how to automatically verify partial correctness of concurrent FIFO queue algorithms using existing abstract interpretation techniques. In particular, we verify all the safety properties originally specified for two concurrent queue algorithms without imposing an a priori bound on the number of allocated objects and threads.

1 Introduction

Concurrent FIFO queues are widely used in concurrent systems. Queues are used in scheduling mechanisms, and as a basis of many concurrent algorithms. Concurrent manipulation of a shared queue requires synchronization to guarantee consistent results. An ill synchronized concurrent queue may be subject to read-write conflicts, write-write conflicts, or both.

Many algorithms were suggested to increase concurrency while maintaining the correctness of queue manipulations [9,15,10,18,14]. The algorithms in [9,15,10,14] are given without a formal proof of correctness, and [18] provides a manual formal proof.

In this paper, we show how the TVLA/3VMC framework can be applied to automatically verify partial correctness of non-trivial concurrent queue algorithms. We focus on the non-blocking queue and two-lock queue algorithms presented in [9]. A Java-like code for the queue implementations is given in Fig. 1. To emulate the intention of [9], our programming model diverges from Java by assuming sequentially consistent memory model and supporting a free operation.

One of the attractive features of TVLA/3VMC is that it provides an expressive formalism for expressing concrete semantics, and includes automatic features for

deriving finite abstract representations. This framework is conservative, however, it may and sometimes does fail to verify a property although the property holds on all execution paths of the program. Therefore, it is not clear that applying such a method to the concurrent queue algorithms will produce useful results.

The imprecision in TVLA/3VMC occurs due to the fact that the system abstracts many of the dynamically allocated objects and threads into a single summary representation. While this often results with a loss of precision, when we succeed, the property is guaranteed to hold for the program with any number of allocated objects and threads. Furthermore, even when the number of allocated threads is bounded, verifying the abstracted version may mitigate the state explosion problem when the bound on the number of threads is large.

It is worth noting that the fact that the formalism of TVLA/3VMC supports general first-order logic, as opposed to propositional logic used in model-checking, allows one to naturally define the behavior of heap-manipulating programs.

Main Contributions This is a case study showing how the TLVA/3VMC system of [20,12] is used to verify properties of the non-trivial concurrent queue algorithms presented in [9].

Related Work Das, Dill, and Park [5] have used predicate abstraction to verify the properties of a cache coherence algorithm and a concurrent garbage-collection algorithm. The garbage collection algorithm was verified in the presence of a single mutator thread executing concurrently with the collector.

Many approaches were proposed to handle verification of unbounded data structures. Traditional approaches consist of manually abstracting the data-structure into a simple finite state machine representing the states of the data-structure that are relevant to the verification problem (e.g., [16,17]). Other, more recent approaches, use a combination of theorem-proving and model-checking techniques to automatically construct such abstractions [1,2,3].

This case-study differs from our previous work in [20] in three aspects: (i) we verify all the safety properties for the non-blocking queue and two-lock queue; (ii) verification is performed on a model of the original program; (iii) we only use standard refinement of the abstraction (instrumentation predicates such as sharing and reachability) and not hand-crafted abstraction for the specific programs.

Limitations Since our tool does not apply any partial-order reductions and does not attempt to decrease the level of interleaving, it is currently limited to small concurrent programs or to ones that are well-synchronized. This is due to the worst-case complexity of our algorithm which is doubly exponential in the number of labels.

A fundamental question in program analysis is how to predict the precision of a given analysis on a given program. In principle, this is a hard question, we note that the abstraction in TVLA/3VMC significantly loses information when arbitrary arithmetic operations on integer variables are performed which affect the safety of the algorithm.

2 Concurrent Queue Algorithms

In this section, we present the concurrent queue algorithms and the correctness properties we will verify for these algorithms.

2.1 Non-Blocking Queue

Java-like pseudo-code for the non-blocking queue algorithm is shown in Fig. 1(a). The queue uses an underlying singly-linked list which is pointed by two reference variables — Head and Tail, pointing to the head and tail of the queue correspondingly. The list always contains a dummy item at its head to avoid degenerate cases.

The algorithm is based on iterated attempts of a thread to perform a queue operation without being interrupted by other threads. A thread operates on shared-variables only using the compare-and-swap (CAS) primitive which allows it to atomically observe possible updates by other threads and apply its own update when the value of the shared variable was not updated by other threads.

The CAS primitive takes 3 arguments — an address, an expected value, and a new value, it then atomically compares the address to the expected value, and if the values are equal updates the address to contain the new value. If the address value is not equal to the expected value, no update is applied.

CAS-based algorithms may suffer from the "ABA" problem [9] in which a sequence of read-modify-CAS results with a swap when it shouldn't. This happens when a thread t_1 reads a value A of a shared variable, computes a new value and preforms a CAS. Meanwhile, another thread t_2 changes the value of the shared variable from A to B and back to A. In order to avoid this problem, each reference variable is augmented with a modification counter and shared references are only updated through the CAS primitive which increments the value of the modification counter. This could have been modeled in Java by adding a wrapper class which contains a reference and an unsigned integer counter. To simplify the exposition of our figures, we have added a primitive type that consists of a reference-value `ref` and an integer value `count` for the modification counter. All reference operations that use only the reference name apply to both components, for example, the assignment at label e_5 assigns the values of `this.Tail.ref` and `this.Tail.count` to `tail.ref` and `tail.count` correspondingly. When we specifically update a single component of the reference variable, we state that explicitly as at label d_6 which performs a comparison of the `ref` component of two reference variables.

2.2 Two-Lock Queue

Fig. 1(b) shows a Java-like code for the two-lock queue algorithm. This algorithm also uses an underlying linked-list, and uses a dummy item at the list head to simplify special cases. The algorithm uses a separate head lock and tail lock to separate synchronization of enqueueing and dequeueing threads.

```
// Non Blocking Queue
class NonBlockingQueue {
    private QueueItem Head;
    private QueueItem Tail;
    ...
public NonBlockingQueue() {
  node = new QueueItem()
  node.next.ref = NULL
  this.Head = this.Tail = node
}

public void enqueue(Object value) {
e1   node = new QueueItem(value);
e2   node.value = value;
e3   node.next.ref = NULL;
e4   while(true) { //Keep trying until done
e5     tail = this.Tail;
e6     next = tail.ref.next;
e7     if (tail == this.Tail) {
e8       if (next.ref == NULL) {
e9         if CAS(tail.ref.next, next,
             <node, next.count+1>); {
e10          break // enqueue done
e11        }
e12      } else {
e13          CAS(this.Tail, tail,
               <next.ref, tail.count+1>);
e14      }
e15    }
e16  }
e17  CAS(this.Tail, tail,
       <node, tail.count+1>);
e18 }

public Object dequeue() {
    Object result = null;
d1   while(true) {
d2     head = this.Head;
d3     tail = this.Tail;
d4     next = head.next;
d5     if (head == this.Head) {
d6       if (head.ref == tail.ref) {
d7         if (next.ref == NULL) {//is empty?
d8           return result;
d9         }
d10        CAS(this.Tail, tail,
             <next.ref, tail.count+1>);
d11      } else { //No need to deal with Tail
d12        result = next.ref.value;
d13        if CAS(this.Head, head,
             <next.ref, head.count+1>); {
d14          break; // dequeue done
d15        }
d16      }
d17    }
d18  }
d19 free(head.ref);
d20 return result;
d21 }
```

(a)

```
// TwoLockQueue.java
class TwoLockQueue {
    private QueueItem head;
    private QueueItem tail;
    private Object headLock;
    private Object tailLock;
    ...
public TwoLockQueue() {
  node = new QueueItem();
  node.next = null;
  this.head = this.hail = node;
}

public void enqueue(Object value) {
lp1    QueueItem x_i =
         new QueueItem(value);
lp2    synchronize(tailLock) {
lp3        tail.next = x_i;
lp4        tail = x_i;
lp5    }
lp6  }

public Object dequeue() {
       Object x_d;
lt1    synchronized(headLock) {
lt2        QueueItem node = this.head;
lt3        QueueItem new_head =
             this.head.next;
lt4        if (new_head != null) {
lt5          x_d = new_head.value;
lt6          new_head = first;
lt7          new_head.value = null;
lt8          free(node);
           }
lt9    }
lt10   return x_d;
lt11 }
}
```

(b)

```
// QueueItem.java
class QueueItem {
    public QueueItem next;
    public Object value;
    ...
}
```

(c)

Fig. 1. Java-like pseudo-code for (a) non-blocking queue, (b) two-lock queue, (c) queue-item.

2.3 Correctness of Algorithms

The correctness of the queue algorithms in [9] is established by an informal proof. Safety of the algorithm is shown by induction, proving that the following properties are satisfied by the algorithm:

P1 The linked list is always connected.

P2 Nodes are only inserted after the last node of the linked list.

P3 Nodes are only deleted from the beginning of the linked list.

P4 *Head* always points to the first node in the linked list.

P5 *Tail* always points to a node in the linked list.

In the following sections, we formally state these claims, and automatically verify them using TVLA/3VMC.

3 Vanilla Verification Attempt

In this section, we describe the basic steps required to verify the concurrent queue algorithms using TVLA/3VMC.

3.1 Representing Program Configurations using First-Order Logical Structures

First-order logical structures provide a natural formalism for representing the global state of a heap-manipulating program — individuals of the first-order structure correspond to heap-allocated objects, properties of objects are represented using unary predicates, and relationships between objects using binary predicates. It is also possible to use first-order logical structures to model non heap-allocated objects, as well as enforce a typing mechanism on objects by using a unary predicate $is\_T(v)$ to denote objects of type T.

Below, we show how this is done for the concurrent queue algorithms.

A *program configuration* encodes a program's global state, which consists of: (i) a global store, (ii) the program-location of every thread, and (iii) the status of locks and threads, e.g., if a thread is holding a lock. For every analyzed program, we assume that there is a set of predicate symbols P, each with fixed arity. Formally, a *program configuration* is a 2-valued logical structure $C^\natural = \langle U^\natural, \iota^\natural \rangle$, where

- $U^\natural$ is the potentially infinite universe of individuals. Each individual in $U^\natural$ represents a heap-allocated object (some of which may represent the threads of the program, and the configuration may also contain an infinite number of individuals representing the unsigned integers).
- $\iota^\natural$ is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota^\natural(p) \colon {U^\natural}^k \to \{0, 1\}$.

In this paper, we use the *natural* symbol ($\natural$) to denote entities of the concrete domain.

Predicates	**Intended Meaning**
$eq(v_1, v_2)$	v_1 equals to v_2
$is\_T(v)$	v is an object of type T
$\{rv[fld](o_1, o_2) \colon fld \in Fields\}$	field fld of the object o_1 points to the object o_2
$\{iv[fld](o_1, o_2) \colon fld \in Fields\}$	integer value of field fld of the object o_1 is o_2
$\{at[lab](t) \colon lab \in Labels\}$	thread t is at label lab
$heldBy(l, t)$	the lock l is held by the thread t
$blocked(t, l)$	the thread t is blocked on the lock l
$zero(n)$	the individual n represents zero
$succ(n_1, n_2)$	n_2 is the successor of n_1

Table 1
Predicates for the semantics of a Java fragment.

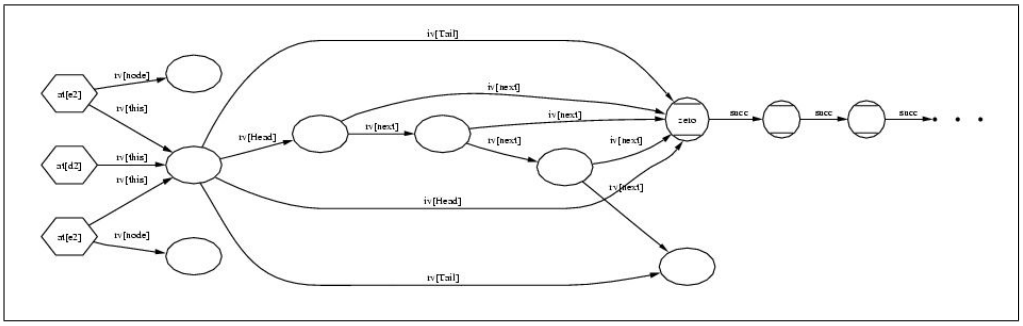

Fig. 2. A concrete configuration $C_2^{\natural}$ with two enqueueing and one dequeueing threads.

Usually, not all logical structures represent valid program configurations, therefore TVLA/3VMC allows the programmer to introduce integrity constraints specified as FO^{TC} (first order-logic with transitive closure) formulae [12]. The integrity constraints for integers are simply the Peano axioms encoded using FO^{TC} formulae.

Table 1 presents some of the predicates used to analyze the example programs. Predicates in the table are written in a generic way and can be applied to analyze different Java programs by modifying the set of labels and fields.

The non-blocking queue algorithm uses unsigned integer values as reference time-stamps. To allow fields of integer values we introduce objects of type unsigned-integer, and a binary predicate $iv[fld](v_1, v_2)$ that represents the integer value of a field by relating an object v_1 to an individual representing an integer value v_2.

It is also possible to support arbitrary arithmetic operations on integers, however, the abstraction presented in Sec. 4 is not precise enough to provide useful results when the verified property depends on the result of such operations.

In this paper, program configurations are depicted as directed graphs. Each individual of the universe is displayed as a node — objects of type thread are presented as hexagon nodes, objects representing unsigned integers are presented as circles with straight margins, round nodes represent objects of other types which are not

	Property	**Property Formula**
P1	tail reachable from head	$\forall q : nbq, v_t.rv[Tail](q, v_t)$ $\rightarrow \exists v_h.rv[Head](q, v_h) \wedge rv[next]^*(v_h, v_t)$
P2	insert after last	$\forall q : nbq, t_i : thread, v_i, v_t.at[e_{18}](t_i) \wedge rv[node](t_i, v_i) \wedge rv[tail](t_i, v_t)$ $\wedge rv[this](t_i, q) \rightarrow rv[next](v_t, v_i) \wedge rv[Tail](q, v_i)$
P3	delete first	$\forall q : nbq, t_d : thread, v_d, v_h.at[d_{19}](t_d) \wedge rv[head](t_d, v_d)$ $\wedge rv[this](t_d, q) \wedge rv[Head](q, v_h) \rightarrow rv[next](v_d, v_h)$
P4	head first	$\neg\exists q : nbq, v, u.rv[Head](q, v) \wedge rv[next](u, v)$
P5	tail exists	$\forall q : nbq.\exists v.rv[Tail](q, v)$

Table 2
Safety properties for non-blocking queue algorithm.

distinguished for ease of presentation. The name of a unary predicate, which is not a type predicate, which holds for an individual (node) is drawn inside the node. A binary predicate $p(u_1, u_2)$ which evaluates to 1 is drawn as directed edge from u_1 to u_2 labelled with the predicate symbol.

Example 3.1 The configuration $C_2^\natural$ shown in Fig. 2 corresponds to a global state of the non-blocking queue program with 3 threads: two enqueueing threads and a single dequeueing thread. The two enqueueing threads are at label e_2 and have just allocated new nodes to be enqueued, each enqueueing thread refers to its node by its `node` field.

All threads in the example use a single shared queue containing 4 items (including the dummy item). The integer values of the fields `Head` and `Tail` in this configuration are both 0. For brevity, predicate $eq(v_1, v_2)$ is not shown.

TVLA/3VMC allows to define a small-step operational semantics. The meaning of a program is defined as a transition-system, consisting of labels and actions. Informally, an action consists of a *precondition* under which the action is *enabled*, and a set of predicate-update formulae which determine the values of predicates in successor configurations. Actions may also create or remove individuals of the universe [20,12]. Supplemental information on actions is available from [19].

3.2 Safety

The first step in verifying the properties of Sec. 2.3 in TVLA/3VMC is to formulate them in FO^{TC} using the predicates defined in Table 1. In Table 2 these formulae are given for the non-blocking queue algorithm. The formulation of these properties for the two-lock queue only differs in label names. For each property defined informally in Sec. 2.3, we provide a corresponding formula in FO^{TC}.

In the table we use the shorthand notation $\forall v\colon type.\varphi \triangleq \forall v.is\_type(v) \rightarrow \varphi$. For brevity, we also use the shorthand nbq to stand for `NonBlockingQueue`.

Formula P1 uses transitive reachability from `Tail` and `Head` to require that each object that is reachable from the queue tail (including the tail node itself) is

also reachable from the queue head – thus the queue is always connected. Note that requirement P5 guarantees that a tail element always exists. Formula P2 uses the (program) location predicate $at[e_{18}](t)$ in order to check the requirement only at the end of an insertion operation, when it is meaningful. In this formula, we treat the local variable node as a field of the thread object. Formula P3 similarly uses the location predicate $at[d_{19}](t)$ to bind the requirement with the end of a deletion operation. Formula P4 simply requires that there is no queue element u such that it precedes the head of the queue. Finally, formula P5 requires that a tail element exists.

3.3 Abstraction

In this section, we present a conservative abstract semantics [4] abstracting the concrete semantics of Sec. 3.1.

Abstract Configurations We conservatively represent multiple concrete program configurations using a single logical structure with an extra truth-value $1/2$ which denotes values which may be 1 and may be 0. We allow an abstract configuration to include *summary nodes*, i.e., individuals that represent one or more individuals in a represented concrete configuration. Technically, a summary node u has $\iota(eq(u,u)) = 1/2$.

Formally, an *abstract configuration* is a 3-valued logical structure $C = \langle U, \iota \rangle$ where:

- U is the potentially infinite universe of the 3-valued structure. Each individual in U represents possibly many objects.
- ι is the interpretation function mapping predicates to their truth-value in the structure, i.e., for every predicate $p \in P$ of arity k, $\iota(p)\colon U^k \to \{0, 1/2, 1\}$.

Canonic Abstraction We now formally define how configurations are represented using abstract configurations. The idea is that each individual from the (concrete) configuration is mapped into an individual in the abstract configuration. More generally, it is possible to map individuals from an abstract configuration into an individual in another less precise abstract configuration. The latter fact is important for our abstract transformer.

Formally, let $C = \langle U, \iota \rangle$ and $C' = \langle U', \iota' \rangle$ be abstract configurations. A function $f\colon U \to U'$ such that f is surjective is said to *embed C into C'* if for each predicate p of arity k, and for each $u_1, \ldots, u_k \in U$ one of the following holds:

$$\iota(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \text{ or } \iota'(p(f(u_1), \ldots, f(u_k))) = 1/2$$

One way of creating an embedding function f is by using *canonic abstraction*. Canonic abstraction maps concrete individuals to an abstract individual based on the values of the individuals' unary predicates. All individuals having the same values for unary predicate symbols are mapped by f to the same abstract individual. Canonic abstraction guarantees that the resulting abstract configuration is of

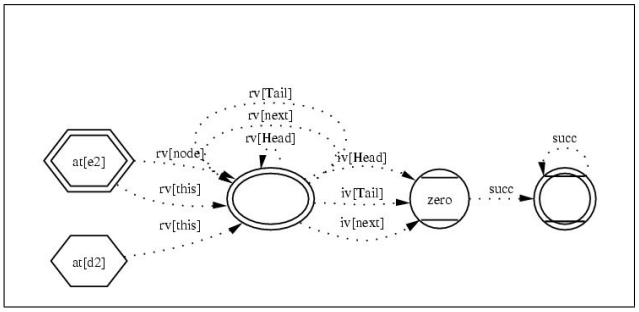

Fig. 3. An abstract configuration C_2 representing the concrete configuration $C_2^\natural$ of Fig. 2.

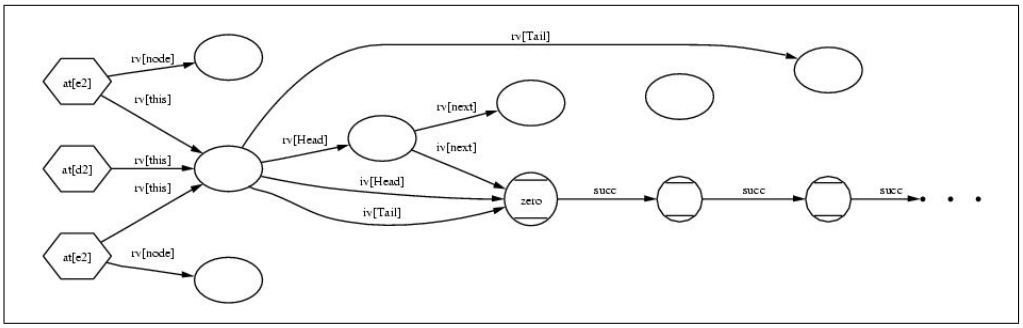

Fig. 4. A concrete configuration $C_{2,1}^\natural$ that is embedded in C_2 and violates queue connectedness (property P1).

bounded size.

We use dashed-edges to draw $1/2$-valued binary predicates, and nodes with double-line boundaries to represent summary nodes.

Example 3.2 The abstract configuration C_2 shown in Fig. 3 is obtained by applying canonic abstraction to the concrete configuration $C_2^\natural$ of Fig. 2.

The summary thread-node represents the two enqueueing threads of the concrete configuration $C_2^\natural$, the summary unsigned-integer node (double-line circle with straight margins) summarizes all unsigned integers but zero, the third summary node summarizes all queue items, and the queue object itself.

Note that this abstract configuration represents an infinite number of configurations. For example, it represents any configuration in which an arbitrary number of enqueuing threads have just allocated new nodes to be enqueued, and are sharing the same queue with an arbitrary number of dequeueing threads that are at their initial labels.

Unfortunately, this abstract configuration also represents the concrete configuration $C_{2,1}^\natural$ which violates the connectedness property (P1), meaning that we fail to verify that P1 holds. Indeed, since each subformulae of P1's body evaluates to $1/2$ over the abstract configuration C_2, using Kleene evaluation of boolean operators yields the value $1/2$ for P1. In the next section, we will see a way to remedy that.

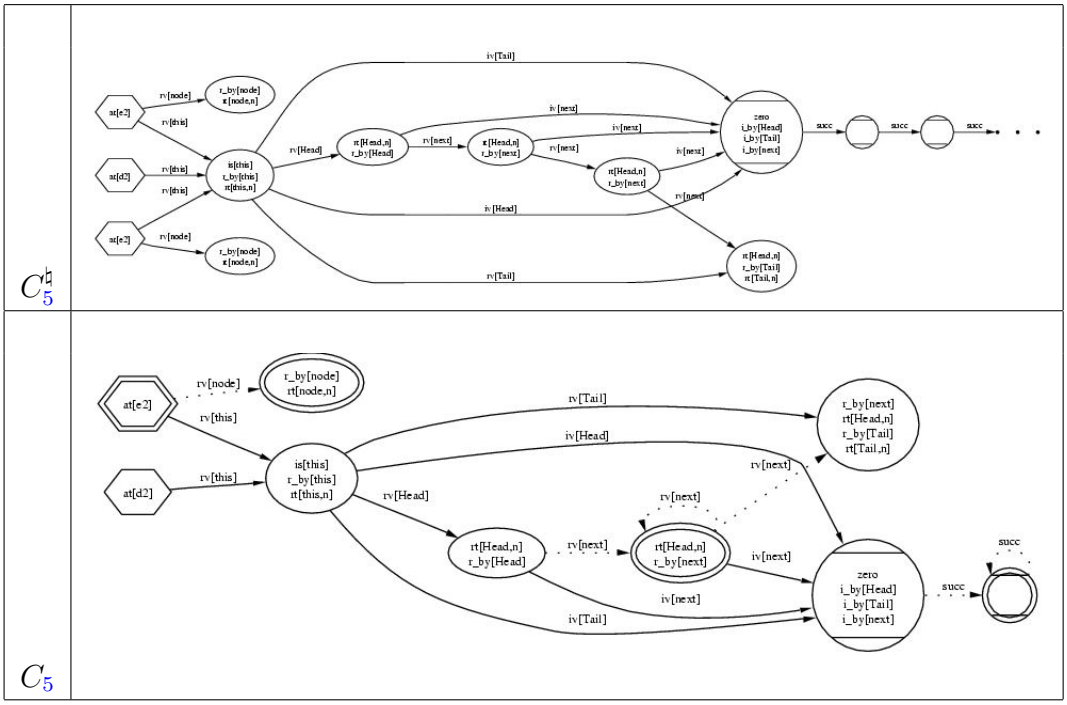

Fig. 5. Concrete configuration $C_5^\natural$ using instrumentation predicates, and its canonic abstraction C_5.

4 Refining the Vanilla Solution

In order to verify the desired properties, in this section we refine the abstraction to record essential information. A natural way to do that would be to record which property-formulae hold using nullary predicates. This is a useful technique, also known as predicate abstraction [7]. TVLA/3VMC allows to also use unary predicates in order to observe whether subformulae hold for a given individual. This allows TVLA/3VMC to provide useful results without changing the set of predicates for each program. We believe that the same distinctions can be used for many programs, furthermore, these distinctions correspond to fundamental properties of data-structures (e.g., sharing, reachability). This paper confirms this by showing that the standard set of distinctions suffices for verifying all the desired properties for the concurrent queue algorithms.

Technically, refining the abstraction is achieved by introducing the unary predicates of Table 3. The additional information recorded refines the abstraction and reduces the set of concrete configurations that are represented by an abstract configuration.

We refer the reader to [12] for a more elaborate discussion of instrumentation predicates.

In principle, some instrumentation predicates could be derived automatically (e.g., [6]), however, for this case study we just use the standard TVLA/3VMC instrumentation predicates.

Predicates $rt[fld, n](t, o)$ (we use n as a shorthand for $next$ in the predicate

Predicate	Intended Meaning	Defining Formula
$r\_by[fld](l)$	l is referenced by the field fld of some object	$\exists o\colon rv[fld](o,l)$
$i\_by[fld](n)$	n is the integer value of fld of some object	$\exists o\colon iv[fld](o,l)$
$is[fld](o)$	o is shared by fld of two different objects	$\exists v_1,v_2.\neg eq(v_1,v_2) \wedge rv[fld](v_1,o)$ $\wedge rv[fld](v_2,o)$
$exists[fld](o)$	there exists an object referenced by fld of o	$\exists v_1.rv[fld](o,v_1)$
$is\_acquired(l)$	l is acquired by some thread	$\exists t\colon heldBy(l,t)$
$rt[fld,n](o)$	o is reachable from object referenced by field fld using path of next fields	$\exists t,o_t\colon rv[fld](t,o_t)$ $\wedge\, rv[next]^*(o_t,o)$

Table 3
Instrumentation predicates used in our example program.

name) allow us to track reachability information of items inside the queue. For example, the instrumentation predicate $rt[Head,n](v)$ may be used to track reachability of items from the head of the queue using a path of *next* references. These predicates are an adaptation for multi-threaded programs of the reachability instrumentation predicates presented in [12]. Similarly, predicates $is[fld](o)$ are an adaptation of sharing predicates of [12]. The predicates $is\_acquired(l)$ and $r\_by[fld](l)$ were originally introduced in [20], and predicates $exists[fld](o)$ used there but not explicitly mentioned in the paper. Since these predicates record widely-used *fundamental properties* of data-structures and thread/lock relationships, they are part of the standard predicates used in TVLA/3VMC.

Subformulae of the safety properties are replaced with the corresponding instrumentation predicate to improve precision.

Example 4.1 Fig. 5 shows the concrete configuration $C_5^\natural$ which is an instrumented version of $C_2^\natural$, and its canonic abstraction C_5. The additional information recorded by the instrumentation predicates $rt[Head,n](v)$ and $rt[Tail,n](v)$ allows us to observe that queue connectedness (property P1) is maintained in the abstract configuration C_5 since P1 evaluates to 1. Moreover, this implies that concrete configuration of the form of $C_{2,1}^\natural$ are no longer represented.

4.1 Abstract Semantics

Implementing an abstract semantics directly manipulating abstract configurations is non-trivial since one has to consider all possible relations on the (possibly infinite) set of represented concrete configurations.

The *best* conservative effect of a program statement [4] is defined by the following 3-stage semantics: (i) a concretization of the abstract configuration is performed, resulting in all possible configurations *represented* by the abstract configuration; (ii) the program statement is applied to each resulting concrete configuration; (iii) abstraction of the resulting configurations is performed, resulting with a

Program	Configs	Space (MB)	Time (sec)	Comments
nbq_enqueue	1833	14.2	727	unbounded number of enqueue-ing threads
nbq_dequeue	1098	5.3	309	unbounded number of dequeue-ing threads
nonblockq_err1	36	0.1	11	err - negated condition at e8
nonblockq_uni	17	0.1	3	err - start with uninitialized queue
tlq_enqueue	982	10	6162	unbounded number of enqueueing thrads
tlq_dequeue	225	4.1	304	unbounded number of dequeuing threads
twolockqn	975	7.5	577	single producer and single consumer
twolockq_err1	24	0.1	30	err - broken producer synchronization

Table 4
Analysis results for variations of the queue algorithms — number of configurations explored, space requirements, and analysis time.

set of abstract configurations *representing* the results of the program statement.

5 Prototype Implementation

Our prototype implementation operates directly on abstract configurations using *abstract transformers*, thereby obtaining actions which are more conservative than the ones obtained by the best transformers. Our experience shows that the abstract transformers used in the implementation are still precise enough to allow verification of our safety properties.

Update formulae for the instrumentation predicates used in this case study were derived automatically using finite differencing [11].

Table 4 presents the analysis results for various variations of the concurrent queue algorithms.

For the non-blocking queue, we have also tested a version in which the conditional in label e_8 is flipped, i.e, it checks for the next field being non-equal to null. As another erroneous version, we have used an uninitialized queue in which no dummy node was present. Both cases reported errors.

For the two-lock queue, we have also tested a version in which no synchronization is imposed on producer threads inserting items into the queue. In this version, we show that it is possible for requirement 1 to be violated, and the underlying linked-list to be broken.

6 Conclusion

We believe the tool is mature enough to be applied to many other challenging examples at the same ease. Our recent experiments with a front-end translating Java program to TVMs are encouraging with this respect.

```
class Producer implements Runnable {
    protected Queue q;
    ...
    public void run() {
         ...
         q.enqueue(val);
    }
}
class Consumer implements Runnable {
    protected Queue q;
    ...
    public void run() {
         ...
         val = q.dequeue();
    }
}
class Main {
    public static void main(String[] args) {
         NonBlockingQueue q = new NonBlockingQueue();
         ...
         new Thread(new Producer(q)).start();
         new Thread(new Consumer(q)).start();
         ...
         }
    }
```

Fig. A.1. simple program using the queue.

A Additional Sources

References

[1] P. A. Abdulla, A. Annichini, S. Bensalem, and A. Bouajjani. Verification of infinite-state systems by combining abstraction and reachability analysis. *Lecture Notes in Computer Science*, 1633, 1999.

[2] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, June 2000. Proceedings available at http://shemesh.larc.nasa.gov/fm/Lfm2000/Proc/.

[3] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. *LNCS*, 1427, 1998.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, New York, NY, 1979. ACM Press.

[5] S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *11th Int. Conf. on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

[6] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. In *Proc. Static Analysis Symposium*, 2003.

[7] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *LNCS*, 1254:72–83, 1997.

[8] N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[9] M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275, New York, USA, May 1996. ACM.

[10] S. Prakash, Y. Lee, and T. Johnson. A non-blocking algorithm for shared queues using Compare-and-Swap. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 68–75, 1991.

[11] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis. In *In Proc. European Symp. on Programming*, 2003.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.

[13] H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Static Analysis Symposium (SAS '00)*, 2000.

[14] J. M. Stone. A simple and correct shared-queue algorithm using Compare-and-Swap. In *Proceedings of Supercomputing '90*, pages 495–504, 1990.

[15] J. M. Stone. A non-blocking Compare-and-Swap algorithm for a shared circular queue. In S. Tzafestas et al., editors, *Parallel and Distributed Computing in Engineering Systems*, pages 147–152. Elsevier Science Publishers, 1992.

[16] R. E. Strom. Mechanisms for compile-time enforcement of security. In *Proc. of the 10th Annual ACM Symposium on Principles of Programming Lanuages*, pages 276–284, Austin, TX, January 1983.

[17] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *tose*, SE-12(1):157–171, January 1986.

[18] J. M. Wing and C. Gong. A library of concurrent objects and their proofs of correctness. Technical Report CMU–CS–90–151, Carnegie-Mellon University, 1990.

[19] E. Yahav. http://www.cs.tau.ac.il/~yahave.

[20] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. of 27th POPL*, pages 27–40, March 2001.