Thread Quantification for Concurrent Shape Analysis

J. Berdine[1], T. Lev-Ami[2], R. Manevich[2], G. Ramalingam[3], and M. Sagiv[2]

[1] Microsoft Research Cambridge, jjb@microsoft.com
[2] Tel Aviv University, {tla,rumster,msagiv}@post.tau.ac.il
[3] Microsoft Research India, grama@microsoft.com

Abstract. We present new algorithms for automatically verifying properties of programs with *an unbounded number of threads*. Our algorithms are based on a new abstract domain whose elements represent *thread-quantified* invariants: i.e., invariants satified by all threads. We exploit existing abstractions to represent the invariants. Thus, our technique lifts existing abstractions by wrapping universal quantification around elements of the base abstract domain.
Such abstractions are effective because they are thread-modular: e.g., they can capture correlations between the local variables of the same thread as well as correlations between the local variables of a thread and global variables, but forget correlations between the states of distinct threads. (The exact nature of the abstraction, of course, depends on the base abstraction lifted in this style.)
We present techniques for computing sound transformers for the new abstraction by using transformers of the base abstract domain. We illustrate our technique in this paper by instantiating it to the Boolean Heap abstraction, producing a Quantified Boolean Heap abstraction. We have implemented an instantiation of our technique with Canonical Abstraction as the base abstraction and used it to successfully verify linearizability of data-structures in the presence of an unbounded number of threads.

1 Introduction

This paper is concerned with verifying (basic safety and other functional correctness) properties of dynamically-allocated data structures in programs with an unbounded number of threads. The techniques in this paper, for example, enable, for the first time, automatic verification of the linearizability of various implementations of concurrent data-structures *even when an unbounded number of client threads manipulate these data-structures concurrently*.

Our approach is based on abstract interpretation, which requires us to address the standard two principal challenges:

- to define a finite representation of infinite sets of program states that can concisely and precisely express the properties of interest, and
- to compute sound transformers, which overapproximate a program's semantics using this representation.

Quantification-Based Abstract Domain: The essence of our approach is to use an abstract domain whose elements represent quantified invariants of the form $\forall t.\varphi(t)$, where the quantification is over threads. The formulas $\varphi(t)$ correspond to an abstraction of the program state from the perspective of a thread t. A second aspect of our

approach is that we exploit existing abstractions to capture the component $\varphi(t)$ inside the quantifier. Informally, assume that we have an underlying abstraction where the abstract domain corresponds to a set of formulas A_{Voc} over a vocabulary Voc. (Usually, the vocabulary captures the dependence of the abstract domain on the program being analyzed.) We refine the abstraction and work with the set of formulas $L_{Voc} = \{\forall t.\varphi(t) \mid \varphi(t) \in A_{Voc \cup \{t\}}\}$. Thus, our technique may be seen as a domain constructor. The thread-quantified domain we construct is bounded to the degree that the underlying abstraction domain is: e.g., using a finite-height base domain yields a finite-height quantified domain.

Transformers: A second key contribution of our work relates to the computation of sound abstract transformers. We show how we can compute sound transformers for our new domain using sound transformers for the base domain. We present a simple technique for computing a *basic* sound transformer. The basic transformer works well when a thread's action does not (potentially) affect the invariants (or state) observed by other threads. We also present a more sophisticated technique for computing a *refined* transformer which is useful for thread actions that affect other threads.

Our technique can be used for concurrent *shape analysis* by using suitable shape analysis abstractions such as *Canonical Abstraction* [14] or *Boolean Heaps* [13] as the base domain. We have implemented our technique on top of TVLA [9][4] system, which is based on Canonical Abstraction, and used it to verify *linearizability* of fine-grained concurrency algorithms [1].

The thread-quantified abstract domain is a natural domain to use to reason about programs with an unbounded number of threads. It permits expressing properties that correlate a thread's local variables with each other and with global variables, but not properties that correlate different, ad-hoc, threads' local variables. (By "ad-hoc", we mean properties that cannot be captured using quantification.) Note that when the underlying base domain is disjunctive, as is the case with Canonical Abstraction and Boolean Heaps, the new domain will permit disjunctions inside the quantifier, which is quite useful.

Related Work

Gulwani et al. [6] also present a technique for adding universal quantification around a base abstraction, and use this for proving properties of collections (such as arrays and lists). However, their requirements for the base abstraction is different from ours (as their base domain is an implication domain, while ours is a disjunctive domain). The techniques they use for constructing transformers are also different from ours. Specifically, we address the needs of the transformers in a concurrent setting, unlike Gulwani et al. Thus, the two approaches target different application contexts.

Our work is also closely related to the work of Podelski and Wies [13], where they lift propositional (predicate) abstraction into Boolean Heaps. Our work shows how a similar lifting can be done parametrically over a base abstraction domain, and exploits instantiation of this technique over Canonical Abstraction. Our techniques for generating transformers are also different from that of Podelski and Wies, especially as we target complications arising in a concurrency setting.

[4] We actually implemented our technique in HeDec [10] which generalizes Canonical Abstraction by allowing coarser and more scalable abstractions.

The abstractions we produce may also be thought of as a kind of thread-modular abstractions [5], except that we allow for an unbounded number of threads.

Yahav [16] addresses the same end goal as us: proving properties of programs with an unbounded number of threads manipulating a shared heap. Yahav's approach uses Canonical Abstraction to uniformly handled an unbounded number of threads as well as an unbounded number of heap objects. Our use of Quantified Canonical Abstraction enabled us successfully prove properties not proved previously automatically (using Canonical Abstraction or other abstractions). (It should be noted that the capabilities of Canonical Abstraction depend on the set of predicates used. By defining a richer set of instrumentation predicates, these abstractions can be made more expressive. The advantage of our new approach is that it can reduce the need for manually coming up with nonstandard or program-specific instrumentation predicates to assist verification and it can reduce the number of predicates required.)

Previously, verification approaches have approached an unbounded number of threads by distinguishing between the thread that is currently being reasoned about, and conflating all the threads executing concurrently with it. When the views of different threads are symmetrical, this principle yields a useful abstraction for reasoning about concurrent programs and has been used in several works dedicated to proving properties of concurrent programs such as *Environment Abstraction* [2] and *Invisible Invariants* [12].

The chief difficulty, particularly for domain *constructions*, is defining the transformers: semantics of program statements on elements of the abstract domain. Here, we use a technique based on a heuristic for quantifier instantiation, guided by the intuition that the desired properties can often be established by distinguishing the thread being analyzed from those executing concurrently with it, but conflating the properties of all those interfering threads. This quantifier instantiation heuristic bears a resemblance to that used in the Simplify theorem prover [3]. There, the idea was that given a procedure for the quantifier-free case, the quantified case can be handled using a heuristic of instantiating the universal quantifiers to translate the query into one in the base theory in an incomplete, but hopefully sufficient, way. Simplify generates quantifier-free queries that are checked using an SMT decision procedure, while in our case the translation is to an existing abstract domain.

Main Results

In this paper, we study the utility of abstractions built by universally quantifying over threads. The contributions of this paper can be summarized as follows:

- We present a generic method for lifting abstract domains in order to provide abstractions for concurrent programs with an unbounded number of threads.
- We present techniques for developing sound abstract transformers for the lifted domain by reusing the abstract transformers of the base domain.
- We describe an implementation of a concurrent shape analysis that lifts Canonical Abstraction and use it to verify linearizability of a non-blocking stack implementation from [15] and the two-lock queue implementation described in [11]. We have also extended our previous work that combined heap decomposition with Canonical Abstraction and used it verify the linearizability of a non-blocking queue implementation from [4].

2 Overview

In this section, we present an informal overview of our method.

```
Object g = null; // global variable
threadProc() {
     Object x = null, y = null;
[1]  x = new Object();
[2]  y = x;
[3]  assert(x == y);
     g = x;
[4]  assert(g != null);
}
```

Fig. 1. A simple multithreaded program. The program consists of an unbounded number of threads executing `threadProc`.

A Motivating Example. Fig. 1 shows a toy concurrent program used to illustrate the ideas in this paper. Our ideas, however, are applicable to realistic programs as shown in Sec. 4. The program contains a couple of very simple invariants (expressed as assertions) that we would like to automatically infer. The first invariant is that when a thread is at statement `[3]`, it's `x` and `y` values are equal. This is an example of a *thread-local* invariant (which cannot be affected by the execution of other threads). The second invariant is that when a thread is at statement `[4]`, the value of the global variable `g` is non-null. This is an example of a *non-local thread* invariant, and can be affected by the execution of other threads. In general, a non-local thread invariant could involve global as well as thread-local variables. As an example, consider an assertion that when a thread is at statement `[4]`, the value of `g` and it's `x` are equal. This is an assertion that fails to hold for the given program (because of interaction with other threads).

Background: The Boolean Heap Abstraction. As explained in Sec. 1, our approach is to lift an existing abstraction to produce a more precise abstraction that is suitable for programs with an unbounded number of threads. We will illustrate our idea using Boolean Heaps [13] as the underlying base abstraction in this paper. Boolean Heaps are abstractions targeted at shape analysis, and describe sets of states consisting of an unbounded number of heap objects using formulas of the following form

$$\bigvee_{i \in I} \{\forall v : \mathit{object}.\, \phi'_i(v)\} \tag{1}$$

where v ranges over heap objects, $\phi'(v)$ is a quantifier-free formulas over a set of unary predicates kept in DNF form. For the running example, we will use the predicates shown in Tab. 1. We assume that a *null* value is represented by a special heap object.

The Quantified Boolean Heap Abstraction. As explained earlier, the essence of our approach is to use quantified invariants of the form $\forall t.\, \varphi(t)$ (similar to [6]), where the set of formulas $\varphi(t)$ allowed inside the invariant are determined by a base domain. Using Boolean Heaps as our base domain leads to the following definition of *Quantified*

Table 1. Predicates used for (Quantified) Boolean Heap Abstraction

Predicate	**Intended meaning**
$x(t,v)$	local variable x of thread t points to object v
$y(t,v)$	local variable y of thread t points to object v
$g(v)$	global variable g points to object v
$null(v)$	v is a special null object
$at[l](t)$	thread t is at program label l

Boolean Heaps. Quantified Boolean Heaps approximate sets of states by formulas of the form

$$\forall t\!:\!\mathit{thread}.\ \bigvee_{i\in I}\{\forall v\!:\!\mathit{object}.\ \phi_i(t,v)\} \tag{2}$$

where t ranges over threads, v ranges over non-thread objects, $\phi(t,v)$ is a quantifier-free formulas over a set of unary and binary predicates, kept in DNF form.[5] Notice that a quantified boolean heap is an universally quantified disjunction of standard Boolean heaps, but where some previously unary predicates have been indexed by the universal variable. This increases the expressive power of the abstract elements.

Tab. 2 shows the part of the Quantified Boolean Heap describing the invariant of our running example for the threads in program location `[4]` as computed by our analysis. (In the sequel, we shall use the following notational convention. Logical variables $\{sc, t, t_1, t_2, \dots\}$ range over thread objects and other logical variables range over non-thread objects.) The convention is that if a predicate does not appear in a disjunction, then its negation implicitly holds.

Table 2. Part of the Computed Quantified Invariants for the Running Example

$$\begin{array}{ll}\forall t. & \dots \vee \\ & \forall v.\ \big(at[4](t) \wedge x(t,v) \wedge y(t,v) \wedge g(v)\big) \vee \big(at[4](t) \wedge null(v)\big) \vee \\ & \forall v.\ \big(at[4](t) \wedge x(t,v) \wedge y(t,v)\big) \vee \big(at[4](t) \wedge null(v)\big) \vee \big(at[4](t) \wedge g(v)\big) \vee \\ & \forall v.\ \big(at[4](t) \wedge x(t,v) \wedge y(t,v) \wedge g(v)\big) \vee \big(at[4](t) \wedge null(v)\big) \vee \big(at[4](t)\big) \vee \end{array}$$

Abstract Transformers. Computing abstract transformers is very challenging, especially in the presence of concurrency, as the execution of one thread may affect the state observable by another thread. In Sec. 3 we present effective techniques for computing sound transformers for our lifted abstractions, utilizing transformers of the base abstraction. The main idea is to "instantiate" two symbolic threads, one for the scheduled thread, and one representing another arbitrary thread, and to utilize the transformer of the underlying base domain to compute the change in the abstract state as observed by each of these threads.

[5] This is similar to Indexed Predicate Abstraction [8], except that the number of index variables is limited to 2, and that we allow a disjunction between the quantifiers.

Discussion. Our new analysis is capable of inferring the invariants mentioned in the program: namely, that for any thread t at program location 3 we have $x(t) = y(t)$, and that g is not null at the end (for any thread's execution). In Sec. 1, we contrasted our new abstraction with [16] which naturally models unbounded objects and threads in a uniform fashion using Canonical Abstraction, but cannot infer the above invariants. Indeed, the previous abstraction cannot even express these invariants. The Boolean Heap abstraction, *with the set of predicates described in this section*, has the same limitations. It should be noted that the capabilities of Canonical Abstraction and Boolean Heap abstraction depend on the set of predicates used. By using a richer set of predicates, especially instrumentation predicates, these abstractions can be made more expressive and be used to prove the above invariants. The advantage of the new abstraction is that it can reduce the need for nonstandard or program-specific predicates or the number of predicates in a very natural way.

3 Abstract Domain for Concurrent Shape Analysis

In this section we will describe the lifting abstract domain construction, in the next we will describe a particular instantiation for verifying linearizability.

3.1 The Concrete Semantics

We start by defining operational concrete semantics useful for describing concurrent programs without procedures. For simplicity of presentation we concentrate on reference variables and fields. Let Threads and Locations be countable sets representing threads and heap locations respectively. Let LVars, GVars, and Fields be the local variables, global variables, and heap fields respectively. Finally, let Labels be a finite set of program labels. Let Σ be the set of possible states. A state $\sigma \in \Sigma$ maps the following:

- For each global variable g, $\sigma(\mathsf{g}) \in$ Locations
- For each local variable x, $\sigma(\mathsf{x})$: Threads $\rightarrow$ Locations
- For each field f, $\sigma(\mathsf{f})$: Locations $\rightarrow$ Locations
- $\sigma(\mathsf{pc})$: Threads $\rightarrow$ Labels

A concurrent program is defined by a transition relation TR : Threads $\rightarrow \mathcal{P}(\Sigma \times \Sigma)$. The transition relation gives for each choice of the next scheduled thread the possible state transitions. The overall transition relation is $OTR = \bigcup_{tid \in \mathsf{Threads}} TR(tid)$.

3.2 The Base Abstraction

We present the lifted abstract interpreter as well as the base abstract interpreter as operating on formulas in a normal form. This is done for simplicity of presentation. For example, Boolean Heaps are already presented using these terms. Details on how Canonical Abstraction can be presented in such terms can be found in [17].

We assume a base abstract domain with elements drawn from a set A of formulas, where $(A, \gamma_A, \alpha_A, \mathcal{P}(\Sigma))$ is a Galois Connection, with meet $\sqcap_A$ and join $\sqcup_A$ operations.

Usually, abstract elements correspond to formulas without free variables. However, our framework exploits abstract transformers for formulas with free variables. So, we first formalize the setting for this requirement [13].

Given a set V of free (thread) variables, let $A[V]$ denote the set of formulas in normal form whose free variables are a subset of V.

Let $\text{Assign}_V = V \to \mathsf{Threads}$ be the set of assignments from thread variables in V to threads. A state $\sigma \in \Sigma$ and an assignment $\theta \in \text{Assign}_V$ satisfy $\varphi \in A[V]$, denoted $\sigma, \theta \models \varphi$, when assigning the parameters according to θ and interpreting the predicates according to σ yields true.

Define Σ_V to be $\Sigma \times \text{Assign}_V$. The above notion of satisfaction helps define a Galois Connection between $\mathcal{P}(\Sigma_V)$ and $A[V]$, by defining $\gamma_{A[V]} : A[V] \to \mathcal{P}(\Sigma_V)$ as $\gamma_{A[V]}(\varphi) = \{\langle \sigma, \theta \rangle \mid \sigma, \theta \models \varphi\}$.

Transformers for formulas With Free Variables We start with the concrete transformer induced by the transition relation, i.e, $\text{post} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ such that $\text{post}(S) = \{\sigma' \mid \sigma \in S, \langle \sigma, \sigma' \rangle \in OTR\}$. The Galois Connection $(\mathcal{P}(\Sigma), \alpha, \gamma, A)$ directly leads to the notion of a sound transformer $\text{post}^\sharp : A \to A$. Note that $\text{post}^\sharp$ transforms closed formulas.

We define the lifted concrete transformer $\text{post}_V : \mathcal{P}(\Sigma_V) \to \mathcal{P}(\Sigma_V)$ as follows: $\text{post}_V(S) = \{(\sigma', \theta) \mid (\sigma, \theta) \in S, \sigma' \in \text{post}(\{\sigma\})\}$.

Now, we can define the notion of soundness for transformers of formulas with free variables. We say that $\text{post}^\sharp_V : A[V] \to A[V]$ is a sound approximation for the concrete transformer $\text{post} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ if it is a sound approximation of the lifted concrete transformer $\text{post}_V : \mathcal{P}(\Sigma_V) \to \mathcal{P}(\Sigma_V)$.

3.3 The Lifted Abstraction (With Basic Transformers)

We define the lifted domain $L = \{\forall t.\, \varphi(t) \mid \varphi(t) \in A[t]\}$, i.e., with the base domain instantiated with $V = \{t\}$. The lifted domain inherits meet and join operations from $A[t]$: e.g., $(\forall t.\, \varphi_1) \sqcup_L (\forall t.\, \varphi_2) = \forall t.\, (\varphi_1 \sqcup_{A[t]} \varphi_2)$.

We define a Galois Connection from $\mathcal{P}(\Sigma)$ to L by defining $\gamma_L : L \to \mathcal{P}(\Sigma)$ by $\gamma_L(\varphi(t)) = \bigcap_{\theta \in \text{Assign}_{\{t\}}} \big\{\sigma \mid \langle \sigma, \theta \rangle \in \gamma_{A[t]}(\varphi(t))\big\}$.

Consider a concrete transformer $\text{post} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. We obtain a sound transformer $\text{post}^\sharp_L : L \to L$ as follows: $\text{post}^\sharp_L(\forall t.\varphi(t)) = \forall t.\text{post}^\sharp_{\{t\}}(\varphi(t))$. Recall that $\text{post}^\sharp_{\{t\}}$ is the sound abstract transformer for formulas with a free variable t discussed earlier.

Consider the statement `y=x` from the example program in Fig. 1 and the abstract state $S1$ shown below. We write only the positive predicates for each location, the rest of the predicates are implicitly negated.

$$\begin{aligned} S1 &= \forall t.\ S1_a(t) \vee S1_b(t) \\ S1_a(t) &= \forall v.\ \big(\mathit{at}[1](t) \wedge x(t,v) \wedge y(t,v) \wedge g(v) \wedge \mathit{null}(v)\big) \vee \\ &\qquad \big(\mathit{at}[1](t)\big) \vee \\ S1_b(t) &= \forall v.\ \big(\mathit{at}[2](t) \wedge y(t,v) \wedge g(v) \wedge \mathit{null}(v)\big) \vee \\ &\qquad \big(\mathit{at}[2](t) \wedge x(t,v)\big) \end{aligned} \tag{3}$$

Applying the Boolean heap transformer for y=x to $S1_a(t)$ yields $S1'_a(t)$. Applying the Boolean heap transformer for y=x to $S1_b(t)$ yields the heaps $S1'_{b_1}(t)$ for the case t is the scheduled thread, and $S1'_{b_2}(t)$ for the complementary case. The final result is obtained by universally quantifying over t, resulting in $S1'$.

$$\begin{aligned} S1' &= \forall t.\ S1'_a(t) \vee S1'_{b_1}(t) \vee S1'_{b_2}(t) \\ S1'_a(t) &= \forall v.\ \big(at[1](t) \wedge x(t,v) \wedge y(t,v) \wedge g(v) \wedge null(v)\big) \vee \\ &\qquad \big(at[1](t)\big) \\ S1'_{b_1}(t) &= \forall v.\ \big(at[3](t) \wedge g(v) \wedge null(v)\big) \vee \\ &\qquad \big(at[3](t) \wedge x(t,v) \wedge y(t,v)\big) \\ S1'_{b_2}(t) &= \forall v.\ \big(at[2](t) \wedge y(t,v) \wedge g(v) \wedge null(v)\big) \vee \\ &\qquad \big(at[2](t) \wedge x(t,v)\big) \end{aligned} \tag{4}$$

Let $\phi_{\texttt{x==y}} = \forall t.at[3](t) \rightarrow \forall v.x(t,v) \leftrightarrow y(t,v)$ be the assertion at line [3]. Now, $S1' \models \phi_{\texttt{x==y}}$ (the only disjunct where $at[3]$ holds is $S1'_{b_1}(t)$, in which x and y point to the same node). The statement y=x changes only information local to one thread and therefore this kind of reasoning is sufficiently precise.

Let $\phi_{\texttt{g!=null}} = \forall t.at[4](t) \rightarrow \forall v.\neg(g(v) \wedge null(v))$ be the assertion at line [4]. Now, however, consider the statement g=x and the abstract state $S2$.

$$\begin{aligned} S2 &= \forall t.\ S2_a(t) \\ S2_a(t) &= \forall v.\ \big(at[3](t) \wedge x(t,v) \wedge y(t,v)\big) \vee \\ &\qquad \big(at[3](t) \wedge g(v) \wedge null(v)\big) \vee \\ &\qquad \big(at[3](t)\big) \end{aligned} \tag{5}$$

When t is not the scheduled thread, applying the Boolean heap transformer to the Boolean heap $S2_a(t)$ yields many Boolean heaps. This is because of the lack of information about the status of other threads, which we get by dropping the universal quantification over t. The scheduled thread may be different from t. Thus, $S2_a(t)$ has no information about it. In particular, $S2_a(t)$ represents a state where x and y are null for the scheduled thread. As a result, the assignment g=x also creates the Boolean heap $S_{bad} = \forall v.\big(at[4](t) \wedge x(t,v) \wedge y(t,v) \wedge g(v) \wedge null(v)\big)$. Obviously, $S_{bad} \not\models \phi_{\texttt{g!=null}}$ and thus the transformer is not precise enough for the purpose of our analysis.

The reason is that g=x changes a global variable. This change is visible by other threads and thus the "local" reasoning used above does not model the effect of the other threads using the information captured by the quantified Boolean heap.

3.4 The Semantics of Non-Deterministic Scheduling

In order to obtain a more precise, sound transformer for our lifted abstract domain, we need to exploit the internal structure of the transition relation and the base abstract transformer imposed by the semantics of non-deterministic scheduling.

Recall that a concurrent program is defined by a *base* transition relation $TR : \mathsf{Threads} \rightarrow \mathcal{P}(\Sigma \times \Sigma)$. This function indicates the transitions a given thread tid can take. The semantics of non-deterministic scheduling of threads is captured by defining the overall transition relation $OTR : \mathcal{P}(\Sigma \times \Sigma)$ by $OTR = \bigcup_{tid \in \mathsf{Threads}} TR(tid)$.

In the base abstraction, we assume that a sound transformer for OTR is obtained from a sound transformer for the base transition function TR as follows. Assume that for any free thread variable sc not in V, we have $TR^\sharp_{V,sc} : A[V] \rightarrow A[V \cup \{sc\}]$ a sound approximation for TR[6]. Lifting the transformers of the base abstraction to support receiving the scheduled thread externally is usually straightforward.

Then, the function $OTR^\sharp_{A[V]} : A[V] \rightarrow A[V]$ is defined as follows $OTR^\sharp_{A[V]}(\varphi) = \mathsf{project}(sc, TR^\sharp_{V,sc}(\varphi(V)))$, where sc is a free variable not in V.

In the above definition, we assume that the base abstraction domain has an operation project for projecting away a thread parameter sc. This is equivalent to overapproximating existential elimination. For example, in Boolean Heaps, we can simply throw away all literals (positive and negative) that contain sc.

We now present a refined transformer $OTR^\sharp_L : L \rightarrow L$ defined by

$$OTR^\sharp_L(\varphi(t)) = \mathsf{project}(sc, TR^\sharp_{\{t\},sc}(\varphi(t) \sqcap \varphi(sc)))$$

. The observation here is that the only thread that changes the state is the scheduled thread. The other threads are observing the interference done by the scheduled thread. Thus, we need to accurately represent the state from both the perspective of the scheduled thread and from the perspective of t.

We demonstrate the refined transformer by computing $OTR^\sharp_L(S2_a(t))$. The first step of the transformer is to compute the meet of $S2_a(t)$ and $S2_a(sc)$ as shown below[7]

$$\begin{aligned}\varphi(sc,t) = S2_a(t) \sqcap S2_a(sc) = (\forall v.\ &\big(at[3](t) \wedge x(t,v) \wedge y(t,v)\big) \vee \\ &\big(at[3](t) \wedge g(v) \wedge null(v)\big) \vee \\ &\big(at[3](t)\big)) \wedge \\ (\forall v.\ &\big(at[3](sc) \wedge x(sc,v) \wedge y(sc,v)\big) \vee \\ &\big(at[3](sc) \wedge g(v) \wedge null(v)\big) \vee \\ &\big(at[3](sc)\big))\end{aligned} \tag{6}$$

[6] For every $\langle \sigma, \theta \rangle \in \gamma_{A[V]}(\varphi(V))$ and $\langle \sigma, \sigma' \rangle \in TR(tid)$ we have $\langle \sigma', \theta[sc := tid] \rangle \in \gamma_{A[V \cup \{sc\}]}(TR^\sharp_{V,sc}(\varphi(V)))$.

[7] For brevity, we have not converted the formula to DNF.

Applying the Boolean heaps transformer on $\varphi(sc,t)$ we get a disjunction of two heaps one for the case in which $sc = t$ and one for the case in which $sc \neq t$.

$$
\begin{array}{rl}
\varphi'(sc,t) = & \varphi'_a(sc,t) \vee \varphi'_b(sc,t) \\
\varphi'_a(sc,t) = \forall v.\, (& \big(at[4](sc) \wedge x(sc,v) \wedge y(sc,v) \wedge g(v)\big) \vee \\
& \big(at[4](sc) \wedge null(v)\big) \vee \\
& \big(at[4](sc)\big) \qquad\qquad\qquad\qquad\qquad\qquad\quad) \wedge \\
& (\big(at[4](t) \wedge x(t,v) \wedge y(t,v) \wedge g(v)\big) \vee \\
& \big(at[4](t) \wedge null(v)\big) \vee \\
& \big(at[4](t)\big) \qquad\qquad\qquad\qquad\qquad\qquad\quad) \\
\varphi'_b(sc,t) = \forall v.\, (& \big(at[4](sc) \wedge x(sc,v) \wedge y(sc,v) \wedge g(v)\big) \vee \\
& \big(at[4](sc) \wedge null(v)\big) \vee \\
& \big(at[4](sc)\big) \qquad\qquad\qquad\qquad\qquad\qquad\quad) \wedge \\
& (\big(at[3](t) \wedge x(t,v) \wedge y(t,v)\big) \vee \\
& \big(at[3](t) \wedge null(v)\big) \vee \\
& \big(at[3](t)\big) \qquad\qquad\qquad\qquad\qquad\qquad\quad)
\end{array}
\tag{7}
$$

We project away sc by removing all literals containing it. This yields the follow result:

$$
\begin{array}{rl}
\varphi''(t) = & \varphi''_a(t) \vee \varphi''_b(t) \\
\varphi''_a(t) = \forall v.\, & \big(at[4](t) \wedge x(t,v) \wedge y(t,v) \wedge g(v)\big) \vee \\
& \big(at[4](t) \wedge null(v)\big) \vee \\
& \big(at[4](t)\big) \\
\varphi''_b(t) = \forall v.\, & \big(at[3](t) \wedge x(t,v) \wedge y(t,v)\big) \vee \\
& \big(at[3](t) \wedge g(v)\big) \vee \\
& \big(at[3](t) \wedge null(v)\big) \vee \\
& \big(at[3](t)\big)
\end{array}
\tag{8}
$$

The interesting observation here is that `g` and `null` are not aliased in both conjuncts of $\varphi'_b(sc,t)$. Thus, after the projection we retain this information.

Finally, the result is universally quantified i.e., ${S_2}' = \forall t.\varphi''(t)$. As expected ${S_2}' \models \phi_{\texttt{g!=null}}$.

4 Case Study: Proving Linearizability

As a case study for the approach we have verified linearizability of three well known concurrent data structure implementations that use fine-grained concurrency.

4.1 Implementation

We have implemented the approach on top of TVLA [9][8]. The thread parameters were implemented as unary predicates. Support for treating a binary formula of the form

[8] We actually implemented our technique in HeDec [10] which generalizes Canonical Abstraction by allowing coarser and more scalable abstractions.

$x(t, v)$ as a unary predicate was done by adding an appropriate instrumentation predicate (i.e., predicate defined using a formula from other predicate and automatically updated by the system).

The meet and join operations required from the base domain are already implemented in TVLA. Thread projection is done by forgetting all information about the unary predicate representing the thread and all instrumentation predicates based on it.

In TVLA it is easier to implement separate transformers for each statement and let the engine deal with constructing the full post operator. As a result we are able to use the basic transformer for some statements and the more expensive refined transformer only for statements that require the extra precision. We use the basic transformer for statements which modify only the local state of the scheduled thread and leave the global state intact. In these cases the abstract state of any thread that is not the scheduled thread is unchanged by the operation, thus the precision of the basic transformer is enough.

4.2 Proving Linearizability

Linearizability [7] is one of the main correctness criteria for implementations of concurrent data structures. Informally, a concurrent data structure is said to be linearizable if the concurrent execution of a set of operations on it is equivalent to some sequential execution of the same operations, in which the global order between non-overlapping operations is preserved. The equivalence is based on comparing the arguments and results of operations (responses). The permitted behavior of the concurrent object is defined in terms of a specification of the desired behavior of the object in a sequential setting.

Verifying linearizability is challenging because it requires correlating any concurrent execution with a corresponding permitted sequential execution. Verifying linearizability for concurrent dynamically allocated linked data structures is particularly challenging, because it requires correlating executions that may manipulate memory states of unbounded size.

Intuitively, we verify linearizability by representing in the concrete state both the state of the concurrent program and the state of the reference sequential program. Each element entered into the data structure is correlated at linearization points with the matching object from the sequential execution. The details are described in [1].

We have taken the instantiation of Canonical Abstraction presented in [1] as the base abstraction for the analysis. The analysis of [1] was performed for a bounded number of threads, by using specialized predicates treating each local variable of each thread as a separate predicate. We removed these extra predicates, instead treating the thread local variables as binary predicates. The analysis has predicates for local and global variables, heap fields and program labels. Finally, we use as is two extra predicates that capture the correlation between the concurrent and sequential executions (see [1]).

4.3 Experimental Results

Tab. 3 summarizes the experimental results of running our analysis on linearizability algorithms. These benchmarks were run a 2.4GHz E6600 Core 2 Duo processor with

2 GB of memory on running Linux. We used two abstractions to analyze these examples. The first is vanilla canonical abstraction as described in Sec. 4.2. The second abstraction is an extension of canonical abstraction with decomposition of the heap as described in [10]. With this abstraction, the state space is significantly reduced, yielding fewer states and better times. The adaptation of the transformer for the decomposing abstraction was no harder than that for vanilla canonical abstraction.

The stack algorithm is described in [15]. The algorithm is lock-free and uses a Compare And Swap (CAS) operation for synchronization. The two-lock queue algorithm is described in [11]. The queue has Head and Tail pointers, each protected with its own lock. The algorithm allows benign data-races in case the queue is empty, i.e., the Head and Tail pointers are aliased. The non-blocking queue algorithm is described in [4]. The algorithm is lock-free and uses CAS for synchronization. It is more complicated than the other two algorithms and has a much larger state space in our abstraction. As a result, canonical abstraction without decomposition on this example resulted in a state space explosion.

Table 3. Experimental results of proving linearizability for an unbounded number of threads

	Canonical Abstraction		with decomposition	
Algorithms	States	secs.	States	secs.
Stack [15]	4,097	53	764	7
Two-lock queue [11]	4,897	47	3,415	17
Non-blocking queue [4]	MemOut	MemOut	10,333	252

5 Conclusion

In this paper we have developed a new shape analysis for fine-grained concurrent programs with unbounded number of threads and demonstrated that it is precise enough to prove linearizability of useful data structure implementations. This is done by universally lifting domain construction applied to existing shape analysis domains.

We believe that universal lifting can be also used for interprocedural shape analysis by universally quantifying over all activation records. This can allow to track correlations between values of local variables of different procedure incarnations.

References

1. D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *19th International Conference on Computer Aided Verification (CAV)*, 2007.
2. E. Clarke, M. Talupur, and H. Veith. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*, 2008.
3. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

4. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In *FORTE*, 2004.
5. Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 213–224. Springer, 2003.
6. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, 2008.
7. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
8. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Logic*, 9(1):1–29, 2007.
9. T. Lev-Ami and M. Sagiv. TVLA: A framework for implementing static analyses. In Jens Palsberg, editor, *Proc. Static Analysis Symp.*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000. Available from http://www.cs.tau.ac.il/∼tvla/.
10. R. Manevich, T. Lev-Ami, M. Sagiv, G. Ramalingam, and J. Berdine. Heap decomposition for concurrent shape analysis. Technical Report TR-2007-11-85453, Tel Aviv University, November 2007. Available at http://www.cs.tau.ac.il/∼rumster/TR-2007-11-85453.pdf, submitted for publication.
11. M.M. Michael and M.L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, 1996.
12. A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *TACAS*, 2001.
13. A. Podelski and T. Wies. Boolean heaps. In *SAS*, pages 268–283, 2005.
14. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
15. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.
16. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, March 2001.
17. G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *TACAS*, 2004.