Shape Predicates Allow Unbounded Verification of Linearizability Using Canonical Abstraction

David Friggens[1,2] **Lindsay Groves**[2]

[1] University of Waikato
Email: friggens@waikato.ac.nz

[2] Victoria University of Wellington,
Email: lindsay@ecs.vuw.ac.nz

Abstract

Canonical abstraction is a static analysis technique that represents states as 3-valued logical structures, and is able to construct finite representations of systems with infinite statespaces for verification. The granularity of the abstraction can be altered by the definition of instrumentation predicates, which derive their meaning from other predicates. We introduce shape predicates for preserving certain structures of the state during abstraction. We show that shape predicates allow linearizability to be verified for concurrent data structures using canonical abstraction alone, and use the approach to verify a stack and two queue algorithms. This contrasts with previous efforts to verify linearizability with canonical abstraction, which have had to employ other techniques as well.

Keywords: canonical abstraction, concurrent data structures, linearizability, verification

1 Introduction

Canonical abstraction (Sagiv et al. 2002) is a powerful static analysis technique that can be used to construct bounded finite systems representing unbounded or infinite systems for verification. Key to this is the ability to vary the coarseness of the abstraction by defining so called "instrumentation predicates" that can explicitly preserve specified properties of a state during abstraction. We observe that many of the instrumentation predicates defined previously in the literature record linear relationships between objects in a state, but it may be necessary to record geometric relationships, such as a group of objects forming a triangle shape or a square shape. To demonstrate the effectiveness of these "shape predicates", we consider the problem of verifying that concurrent data structures are linearizable (Herlihy & Wing 1990) with respect to a sequential specification. Preserving the relationship between the implementation and specification data structures is tricky, and without shape predicates other authors (see Section 5) have had to invent other techniques to augment canonical abstraction.

The contributions of this paper are:

- Description of shape predicates, which have not been used in the literature before, to our knowledge.
- Demonstration that linearizability can be verified using canonical abstraction alone.
- (Re-)Verification of a stack and two queue algorithms using canonical abstraction with shape predicates.

The paper is structured as follows. Section 2 gives some background in two parts. In Section 2.1, an overview of canonical abstraction; in Section 2.2, a brief overview of concurrent data structures and the linearizability correctness condition, as well as a stack algorithm to be used as an example. Section 3 defines and explains the canonical abstraction model, including the shape predicates used to verify linearizability of the stack algorithm. Section 4 provides empirical results of verifying linearizability for the stack algorithm and two queue algorithms using canonical abstraction in the TVLA tool. Section 5 discusses related work. Finally, Section 6 concludes and discusses future work.

2 Background

2.1 Canonical Abstraction

Sagiv et al. (2002) represent states as logical structures, where predicates describe relationships between objects. Concrete states are represented using 2-valued structures. Abstract states are represented using 3-valued structures, which allow multiple concrete objects to be represented by a single abstract "summary object". Since a summary object can represent two or more concrete objects, an abstract state with summary objects can represent an infinite number of concrete states.

First, a finite set of predicates $\mathcal{P} = \{\mathsf{eq}, \mathsf{p}_1, \ldots, \mathsf{p}_n\}$ is fixed for the analysis, and we define $\mathcal{P}_k$ to be the set of k-ary predicates in $\mathcal{P}$ (the equality predicate eq has arity 2). Then, a *concrete configuration* $S^\natural = \langle U^\natural, \iota^\natural \rangle$ has a *universe* $U^\natural$ that is a (finite or infinite) set of objects and an *interpretation* $\iota^\natural$ over the logical values true (1) and false (0). For each k-ary predicate p,

$$\iota^\natural(\mathsf{p}) : (U^\natural)^k \to \{0, 1\}$$

Additionally, for each $u_1, u_2 \in U^\natural$ where $u_1 \neq u_2$, $\iota^\natural(\mathsf{eq})(u_1, u_1) = 1$ and $\iota^\natural(\mathsf{eq})(u_1, u_2) = 0$.

The definition of an *abstract configuration* $S = \langle U, \iota \rangle$ is similar to that of a concrete configuration, but the interpretation is over the truth values true (1),

false (0) and unknown ($\frac{1}{2}$). For each k-ary predicate p,

$$\iota(\mathsf{p}) : U^k \to \{1, 0, \tfrac{1}{2}\}$$

Note that a concrete configuration is also trivially an abstract configuration. An object u, for which $\iota(\mathsf{eq})(u, u)$ is unknown, is called a *summary object*.

Intuitively, an abstract configuration represents a concrete one if it contains the same information, except for some conservative information loss. In other words, it has the same universe of objects, though some may have been merged together into summary objects, and it has the same predicate interpretations, though some may have become unknown. This is formalised by the notion of embedding, which relates configurations (concrete or abstract[1]) that are related by conservative information loss.

We say that a configuration $S_1 = \langle U_1, \iota_1 \rangle$ *embeds* into an abstract configuration $S_2 = \langle U_2, \iota_2 \rangle$ if there exists a surjective function $f : U_1 \to U_2$ such that for every k-ary predicate p, and $u_1, \ldots, u_k \in U_1$,

$$\iota_1(\mathsf{p})(u_1, \ldots, u_k) \sqsubseteq \iota_2(\mathsf{p})(f(u_1), \ldots, f(u_k))$$

where, for $l_1, l_2 \in \{1, 0, \frac{1}{2}\}$, $l_1 \sqsubseteq l_2$ iff $l_1 = l_2$ or $l_2 = \frac{1}{2}$.

We further define a *tight embedding* to be one that minimises information loss, i.e. a predicate interpretation only becomes unknown if two objects are being merged together, one which has a true interpretation and the other a false interpretation. Formally, there exists a surjective function $f : U_1 \to U_2$ such that for every k-ary predicate p, and $u_1, \ldots, u_k \in U_2$,

$$\iota_2(\mathsf{p})(u_1, \ldots, u_k) = \begin{cases} 1 & \text{if } \forall\, u'_1 \in f^{-1}(u_1), \ldots, u'_k \in f^{-1}(u_k) \bullet \\ & \quad \iota_1(\mathsf{p})(u'_1, \ldots, u'_k) = 1 \\ 0 & \text{if } \forall\, u'_1 \in f^{-1}(u_1), \ldots, u'_k \in f^{-1}(u_k) \bullet \\ & \quad \iota_1(\mathsf{p})(u'_1, \ldots, u'_k) = 0 \\ \frac{1}{2} & \text{otherwise} \end{cases}$$

Canonical abstraction is a method for constructing tight embeddings. Given a subset of the unary predicates $\mathcal{A} \subseteq \mathcal{P}_1$, called the *abstraction predicates*, we map objects in the original configuration to the same abstract object if they have the same interpretations over the abstraction predicates. The interpretation in the abstract configuration is constructed as per the definition of tight embeddings above. We say that a configuration is canonically abstract, with respect to $\mathcal{A}$, if it is the canonical abstraction of itself.

Canonical abstraction has a number of important properties:

- Every configuration has a single canonical abstraction, as each object has a single canonical mapping in the embedding function.
- Since there are a finite number of abstraction predicates, it follows that there is a finite bound on the number of objects in the universe of a canonically abstract configuration, and thus a finite bound on the number of potential states in an abstract system.

The soundness of the canonical abstraction approach rests upon the Embedding Theorem of Sagiv et al. (2002, Theorem 4.9). Informally, this says that if a structure S embeds into a structure S', then any information extracted from S' via a formula φ is a conservative approximation of the information extracted from S via φ. Alternatively, if we prove a property φ true or false in S', then we know it has the same value in S.

The initial work of Sagiv et al. (2002) focused on sequential heap-manipulating programs, with a configuration universe representing the objects of the heap. This can be extended to represent concurrent programs, by including an object in the universe for each thread, and defining predicates to represent the threads' locations and fields (Yahav & Sagiv 2010).

2.1.1 Refining abstractions

Canonical abstraction using the fixed predicates $\mathcal{P}$ is often too coarse, resulting in too much information being lost (i.e. evaluating to unknown) for a property to be verified. A key method for refining abstractions is to introduce additional predicates that record properties derived from the other predicates. These *instrumentation predicates* add no new information to a concrete state, since they evaluate to the same truth values as their defining formulas. However, in an abstract state they may add information: an instrumentation predicate may evaluate to a definite value (true or false) whilst its defining formula may evaluate to unknown. Additionally, unary instrumentation predicates may be added to the set of abstraction predicates, which can prevent some objects from being merged together into summary objects.

Defining instrumentation predicates to sufficiently refine the abstraction is the principal focus of Section 3.

2.2 Concurrent Data Structures

In this paper, we consider concurrent data structure algorithms (see e.g. Moir & Shavit 2004), which have multiple threads interacting with shared data, and synchronising access using locks or atomic primitives such as compare-and-swap (CAS).

A common correctness condition is linearizability (Herlihy & Wing 1990), which informally requires each operation to appear to take effect atomically at some point between its invocation and response. A system is linearizable, with respect to a given sequential specification, if the operations in any execution can be rearranged — respecting the ordering of non-concurrent operations — into an execution of the specification. One way of showing this is by determining "linearization points" for each operation, where the operation can be seen to take effect. If the specification is composed with the implementation and can perform a matching operation atomically at each linearization point then the implementation is linearizable.[2]

2.2.1 Example: Stack

Figure 1 gives the pseudocode for a linked list based stack algorithm. Each node of the list contains a value in the *val* field and a *next* field pointing to another node (or is *null*). A shared *Head* variable points to the first element when the stack is non-empty, and is *null* when the stack is empty. The algorithm assumes a garbage collector is present — popped nodes are not explicitly freed.

[1] Since 2-valued configurations are trivially 3-valued configurations also, we will assume that configurations are 3-valued unless otherwise noted.

[2] In general this is more complicated, as an operation's linearization point may be a step of another operation, one step may be the linearization point for several operations, or a step may or may not be a linearization point depending on the future behaviour of other threads.

Type: Node = {*val* : T; *next* : Node}
Shared: *Head* : Node := null

```
1:  operation PUSH(lv:T)
2:      n := new(Node)
3:      n.val := lv
4:      repeat
5:          ss := Head
6:          n.next := ss
7:      until CAS(Head, ss, n)
8:  end operation
9:  operation POP( )
10:     repeat
11:         ss := Head
12:         if ss = null then
13:             return empty
14:         end if
15:         ssnext := ss.next
16:         lv := ss.val
17:     until CAS(Head, ss, ssnext)
18:     return lv
19: end operation
```

Figure 1: A lock-free stack algorithm

A push operation obtains a new node n and sets its value. It then takes a "snapshot" of *Head* and points n's *next* field at the snapshot. A CAS operation is used to ensure that *Head* is updated to point to n only if it has not been modified. If *Head* has been modified then there has been a conflict with another (successful) operation so the loop is restarted.

A pop operation first takes a snapshot of *Head* and tests to see if the snapshot is *null*; if so it returns "empty". Otherwise it takes a snapshot of this node's *next* field and records the value in the *val* field. As for push, a CAS is used to detect a conflict with another successful operation — if *Head* has been modified it retries, otherwise it uses the snapshots to advance *Head* along the list.

This algorithm was first introduced by Treiber (1986) in IBM System/370 assembler. The version here assuming garbage collection follows that given by Colvin et al. (2005). Versions of the algorithm have been formally verified by several authors (including Colvin et al. 2005).

We can see that the algorithm is linearizable by determining the linearisation points of the operations:

- A push operation takes effect at line 7, when the CAS is successful.
- A non-empty pop operation takes effect at line 17, when the CAS is successful.
- An empty pop operation "takes effect" at line 11 when it reads a null *Head* value. The linearisation point is not at line 12 when the snapshot is tested, because *Head* may have been changed by other threads, so the stack cannot be guaranteed to be empty at that point in time.

For the first two, the successful CAS step is where the change of an added or removed node becomes observable to the other threads, and it is the trigger for leaving the loop, so cannot repeat. For the third, a *null* snapshot causes the thread to execute lines 12–13 and exit the loop (and operation), so the linearisation point cannot be repeated.

3 Verification of Stack

In order to attempt to verify linearizability for the concurrent stack algorithm in Section 2.2.1, we include an additional linked-list stack, which performs a Push or Pop operation atomically at the linearisation points of the implementation operation. If the implementation and specification operations always match, i.e. they always push and pop the same values, then the concurrent stack is linearizable. If they do not match, e.g. the specification Pop returns empty

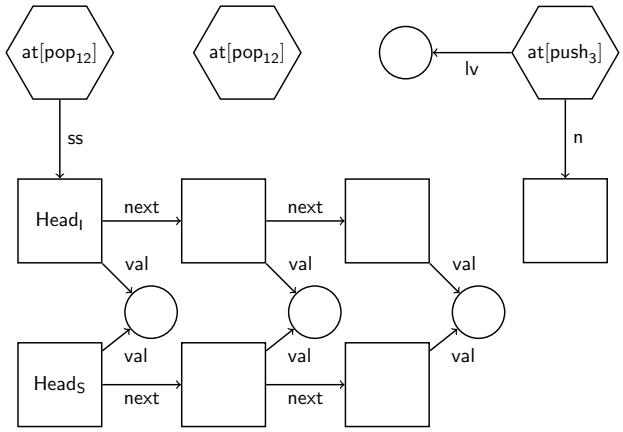

Figure 2: A potential concrete configuration

but the implementation Pop returns a value, then linearizability has not been shown.[3]

To represent this system for canonical abstraction, we define the set of predicates initially to contain unary predicates representing object types (is_thread, is_node, is_data), shared variables for the two stack lists ($\mathsf{Head_l}$, $\mathsf{Head_S}$) and thread locations (at[loc], for $\mathsf{loc} \in \{\mathsf{idle}, \mathsf{push_2}, \mathsf{push_3}, \ldots, \mathsf{pop_{11}}, \mathsf{pop_{12}}, \ldots\}$). Additionally, we have binary predicates representing the fields of the nodes (next, val) and the threads (n, lv, ss, ssnext).

For clearer explanations, we will describes states diagrammatically, rather than logically. We use different object shapes to represent the type predicates — hexagons for threads, squares for nodes, and circles for data values. Unary predicates are shown as labels on objects when true, binary predicates are shown as arrows (solid for true, dotted for unknown, not shown for false), and summary objects have a double line. Figure 2 shows a potential concrete configuration, and Figure 3 its canonical abstraction.

The two stack lists have length three, with three distinct data values. One of the three threads has just begun a push operation; the other two have just begun pop operations, though one has a current snapshot of the $\mathsf{Head_l}$ and the other has a stale null snapshot taken when the stack was empty.

As is common, the canonical abstraction on core predicates alone is too coarse. For example, we cannot distinguish between the nodes of the two different lists, nor those from the nodes not yet pushed, and cannot tell whether a thread has a null or non-null field. This means that abstraction of some reachable

[3] The algorithm may not be linearizable, or it may be linearizable but we have chosen incorrect linearization points. Determining which is the case is outside the scope of this paper.

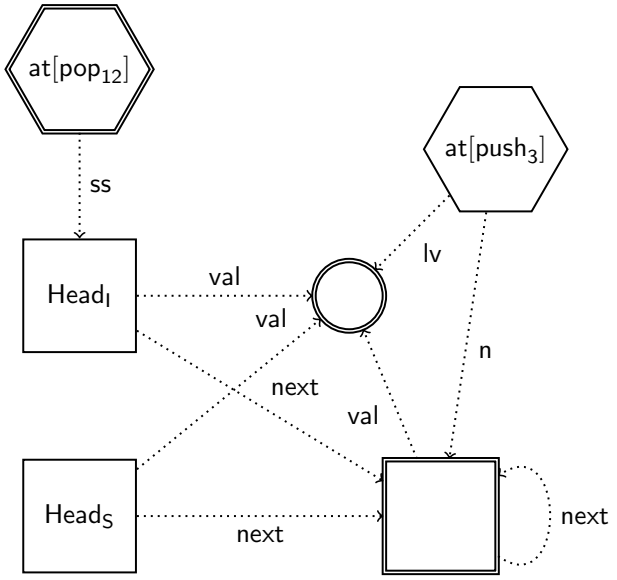

Figure 3: Canonical abstraction of potential configuration

states will also represent some nonreachable states, which will lead to executions with spurious errors detected. We can refine the abstraction by defining instrumentation predicates, of forms previously used by other authors (e.g. Sagiv et al. 2002, Yahav & Sagiv 2010):[4]

$$\begin{aligned}
\mathsf{has[field]}(v) &\Leftrightarrow \exists\, u \bullet \mathsf{field}(v,u)\\
\mathsf{r_by[n]}(v) &\Leftrightarrow \exists\, u \bullet \mathsf{n}(u,v)\\
\mathsf{shared[n]}(v) &\Leftrightarrow \exists\, u_1,u_2 \bullet \mathsf{n}(u_1,v) \wedge \mathsf{n}(u_2,v)\\
&\qquad \wedge \neg\, \mathsf{eq}(u_1,u_2)\\
\mathsf{circ}(v) &\Leftrightarrow \mathsf{next}^+(v,v)\\
\mathsf{reach_l}(v) &\Leftrightarrow \exists\, u \bullet \mathsf{Head_l}(u) \wedge \mathsf{next}^*(u,v)\\
\mathsf{reach_S}(v) &\Leftrightarrow \exists\, u \bullet \mathsf{Head_S}(u) \wedge \mathsf{next}^*(u,v)
\end{aligned}$$

where p^+ is the transitive closure of p, and p^* is the reflexive transitive closure. These instrumentation predicates allow more information to be preserved in the abstract states by, e.g. preventing the two list bodies from being merged together, and recording that each is connected and acyclic.

The two lists should have the same data values, in the same order. However, when the tails of the lists are abstracted to summary objects, this property is lost. In order to specify properties of the pair of ith nodes in the two lists, we introduce an auxiliary core binary predicate R to relate them. R is set between the head nodes of the lists by the specification Push operation, and is unset for the head nodes by the specification Pop operation.

Even with the auxiliary predicate, the instrumentation predicates defined above are not sufficient to preserve all the properties we need about the lists and the threads' fields. We observe that these predicates all define linear properties — has[field] and r_by[n] describe two objects related by one predicate; shared[n] describes three objects related by two predicates; $\mathsf{reach_l}$ describes an arbitrary number of objects related by a chain of predicates; circ describes the same, but the chain begins and ends with the same object. We defined three additional instrumentation predicates that describe geometric shapes relating three or four objects.

[4]The square brackets have no meaning other than being a visual indicator of which core predicates are used in the definition. (TVLA allows parametrised definitions of sets of predicates in this way — e.g. to define reach[y, next] and reach[z, next] at the same time.)

3.1 Matching triangle predicate

Consider Figure 5, which shows the abstraction of (the lists of) two states where the implementation and specification stacks have length 3 — in $S_1^\natural$ the lists have the same values in the same order, and in $S_2^\natural$ the lists' head values differ. Both states have the same canonical abstraction and the information about the values is lost.

In order to preserve the property that each corresponding pair of nodes in the lists have the same data value, we define an instrumentation predicate, called matching:

$$\begin{aligned}
&\mathsf{matching}(n_1) \Leftrightarrow \exists\, n_2, d_1 \bullet\\
&\quad \mathsf{R}(n_1,n_2) \wedge \mathsf{val}(n_1,d_1) \wedge \mathsf{val}(n_2,d_1)
\end{aligned}$$

The predicate records a "triangular" relationship between nodes and data values, as shown in the first diagram in Figure 4.

Adding this instrumentation predicate to the concrete states in Figure 5 results in different canonically abstract states. Both would differ from S_1 as the implementation list summary node would be labelled with matching; and both would differ from each other as one would have the head implementation node labelled with matching and the other would not.

3.2 Commutes square predicate

Consider the two concrete states in Figure 6 — they both have three elements, and matching is true for all the implementation list nodes. In $S_4^\natural$, the R relations have "crossed", so after a Pop operation the head nodes will have different values — another Pop from both lists will trigger a linearizability error. We see that these two states have the same canonical abstraction, so analysis of a linearizable stack can still provide spurious errors.

In order to preserve the property that related pairs of nodes have the same order in both lists, we define an instrumentation predicate that records whether the next and R predicates "commute":

$$\begin{aligned}
&\mathsf{commutes}(n_1) \Leftrightarrow \exists\, n_2, n_3, n_4 \bullet\\
&\quad \mathsf{next}(n_1,n_2) \wedge \mathsf{R}(n_1,n_3) \wedge\\
&\quad \mathsf{next}(n_3,n_4) \wedge \mathsf{R}(n_2,n_4)
\end{aligned}$$

The predicate records a "square" relationship between nodes, as shown in the second diagram in Figure 4.

Adding this instrumentation predicate to the concrete states in Figure 6 results in different canonically abstract states. The first two implementation nodes in $S_3^\natural$ are labelled with commutes; then since all three implementation nodes have different abstraction predicate labels none of them will be merged in to summary objects in the canonical abstraction. The diagrams for the other concrete state and its canonical abstraction are identical to $S_4^\natural$ and S_3, as commutes is false for all of the implementation nodes.

Together, matching and commutes preserve sufficient information about the two lists to allow linearizability to be verified.

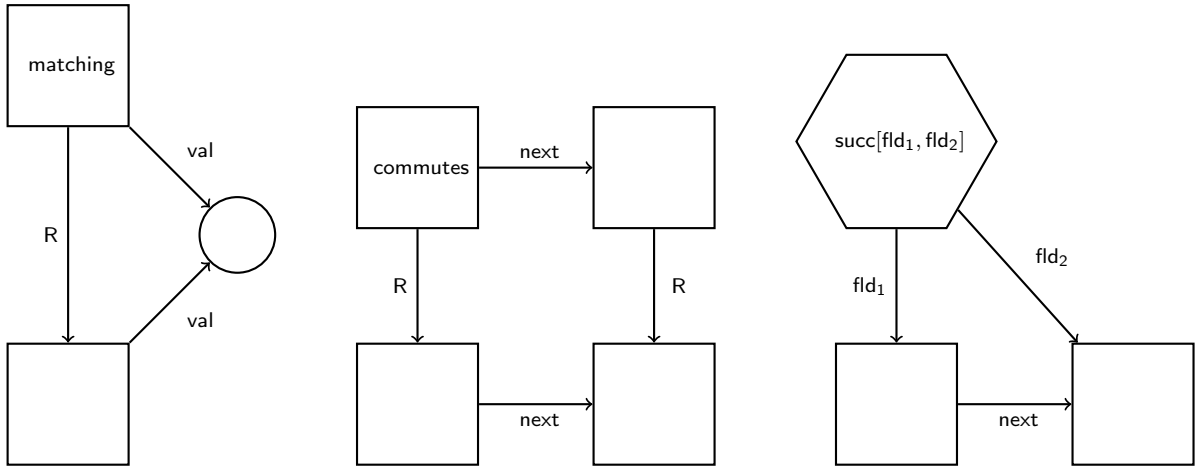

Figure 4: Three shape predicate diagrams

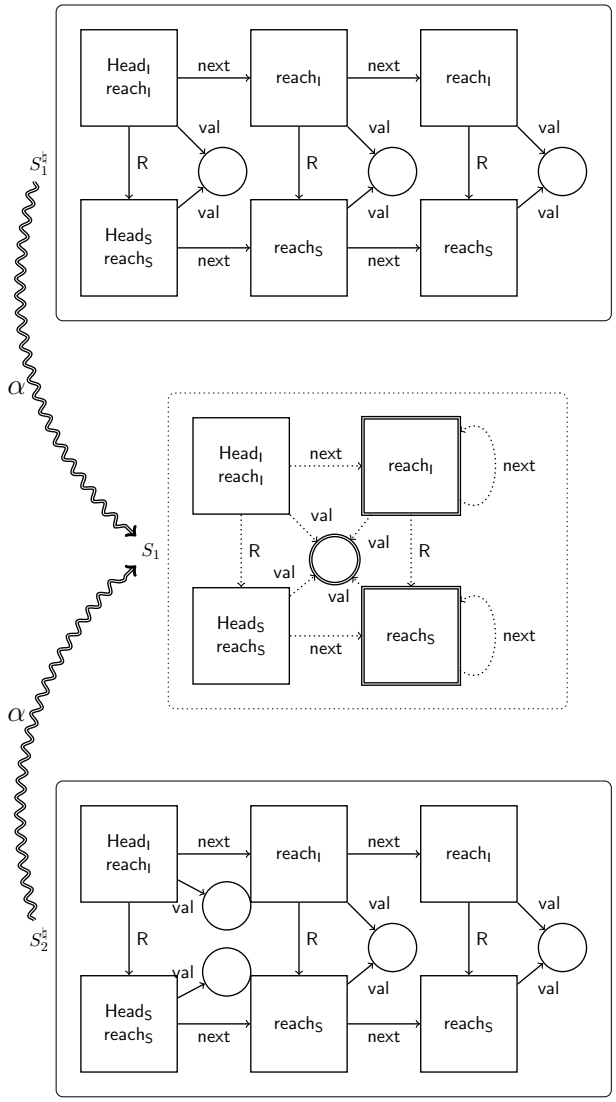

Figure 5: Canonical abstraction of two lists: the property of matching values is lost

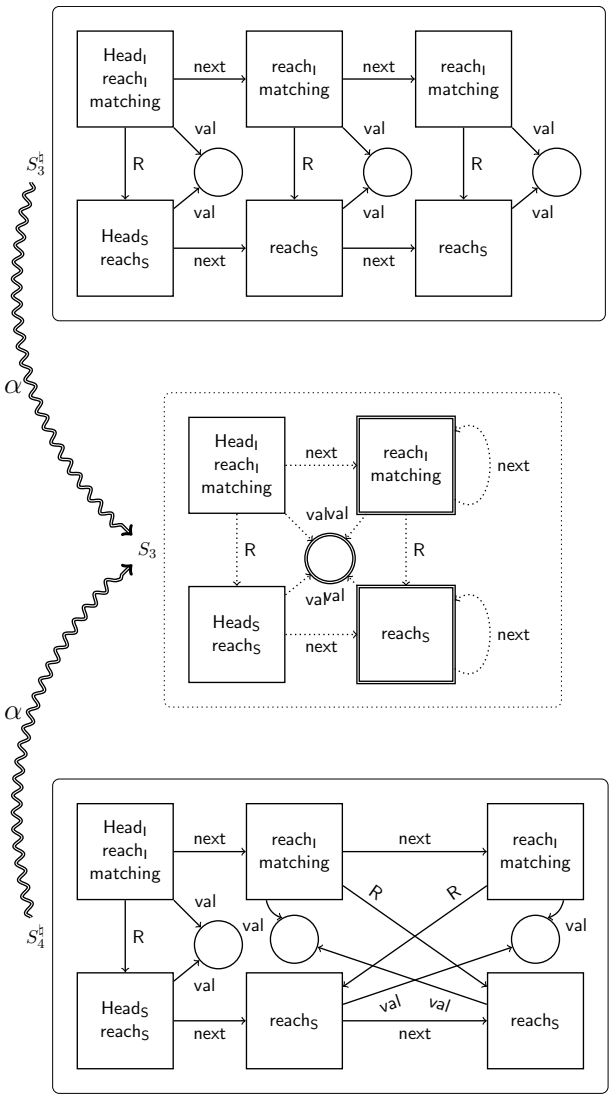

Figure 6: Canonical abstraction with "crossed" R predicates: the property of ordered values is lost

3.3 Successor triangle predicate

In the stack's Pop operation, the ssnext field is the next-successor of the ss field when it is read at line 15. This property is assumed to persist, so it is not checked before the CAS step at line 17 that attempts to set Head_l to ssnext.

Figure 7 shows that this property is not retained in canonical abstraction using the predicates defined so far. Both states (shown without the data values and specification lists) have two threads performing a Pop operation — in $S_5^\natural$, both ssnext predicates are the next-successors of the respective ss predicates, but this is not the case in $S_6^\natural$; nevertheless, both states have the same canonical abstraction (S_5). As a consequence, the CAS transition can remove an arbitrary prefix of the list because ssnext can be concretised at any point.

In order to preserve the relationship between the thread fields, we define an instrumentation predicate that records whether they are next-successors:

$$\mathsf{succ}[\mathsf{ss},\mathsf{ssnext}](t_1) \Leftrightarrow \exists\, n_1, n_2 \bullet \mathsf{ss}(t_1, n_1) \wedge \mathsf{ssnext}(t_1, n_2) \wedge \mathsf{next}(n_1, n_2)$$

This predicate records a "triangular" relationship between threads and nodes, as shown in the third diagram in Figure 4.

Adding this instrumentation predicate to the concrete states in Figure 7 results in different canonically abstract states. The diagrams are almost the same, but are distinguished by whether the thread summary object has is labelled with $\mathsf{succ}[\mathsf{ss},\mathsf{ssnext}]$ or not.

A similar situation arises in the Push operation. The ss predicate is set to be the next-successor of the n predicate at line 6, which is assumed to be unchanged at the CAS step that sets Head_l to n at line 7. Thus we similarly define the instrumentation predicate $\mathsf{succ}[\mathsf{n},\mathsf{ss}]$.

For space restrictions, we omit further discussion on the basic model constructs, notably details about transitions. A complete presentation can be found in Friggens (2013, Chapter 7).

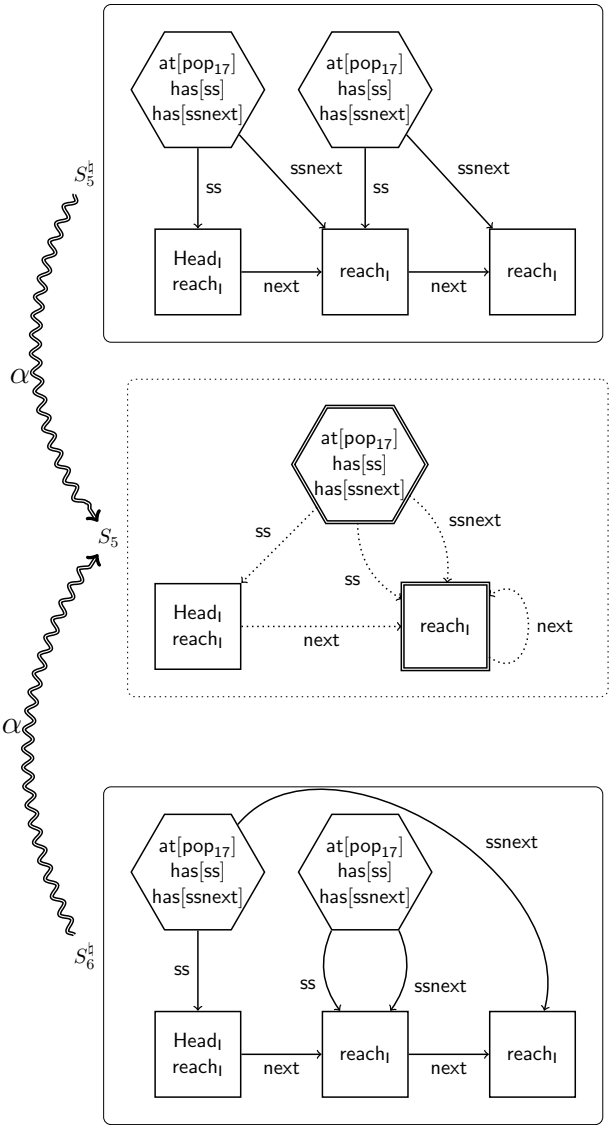

Figure 7: Canonical abstraction of threads: the relationships between fields' values are lost

4 Empirical Results

To perform analyses and gather empirical results, we used TVLA[5] (Three Valued Logic Analyzer) (Lev-Ami & Sagiv 2000, Bogudlov et al. 2007), a prototype static analysis tool developed at Tel Aviv University that implements canonical abstraction.

4.1 Stack

We analysed thread-bounded and unbounded models of the stack algorithm using TVLA 3.0α on a machine with an Intel Core 2 3.0 GHz processor and 4 GB of RAM, running Java 1.6.0 on a 32-bit GNU/Linux operating system. By default, TVLA can construct models with one thread or an unbounded number of threads, depending on whether the initial configuration has a summary or non-summary idle thread object. To obtain models with some other bounded number of threads, we defined compatibility constraints (Sagiv et al. 2002, Section 6.4.2) that discard any configuration that satisfy a formula identifying $n+1$ or more distinct thread objects. For example, bounding to two threads:

$$\exists\, t_1, t_2, t_3 \bullet \mathsf{is\_thread}(t_1) \wedge \mathsf{is\_thread}(t_2) \wedge \mathsf{is\_thread}(t_3) \wedge \neg\, \mathsf{eq}(t_1, t_2) \wedge \neg\, \mathsf{eq}(t_2, t_3) \wedge \neg\, \mathsf{eq}(t_1, t_3)$$

Table 1 contains results of verifying linearizability with unbounded lists and data values, and up to three threads. Time, memory and statespace figures are as reported by TVLA. With four threads, the model was too large and TVLA ran out of memory.

Since TVLA does not implement techniques such as partial order reduction (Peled 1998) to reduce unnecessary interleavings of threads, we manually modified the models to restrict interleaving of transitions that only read or write to local variables. Table 2 contains results of these models, which appear to have an exponential reduction in statespace. The principal result is that linearizability of the stack algorithm is able to be verified for unbounded numbers of threads and data values, and for lists of unbounded length, using only canonical abstraction.

We note that the time for the analyses of bounded models increases exponentially - the general approach taken by TVLA of evaluating our bounding formulas becomes increasingly impractical as the number of

[5] http://www.cs.tau.ac.il/~tvla/

Th.	Heap Limit (MB)	Time (s)	Ave RAM (MB)	Max RAM (MB)	Stored States
1	800	2	16	38	148
2	800	88	178	335	7,731
3	2,048	2,931	1,089	1,946	148,191
4	2,048	—	—	—	>282,441

Table 1: Stack verification results with full interleaving

Th.	Heap Limit (MB)	Time (s)	Ave RAM (MB)	Max RAM (MB)	Stored States
1	800	1	4	6	86
2	800	18	94	252	1,312
3	800	102	173	336	6,493
4	800	535	255	479	18,564
5	2,048	6,409	502	1,184	36,749
6	2,048	143,302	480	1,052	55,069
7	4,096	2,625,113	1,182	2,633	67,334
∞	1,024	6,524	647	1,057	74,056
∞	2,048	1,934	849	1,603	74,056

Table 2: Stack verification results with restricted interleaving

thread objects being identified, and thus the length of the formula, increases. It may well be possible to implement a more direct and efficient way in TVLA for limiting numbers of specific objects; if so, it would make verifying models with bounds of greater than six threads practical.

4.2 Queues

We additionally analyzed linearizability for two nonblocking queue data structures, the original due to Michael & Scott (1998). Doherty et al. (2004) give a variation with a simplified dequeue operation; they also provide a formal verification using a theorem prover.

The canonical abstraction models are constructed similarly to the stack models, with similar shape predicates — full details are available in Friggens (2013, Section 7.9).

Table 3 contains results, using the same software and hardware as for the stack. To reduce the statespace we again added manual restrictions to interleaving for steps that only read and write to local variables. For both algorithms we verify linearizability for one or two threads, unbounded numbers of data values and lists of unbounded length. For the models with three threads, the model is too large and TVLA runs out of memory.

Deq	Th.	Heap Limit (MB)	Time (s)	Ave RAM (MB)	Max RAM (MB)	Stored States
MS	1	800	1	13	30	115
MS	2	800	393	260	476	24,271
MS	3	2,048	—	—	—	>235k
Doh	1	800	1	14	33	117
Doh	2	800	83	189	354	10,746
Doh	3	2,048	—	—	—	>230k

Table 3: Queue verification results

5 Related Work

The closest work to ours is by Amit et al. (2007), who analysed the same nonblocking data structures (plus two lock-based data structures). They also restricted interleaving of threads manually, and were able to verify linearizability for the stack algorithm with three threads and the queue algorithms with two threads (limiting to 1.5 GB of RAM). They combine canonical abstraction with an additional approach called "delta heap abstraction": the relationship between each pair of implementation and specification nodes and their identical value is represented in the state graph by a single object. Delta heap abstraction requires each push/enqueue etc. to be for a unique value, whereas our approach can represent data values being entered into the list multiple times. Their analyses use unique predicates to distinguish each thread and its field values; this is exponentially more expensive than using the shape predicates we have defined, and does not allow unbounded numbers of threads to be considered.

This approach is made more efficient by Manevich et al. (2008), who combine canonical abstraction with heap decomposition. Heap decomposition splits the state into (overlapping) subgraphs and only stores one copy of a subgraph no matter how many states it appears in. They were able to verify linearizability for the stack algorithm with 20 threads (limiting to 2 GB of RAM), and for the second queue algorithm with 15 threads (limiting to 16 GB of RAM).

Berdine et al. (2008) combine the above approaches with an additional approach called "quantified canonical abstraction" to verify linearizability for unbounded threads. Like heap decomposition, the approach splits the state into (overlapping) subgraphs, each containing the data structure and one non-summary thread. Unlike heap decomposition, each subgraph can represent an unbounded number of identical subgraphs, thus the bounded number of subgraphs together can represent states with unbounded numbers of threads. Extending the models of Amit et al. (2007), and limiting to 2 GB of RAM, Berdine et al. (2008) were able to verify linearizability for the stack algorithm, but ran out of memory for the queue. Extending the models of Manevich et al. (2008), using heap decomposition to create smaller subgraphs, they were able to verify linearizability for both the stack algorithm (with an 80% reduction in statespace) and queue algorithm. This is the first published work to verify linearizability for unbounded threads using canonical abstraction, though it uses two additional approaches to do so.

6 Conclusions and Further Work

In this paper we have introduced shape predicates, a type of instrumentation predicate for refining canonical abstractions. Though defining triangles and squares may seem obvious in hindsight, these predicates have not been used before in the canonical abstraction literature and can prove to be powerful in constructing an appropriate abstraction. They will almost certainly be of use in a wide range of canonical abstraction applications.

We have demonstrated the utility of shape predicates by verifying linearizability for three concurrent data structure algorithms. In doing so we have demonstrated the interesting theoretical result that verification of linearizability is possible with canonical abstraction alone, and does not require delta heap abstraction or thread quantification.

The abstract models that are constructed for the

stack and two queue algorithms are finite, but still very large. Restricting the interleaving of local steps, we were able to completely verify the stack algorithm. However, for the un-restricted stack and for the restricted queues, the analyses ran out of memory for models with four or more threads. The principal problem is the exponential permutations of thread objects and list configurations. One approach to improving the performance would be to employ heap decomposition (Manevich et al. 2008) or thread quantification (Berdine et al. 2008), with which (an extension to) TVLA decomposes each state into list and thread components, storing each only once, no matter how many states the component appears in. An alternative approach would be to collapse all of the thread objects in a state into a single summary object, defining "soft invariant" instrumentation predicates (Friggens & Groves 2013) to preserve properties of the threads that would be lost otherwise.

Finally, we would like to extend the verifications of the stack and queue algorithms to other concurrent data structures. Some data structures, such as deques, have a similar property of having a close correspondence between the implementation and specification data structures, so a similar approach would be reasonable to expect. Other data structures, such as elimination stacks and sets have more difference between the implementation and specification data structures, so more ingenuity in the model construction may be required.

References

Amit, D., Rinetzky, N., Reps, T., Sagiv, M. & Yahav, E. (2007), Comparison under abstraction for verifying linearizability, *in* W. Damm & H. Hermanns, eds, 'Proceedings of the 19th International Conference on Computer Aided Verification (CAV)', Vol. 4590 of *Lecture Notes in Computer Science*, Springer, pp. 477–490.

Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G. & Sagiv, M. (2008), Thread quantification for concurrent shape analysis, *in* A. Gupta & S. Malik, eds, 'Proceedings of the 20th International Conference on Computer Aided Verification (CAV)', Vol. 5123 of *Lecture Notes in Computer Science*, Springer, pp. 399–413.

Bogudlov, I., Lev-Ami, T., Reps, T. & Sagiv, M. (2007), Revamping TVLA: Making parametric shape analysis competitive, *in* W. Damm & H. Hermanns, eds, 'Proceedings of the 19th International Conference on Computer Aided Verification (CAV)', Vol. 4590 of *Lecture Notes in Computer Science*, Springer, pp. 221–225.

Colvin, R., Doherty, S. & Groves, L. (2005), Verifying concurrent data structures by simulation, *in* J. Derrick & E. A. Boiten, eds, 'Proceedings of the Refinement Workshop', Vol. 137.2 of *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 93–110.

Doherty, S., Groves, L., Luchangco, V. & Moir, M. (2004), Formal verification of a practical lock-free queue algorithm, *in* D. de Frutos-Escrig & M. Núñez, eds, 'Proceedings of the 24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)', Vol. 3235 of *Lecture Notes in Computer Science*, Springer, pp. 97–114.

Friggens, D. (2013), On the Use of Model Checking for the Bounded and Unbounded Verification of Non-blocking Concurrent Data Structures, Ph.D. thesis, Victoria University of Wellington.

Friggens, D. & Groves, L. (2013), 'Collapsing threads safely using soft invariants', In preparation.

Herlihy, M. P. & Wing, J. M. (1990), 'Linearizability: A correctness condition for concurrent objects', *ACM Transactions on Programming Languages and Systems* **12**(3), 463–492.

Lev-Ami, T. & Sagiv, M. (2000), TVLA: A system for implementing static analyses, *in* J. Palsberg, ed., 'Proceedings of the 7th International Symposium on Static Analysis (SAS)', Vol. 1824 of *Lecture Notes in Computer Science*, Springer, pp. 280–301.

Manevich, R., Lev-Ami, T., Sagiv, M., Ramalingam, G. & Berdine, J. (2008), Heap decomposition for concurrent shape analysis, *in* M. Alpuente & G. Vidal, eds, 'Proceedings of the 15th International Symposium on Static Analysis (SAS)', Vol. 5079 of *Lecture Notes in Computer Science*, Springer, pp. 363–377.

Michael, M. M. & Scott, M. L. (1998), 'Non-blocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors', *Journal of Parallel and Distributed Computing* **51**(1), 1–26.

Moir, M. & Shavit, N. N. (2004), Concurrent data structures, *in* D. P. Mehta & S. Sahni, eds, 'Handbook of Data Structures and Applications', Chapman and Hall/CRC, chapter 47.

Peled, D. A. (1998), Ten years of partial order reduction, *in* A. J. Hu & M. Y. Vardi, eds, 'Proceedings of the 10th International Conference on Computer Aided Verification (CAV)', Vol. 1427 of *Lecture Notes in Computer Science*, Springer, pp. 17–28.

Sagiv, M., Reps, T. & Wilhelm, R. (2002), 'Parametric shape analysis via 3-valued logic', *ACM Transactions on Programming Languages and Systems* **24**(3), 217–298.

Treiber, R. K. (1986), Systems programming: Coping with parallelism, Technical Report RJ 5118, IBM Almaden Research Centre.

Yahav, E. & Sagiv, M. (2010), 'Verifying safety properties of concurrent heap-manipulating programs', *ACM Transactions on Programming Languages and Systems* **32**(5). Article 18.