# Automatic Verification of Temporal Properties of Concurrent Heap-Manipulating Programs using Evolution Logic

**Tel Aviv University, School of Computer Science, TR 338/02**

Eran Yahav[1], Thomas Reps[2], Mooly Sagiv[1], and Reinhard Wilhelm[3]

[1] School of Computer Science, Tel-Aviv University, Tel-Aviv 69978,
{yahave,msagiv}@post.tau.ac.il
[2] Computer Science Dept., University of Wisconsin, Madison, WI 53706; USA,
reps@cs.wisc.edu
[3] Informatik; Universität des Saarlandes, 66123 Saarbrücken; Germany,
wilhelm@cs.uni-sb.de

**Abstract.** This paper addresses the problem of establishing temporal properties of programs written in languages, such as Java, that make extensive use of the heap to allocate—and deallocate—new objects and threads. Establishing liveness properties is a particularly hard challenge. One of the crucial obstacles is that heap locations have no static names and the number of heap locations is unbounded. The paper presents a framework for the verification of Java-like programs. We use first-order modal logic to specify temporal properties of heap evolutions; this logic is a simple variant of existing first-order modal logics that allow expressing universe changes. The paper also presents an abstract-interpretation algorithm that automatically verifies temporal properties expressed using the logic.

## 1 Introduction

Modern programming languages, such as Java, make extensive use of the heap. The contents of the heap may evolve during program execution due to dynamic allocation and deallocation of objects. Moreover, in Java, threads are first-class objects that can be dynamically allocated. Statically reasoning about temporal properties of such programs is quite challenging, because there are no a-priori bounds on the number of allocated objects, or restrictions on the way the heap may evolve. In particular, proving liveness properties of such programs, e.g., that a thread is eventually created in response to a request made to a web server, can be a quite difficult task.

### 1.1 Main Results and Comparison to Previous Work

The contributions of this paper can be summarized as follows:

1. We introduce a first-order modal (temporal) logic [10, 9], that allows to give natural specifications of temporal properties of programs with dynamically evolving heaps.
2. We develop an abstract interpretation [4] for verifying that a program satisfies such a specification.
3. We implemented a prototype of the analysis using the TVLA system [12].
4. We applied this implementation to verify several temporal properties, including liveness properties of Java programs with evolving heaps.

The rest of this section elaborates on these contributions and contrasts them with previous work. More related work is discussed in the full paper.

**A Temporal Logic Supporting Evolution** The specification language, *Evolution Logic*, is a first-order linear temporal logic, which allows specifying properties of the way program execution causes the heap to evolve.

It is natural to consider the concrete semantics of a program as the set of its execution traces [5, 18], i.e., infinite sequences of configurations, later on called *worlds*. First-order logical structures provide a natural representation of configurations with an unbounded number of objects: an individual of the structure's universe corresponds to an anonymous, unique store location, and predicates represent properties of store locations. Such a representation is *storeless*, and allows properties of the the heap contents to be maintained while abstracting from the actual physical store location.

This gives rise to traces in which configurations along the trace may have different universes. Such traces can be seen as models of a first-order modal logic with a varying-domain semantics [9].

This framework generalizes other specification methods that address dynamic allocation and deallocation of objects and threads. In particular, its descriptive power goes beyond PLTL and finite-state machines (e.g., [2]).

[3] introduces the Bandera Specification Language (BSL), which allows the user to write complicated specifications via common high-level patterns. BSL defines observable primitive predicates that are used as propositions of the temporal specification. Thus, BSL properties correspond to the class of temporally separable properties (see Section 2.2) in which it is impossible to relate individuals of different configurations. Moreover, it is impossible to refer to the exact moments of allocation and deallocation of an object in BSL.

Program properties can be verified by showing that they hold for all traces. Technically, this can be done by evaluating their first-order modal-logic formulae against all traces. We use a variant of Lewis's counterpart theory in order to cast modal models (and formulae evaluation) in terms of classical predicate logic with transitive closure [13]. However, while Lewis quickly turns away from adopting a counterpart relation that is reflexive, transitive, and symmetric, these are the exact characteristics in which we are interested for our transworld-equality relation (see Section 4.3).

**Abstract Interpretation of the Trace Semantics** Program verification using the above concrete semantics is clearly non-computable in general. We therefore represent potentially infinite sets of infinite concrete traces by one abstract trace. Infinite parts of the concrete traces are folded into cycles of the abstract traces. Termination of the abstract interpretation on an arbitrary program is guaranteed by bounding the size of the abstract trace. Three abstractions are employed: (i) explicitly representing finite prefixes of the infinite trace, (ii) representing multiple concrete configurations by a single abstract configuration, and (iii) creating cycles when the same abstract configuration reoccurs in the trace.

Because of this simple abstraction, we may fail to show the correctness of certain programs, even though they are correct. Fortunately, we can use reduction mechanisms as employed in program verification. These reductions replace a formula that specifies a property by a weaker formula, which, together with an assumption, implies the original one. For instance, one can verify that a loop that checks the condition `x != NULL`, where `x` points to a linked list, will always terminate by showing that on every loop iteration the set of memory locations reachable from `x` decreases. Another example is using fairness to simplify the task of proving liveness.

As in finite-state model-checking (e.g., [18]), we let the specification formula affect the abstraction by making sure that abstract traces that fulfill the formula are distinguished from the ones that do not. However, our abstraction does not fold the history of the trace into a single state. This idea of using the specification to affect the precision of the analysis was not used in [17, 20]. The above cited works only handled safety properties.

A lot of research has been aimed at verifying properties of parametric systems in which the size of the system depends on a parameter. One approach for the verification of parametric systems is the construction of a *network invariant* simulating the behavior of an environment that consists of an arbitrary number of threads [19]. In this approach, a network invariant is first constructed, and then used as an environment in the verification of a single thread. Construction of a network invariant can be automated in some cases [1], but often requires some insight and understanding of system behavior. In contrast, when using our abstract interpretation abstract traces are constructed that represent the network invariants, as well as other properties of data.

In [15], a special case of the abstraction from [20, 21], named "counter abstraction", is used to abstract an infinite-state parametric system into a finite-state one. They use static abstraction, i.e., have a preceding model-extraction phase, while abstraction in this paper is applied dynamically on every step of state-space exploration, enabling us to handle dynamic allocation and deallocation of objects and threads. Note that program behavior may depend on the allocation and deallocation patterns, and not all systems may be turned to parametric ones in which all threads are pre-allocated.

In [21], we have used observing-propositions defined over a first-order configuration to extract a propositional Kripke structure from a first-order one. The extracted propositional structure was than subject to classic PLTL model-

checking techniques. This approach is rather limited, because individuals of different configurations could not be specifically related. In Section 2.2, we state this approach in terms of our current trace-based framework.

**Prototype Implementation** We have implemented a prototype of the framework, and used it to verify temporal properties of small but interesting programs. Results are reported in Section 5.

**Applications** We have used the framework to specify the following properties, and verify some of them.

*Termination of sequential heap-manipulating programs*: The framework has been applied to verify the termination of sequential heap-manipulating programs. Termination is shown by providing a ranking-function based on the set of items reachable from a variable iterating over the linked data structure. In particular, we have verified termination of all example programs from [6].

*Temporal properties of concurrent heap-manipulating programs*: We have used the framework to verify temporal properties of concurrent heap-manipulating programs—in particular, liveness properties, such as the absence of starvation in programs using mutual exclusion, and response [14] properties. We have also applied the framework to programs with an unbounded number of threads.

*Specification of general heap-evolution properties*: The framework has been used to specify in a general manner, various properties of heap-evolution, such as properties of garbage-collection algorithms.

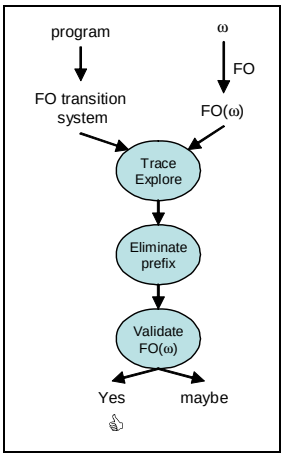


**Fig. 1.** Framework Overview

### 1.2 Running Example

Consider a web server in which threads are dynamically allocated in response to received `http` requests. Whenever a request is received, a new thread is allocated to handle it. Assume that worker-threads are competing for some exclusively shared resource, such as exclusive access to a data file, and that each thread handles a single request and then terminates and is subject to garbage collection. Figure 2 shows fragments of a Java program implementing such a naive web server.

Dynamic allocation of threads is common when implementing servers [8]. Even when using thread-pools to avoid the cost of rapid construction and garbage collection of thread objects, the size of the thread pool is usually adapted dynamically to improve server responsiveness [11].

```
public class Listener implements Runnable {
  ...
  public void run() {
    while(true) {
      ...
      request = requestStream.readObject();
      ...
      worker = new Thread(new Worker(request));
      worker.start();
      ...
    } } }
public class Worker implements Runnable {
  Request myRequest;
  Resource myResource;
  ...
  public void run() {
    ...
    synchronized(myResource) {                                    lw1
      ...
      myResource.processRequest(myRequest);                       lwc
      ...
    }                                                             lw2
  } }
```

**Fig. 2.** Java code fragment for a web-server with no explicit scheduling

A number of interesting properties for the naive web server implementation are shown in Table 1 as properties P1–P4.

Due to the unbounded arrival of requests to the web server, and the fact that a thread is dynamically created for each request, property P2 does not hold for

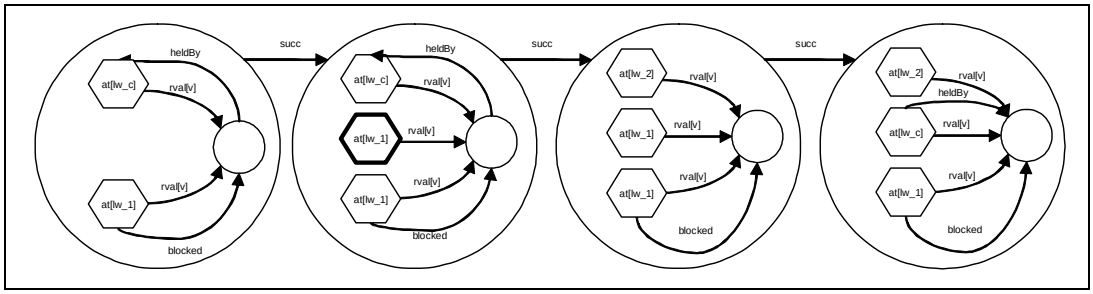


**Fig. 3.** A trace fragment demonstrating barging: a newly allocated thread acquires the lock before a thread that was already blocked on the lock.

the naive implementation (e.g., see Figure 3). In order to guarantee the absence of starvation (P2), we introduce a scheduler thread into the web server. The web server now consists of a listener thread (as before) and a queue of worker threads managed by the scheduler thread. The listener thread receives an `http` request, creates a corresponding worker thread, and places the new thread on a scheduling queue. The scheduler thread picks up a worker thread from the queue and starts its execution (which is still a very naive implementation).

When using a web server with a scheduler, a number of additional properties of interest exist, labeled P5–P8. These are accompanied by the queue-related properties, Q1–Q4.

Figure 4 shows fragments of a web-server program in which threads use an explicit FIFO scheduler that imitates the scheduling policy implemented in most JVMs in practice.

### 1.3 Overview of the Verification Procedure

Figure 1 provides an overview of how a temporal property involving heap evolution is verified. First, the property $\varphi$ is specified in ETL . This is by no mean a trivial task, as is common for temporal-logic-based specifications. The formula is then translated in a straightforward manner into a first-order logical formula, $FO(\varphi)$, using a translation procedure described in Appendix A. An abstract-interpretation procedure "Trace Explore" is then applied to explore finite prefixes of the trace, using 3-valued logic to interpret formulae. A post-processing phase, "Eliminate Prefix", eliminates trace prefixes that are not maximal, i.e., are a prefix of other explored traces. Finally, the formula $FO(\varphi)$ is validated against the remaining prefixes. When $FO(\varphi)$ is satisfied by all remaining (finite) traces, it is guaranteed that the original ETL formula $\varphi$ satisfies all infinite traces. However, it may be the case that for some programs that satisfy the ETL specification, our analysis only yields "maybe".

```
public class Listener implements Runnable {
  protected Queue schedQueue;
  ...
  public void run() {
    while(true) {                                                    la1
      ...
      request = requestStream.readObject();                          la2
      ...
      worker = new Thread(Worker(request));                          la3
      schedQueue.enqueue(worker);                                    la4
      ...
    } } }
public class Scheduler implements Runnable {
  protected Queue schedQueue;
  protected Resource protectedResource;
  ...
  public void run() {
    while(true) {                                                    ls1
      ...
      synchronized(protectedResource) {                              ls2
        while(protectedResource.isAcquired())
          protectedResource.wait();                                  ls3
        // may block until queue is not empty
        worker = schedQueue.dequeue();                               ls4
        worker.start();                                              ls5
      } } } }
public class Worker implements Runnable {
  Request myRequest;
  Resource myResource;
  ...
  public void run() {
    synchronized(myResource) {                                       lw1
      ...
      myResource.processRequest(myRequest);                          lwc
      ...
      myResource.notifyAll()
    }                                                                lw2
  } }
```

**Fig. 4.** Java code fragment for a web-server with an explicit scheduler

| Pr. | Desc | Formula |
|---|---|---|
| P1 | mutual exclusion over the shared resource | $\Box\forall t_1, t_2 : thread.(t_1 \neq t_2)$ $\rightarrow \neg(at[lw_c](t_1) \wedge at[lw_c](t_2))$ |
| P2 | absence of starvation for worker threads competing for shared resource | $\Box\forall t : thread.at[lw_1](t) \rightarrow \Diamond at[lw_c](t)$ |
| P3 | a thread is only created when a request is received | $\Box(\forall t : thread.\neg \odot t) \vee$ $(\forall t : thread.\neg \odot t)\ \mathcal{U}\ (\exists v : request. \odot v)$ |
| P4 | each request is followed by thread creation | $\Box\exists v : request. \odot v$ $\rightarrow \Diamond\exists t : thread. \odot t$ |
| P5 | mutual exclusion of listener and scheduler over scheduling queue | $\Box\forall t_1, t_2 : thread.(t_1 \neq t_2)$ $\rightarrow \neg(at[ls_2](t_1) \wedge at[la_3](t_2))$ |
| P6 | each created thread is eventually inserted to the scheduling queue | $\Box\forall t : thread. \odot t$ $\rightarrow \Diamond\exists q : queue.\exists h.rval[head](q, h) \wedge rval[next]^*(h, t)$ |
| P7 | each scheduled worker thread was removed from the scheduling queue | $\Box\forall t : thread.at[lw_1](t)$ $\rightarrow \neg\exists q : queue.\exists h.rval[head](q, h) \wedge rval[next]^*(h, t)$ |
| P8 | each worker thread waiting in the queue eventually leaves the queue | $\exists q : queue.\Box\forall t : thread.$ $((\exists h : rval[head](q, h) \wedge rval[next]^*(h, t))$ $\rightarrow \Diamond\neg(\exists h : rval[head](q, h) \wedge rval[next]^*(h, t)))$ |
| Q1 | when no queue action is currently executing, the tail is reachable from the head | $\Box\exists v_1 : rval[tail](q, v_1)$ $\rightarrow \exists h : rval[head](q, h) \wedge rval[next]^*(h, v_1)$ |
| Q2 | a thread created by listener is eventually inserted at queue tail | $\Box\exists t_l : thread.\exists v.at[la_5](t_l) \wedge rval[x_i](t_l, v)$ $\rightarrow \Diamond\exists q : queue.rval[tail](q, v)$ |
| Q3 | the head of queue is eventually taken by scheduler | $\Box\exists t_s : thread.\exists q : queue.\exists v.at[ls_4](t_s)$ $\wedge rval[head](q, v) \rightarrow \Diamond rval[x_d](t_s, v)$ |
| Q4 | head always points to the first element of the queue | $\Box\forall v_1, v_2\forall q : queue.rval[head](q, v_2)$ $\rightarrow \neg rval[next](v_1, v_2)$ |

**Table 1.** Web server ETL specification

### 1.4 Paper Outline

Section 2 introduces trace-semantics based on first-order modal logic, and shows how to extract trace properties by using the language of first-order linear-temporal logic. Section 3 gives an implementation of this concrete semantic via first-order logic. Section 4 shows how abstract traces are used to conservatively represent sets of concrete traces. Section 5 gives a short description of the prototype implementation. Finally, Section 6 concludes the paper and discusses future work.

## 2 Trace Semantics

We consider the semantics of a program to be the set of its possible infinite traces. This means that the exit of the program is connected to itself via a special skip action making the transition system total. Each trace consists of a sequence of configurations.

**Definition 1.** *[Configuration] A program configuration is represented via a first-order logical structure $W = \langle U_w, \iota_w \rangle$, where $U_w$ is the universe of the structure, and $\iota_w$ is the interpretation function mapping predicates to their truth values, that is, for each $p \in P$ of arity $k$, $\iota_w(p) \colon U_w^k \to \{0, 1\}$.*

**Definition 2.** *[Trace] A trace is an infinite sequence of configurations $\pi_1 \xrightarrow{D_{\pi_1}, e_{\pi_1}, A_{\pi_2}} \pi_2 \xrightarrow{D_{\pi_2}, e_{\pi_2}, A_{\pi_3}} \ldots$, where: (i) each configuration represents a global state of the program, $\pi_1$ is an initial state, and for each $\pi_i$, its successor configuration $\pi_{i+1}$ is derived by applying a single program action to $\pi_i$; (ii) $D_{\pi_i} \subseteq U_{\pi_i}$ is the set of individuals deallocated at $\pi_i$, and $A_{\pi_{i+1}} \subseteq U_{\pi_{i+1}}$ is the set of individuals newly allocated at $\pi_{i+1}$; (iii) each pair of consecutive configurations $\pi_i, \pi_{i+1}$ is related by a stepwise evolution function, a bijective renaming function $e_{\pi_i} \colon U_{\pi_i} \setminus D_{\pi_i} \to U_{\pi_{i+1}} \setminus A_{\pi_{i+1}}$.*

### 2.1 Extracting Trace Properties

In order to extract trace properties, we need a language that can relate information from different worlds (configurations in the trace). We define the language of evolution logic (ETL), which is a first-order linear temporal logic with transitive closure as follows:

**Definition 3.** *[ETL Syntax]*

1. *a logical literal $\varphi = \boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}\}$ is an atomic formula with no free variables $FV(\varphi) = \emptyset$. A logical formula $\varphi = p(x_1, \ldots, x_n)$ where $p$ is an $n$-ary predicate is an atomic formula with $FV(\varphi) = \{x_1, \ldots, x_n\}$. If $x$ is a logical variable, $\varphi = \odot x$ (resp. $\varphi = \oslash x$) is an atomic formulae, and $FV(\varphi) = \{x\}$ The formula $\varphi = (x_1 = x_2)$ is an atomic formula with $FV(\varphi) = \{x_1, x_2\}$.*

2. *if* $\varphi, \psi$ *are formulae, then* $\theta_1 = (\varphi \vee \psi)$ *and* $\theta_2 = \neg\varphi$ *are formulae, where* $FV(\theta_1) = FV(\varphi) \cup FV(\psi)$ *and* $FV(\theta_2) = FV(\varphi)$
3. *if* $\varphi, \psi$ *are formulae, then* $\theta_1 = \varphi \; \mathcal{U} \; \psi$ *and* $\theta_2 = \chi\varphi$ *are formulae, where* $FV(\theta_1) = FV(\varphi) \cup FV(\psi)$, *and* $FV(\theta_2) = FV(\varphi)$
4. *if* $\varphi$ *is a formula, and* $x \in FV(\varphi)$, *then* $\exists x.\varphi$ *is a formulae with free variables* $FV(\varphi) \setminus \{x\}$
5. *if* $\varphi$ *is a formula such that* $v_1, v_2 \in FV(\varphi)$ *and* $v_3, v_4 \notin FV(\varphi)$, *then* $(TC \; v_1, v_2 \colon \varphi)((v_3, v_4))$ *is a formula with free variables* $(FV(\varphi) \setminus \{v_1, v_2\}) \cup \{v_3, v_4\}$

The designated operators $\odot$ and $\oslash$ allow the specification to refer to the exact moments of birth and death (respectively) of an individual.[1]

*Shorthand Formulae*: For convenience, we also allow formulae to contain the usual shorthand notations $\varphi \wedge \psi = \neg(\neg\varphi \vee \neg\psi)$, $\varphi \rightarrow \psi = \neg\varphi \vee \psi$, $\forall x.\varphi = \neg(\exists x.\neg\varphi)$, $\Diamond\varphi = T \; \mathcal{U} \; \varphi$, and $\Box\varphi = \neg(T \; \mathcal{U} \; \neg\varphi)$. We also use the shorthand $p^*(v_3, v_4)$ for $(TC \; v_1, v_2 \colon p(v_1, v_2))(v_3, v_4) \vee (v_3 = v_4)$, when $p$ is a binary predicate.

The predicates we use in this paper to record information about a single program configuration include the predicates of Table 2 and additional predicates defined in the following sections. In the sequel, we will use unary predicates such as *thread* to represent type information. This could have been expressed using many-sorted logics, which we have decided to avoid for expository purposes. Instead for convenience we define the shorthands:

$$\exists x \colon type.\varphi = \exists x.type(x) \wedge \varphi$$
$$\forall x \colon type.\varphi = \forall x.type(x) \rightarrow \varphi$$

| **Predicates** | **Intended Meaning** |
|---|---|
| $thread(t)$ | $t$ is a thread |
| $\{at[lab](t) : lab \in Labels\}$ | thread $t$ is at label $lab$ |
| $\{rval[fld](o_1, o_2) : fld \in Fields\}$ | field $fld$ of the object $o_1$ points to the object $o_2$ |
| $heldBy(l, t)$ | the lock $l$ is held by the thread $t$ |
| $blocked(t, l)$ | the thread $t$ is blocked on the lock $l$ |
| $waiting(t, l)$ | the thread $t$ is waiting on the lock $l$ |

**Table 2.** Predicates used to record information about a single program configuration

[1] these operators could be extended to handle allocation and deallocation of a (possibly unbounded) set of individuals.

*Example 1.* The formula $\Box\forall t\colon thread.(at[lw_1](t) \rightarrow \Diamond at[lw_c](t))$ specifies the absence of starvation for worker threads in the running example program (Figure 2).

The formula $\Box\exists t\colon thread.\Diamond at[lw_c](t)$ expresses the fact that always some thread eventually enters the critical section. The formula $\exists t\colon thread.\Diamond at[lw_c](t)$ states that some thread eventually enters the critical section.

*Example 2.* Table 3 shows a number of interesting properties, and their specification in ETL .

Property 1 of Table 3 states that globally, each individual that is allocated during program execution is eventually deallocated. Note that the universal quantifier quantifies over individuals of the configuration in which it is evaluated. The individuals that are allocated in the current configuration are related to some future configuration in which they will be deallocated. The temporal structure of this property could be classified as a Response structure [14], in which allocations have deallocation responses. Response properties are very commonly used in specifications of finite-state systems [7].

Properties 2 and 3 establish a ranking function for linked data structures based on transitive reachability. These properties require that the set of individuals transitively reachable from an index variable traversing the structure will decrease on each iteration of the traversing loop. Note that these properties relate an unbounded number of individuals of one configuration to another. For example, in the case of non-increasing reachability, the non-reachable individuals of the configuration in which the program is at the loop head are related to a future configuration in which the program is at the loop head.

Properties 4 and 5 are again response properties, stating that in a concurrent producer/consumer program in which the producer enqueues item into an unbounded queue and the consumer dequeues items from that queue, every produced item is eventually enqueued and every consumed item is eventually dequeued.

Property 6 is self explanatory, and property 7 is a desired property of a garbage collector — that all non-reachable items are eventually collected.

**Evolution Semantics** First-order modal logic may be given a constant domain semantics in which the domain of all worlds is fixed or a varying domain semantics in which the domains of worlds may vary and are generally not required to intersect. In these semantics, an object may exist in more than a single world, and an equality relation is predefined to express global equality between individuals of the domain of the model [9].

To maintain the notion of equality in the presence of dynamic allocation and deallocation without the need to update a predefined global equality relation, we use an evolution semantics, which is adapted from Lewis's counterpart semantics [13].

In this semantics, an individual cannot exist in more than a single world; each world has its own domain, and domains of different worlds are non-intersecting.

Under this model, equality need only be defined within a single world's boundary. Individuals of different worlds are incomparable by definition. To relate individuals of different worlds, an evolution mapping is defined.

In the following definitions, $head(\pi)$ denotes the first configuration in a trace $\pi$, $tail(\pi)$ denotes the suffix of $\pi$ without the first configuration, and $last(\pi)$ denotes the last configuration of $\pi$. $\pi^i$ denotes the suffix of $\pi$ starting at the $i$-th configuration.

**Definition 4.** *[Evolution mapping] Let $\tau$ be the finite prefix of length $k$ of the trace $\pi$. We say that an individual $u \in U_{head(\tau)}$ evolves into an individual $u' \in U_{last(\tau)}$ in the trace $\pi$ in $k$ steps, and write $\pi \models_k u \rightsquigarrow u'$ when there is a sequence of individuals $u_1, \ldots, u_k$ such that $u_1 = u$ and $u_k = u'$ and for each two successive configurations in $\tau$, $u_{i+1} = e_{\tau_i}(u_i)$.*

**Definition 5.** *[Assignment evolution] Let $\tau$ be the finite prefix of length $k$ of the trace $\pi$. Given a formula $\varphi$ and an assignment $Z$ mapping free variables to individuals of a universe $U_{head(\tau)}$. We say that $\pi \models_k Z \rightsquigarrow Z'$ ($Z$ evolves to $Z'$ in $\pi$ in $k$ steps) if for each free variable $fv_i$ of $\varphi$, $\pi \models_k Z(fv_i) \rightsquigarrow Z'(fv_i)$, $Z(fv_i) \in U_{head(\tau)}$, and $Z'(fv_i) \in U_{last(\tau)}$.*

**Definition 6.** *[ETL evolution semantics]*

*We inductively define when an ETL formula $\varphi$ is satisfied over a trace $\pi$ with an assignment $Z$ (denoted by $\pi, Z \models \varphi$).*

- $\pi, Z \models \mathbf{1}$, *and not* $\pi, Z \models \mathbf{0}$.
- $\pi, Z \models p(x_1, \ldots, x_k)$ *iff* $\iota_{head(\pi)}(p)(Z(x_1), \ldots, Z(x_k)) = 1$
- $\pi, Z \models (x_1 = x_2)$ *iff* $Z(x_1) = Z(x_2)$ *and* $Z(x_1) \in U_{head(\pi)}$
- $\pi, Z \models \neg\varphi$ *iff not* $\pi, Z \models \varphi$
- $\pi, Z \models \varphi \vee \psi$ *iff* $\pi, Z \models \varphi$ *or* $\pi, Z \models \psi$
- $\pi, Z \models \exists x.\varphi$ *iff there exists* $u \in U_{head(\pi)}$
  *s.t.* $\pi, Z[x \mapsto u] \models \varphi(x)$
- $\pi, Z \models (TC\ v_1, v_2 \colon \varphi)(v_3, v_4)$ *iff*

$$\begin{array}{l} \text{there exists } u_1, \ldots, u_{n+1} \in U_{head(\pi)} \\ \text{s.t. } Z(v_3) = u_1, Z(v_4) = u_{n+1}, \\ \text{and for all } 1 \leq i \leq n, \pi, Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi \end{array}$$

*Temporal modalities:*

- $\pi, Z \models \chi\varphi$ *iff* $tail(\pi), Z' \models \varphi$ *where* $\pi \models_1 Z \rightsquigarrow Z'$.
- $\pi, Z \models \varphi\ \mathcal{U}\ \psi$ *iff there exists* $k \geq 1$ *s.t.,* $\pi^k, Z' \models \psi$ *and* $\pi \models_k Z \rightsquigarrow Z'$ *and for all* $1 \leq j < k$, $\pi^j, Z'' \models \varphi$ *and* $\pi \models_j Z \rightsquigarrow Z''$,

*Allocation and deallocation operators:*

- $\pi, Z \models \odot v$ *when* $Z(v) \in A_{head(tail(\pi))}$.
- $\pi, Z \models \oslash v$ *when* $Z(v) \in D_{head(\pi)}$.

*We write $\pi \models \varphi$ when $\pi, Z \models \varphi$ for each assignment $Z$.*

It is worth noting that the first-order quantifiers in this definition range only over the individuals of a single world, yet the overall effect achieved by using the evolution mapping is the ability to reason about individuals of different worlds, and how they relate to each other. In essence, the assignment $Z[x \mapsto u]$ binds $x$ to (the evolution of) an individual from the domain of the world over which the quantifier was evaluated (cf. the semantics of $\chi$ and $\mathcal{U}$ ).

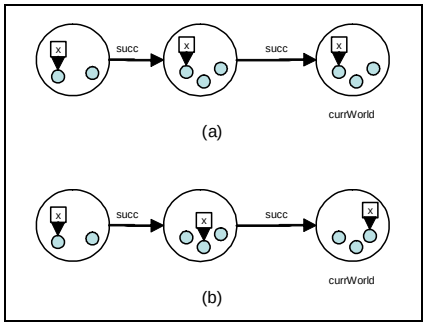


**Fig. 5.** Interaction of first-order quantifiers and temporal operators

The combination of first-order quantifiers and modal operators creates complications that do not occur in propositional temporal logics. In particular, the quantification domain of a quantifier may vary as the universe of underlying configurations varies.

*Example 3.* The formula $\exists x.\Box p(v)$ states that the pointer variable `p` remains constant throughout program execution, and points to an object that existed in the program's initial configuration. On the other hand, the formula $\Box\exists x.p(v)$ simply states that `p` never has the value null, but allows it to point to different objects throughout program execution, and in particular to objects that did not exist in the initial program configuration. An example for such trace prefixes is shown in Figure 5, where in $(a)$ $x$ points to the same object in all configurations and in $(b)$ it points to different objects in different configurations.

**Definition 7.** *[ETL Satisfaction] We say that the program satisfies an ETL formula $\varphi$ when all infinite traces of the program satisfy $\varphi$.*

The evolution semantics allows each world to have a different universe, thus conceptually representing a varying-domain model, which allows dynamic allocation and deallocation of objects and threads. Note that in this semantics, the only relation that crosses world boundaries is the evolution mapping.

In Section 3 we give a possible implementation of this semantics via our first-order logic framework.

| No. | Property | Formula |
|---|---|---|
| 1 | all allocated objects are eventually deallocated | $\Box(\forall v.\, \odot v \rightarrow \Diamond \oslash v)$ |
| 2 | diminishing reachability from index variable `i` at loop head ($l_{lh}$) | $\Box(\exists v, v_0.at[l_{lh}](t_0) \wedge rval[i](t_0, v_0) \wedge rval[next]^*(v_0, v)$ $\wedge \Diamond(at[l_{lh}](t_0) \wedge \neg\exists v_0 \colon rval[i](t_0, v_0) \wedge rval[next]^*(v_0, v)))$ |
| 3 | non increasing reachability from index variable `i` at loop head ($l_{lh}$) | $\Box(\forall v.at[l_{lh}](t_0) \wedge \neg\exists v_0 \colon rval[i](t_0, v_0) \wedge rval[next]^*(v_0, v)$ $\rightarrow \Box\neg at[l_{lh}](t_0) \vee \neg\exists v_0 \colon rval[i](t_0, v_0) \wedge rval[next]^*(v_0, v))$ |
| 4 | every produced item is eventually queued | $\Box(\forall v.at[lp_1](t) \wedge \odot v$ $\rightarrow \Diamond\exists v_0.rval[head](t_0, v_0) \wedge rval[next]^*(v_0, v))$ |
| 5 | every consumed item is eventually dequeued | $\Box(\forall v.at[lt_4](t) \wedge rval[x_d](t, v)$ $\rightarrow \Diamond\neg(\exists v_0.rval[head](t_0, v_0) \wedge rval[next]^*(v_0, v)))$ |
| 6 | absence of starvation in critical section | $\Box(\forall t \colon thread.\Diamond at[l_{crit}](t))$ |
| 7 | non reachable objects eventually collected | $\Box(\forall v.\Diamond\Box\bigwedge_{\substack{x \in Var \\ sl \in Sel}}\neg\exists v_0.rval[x](t_0, v_0) \wedge rval[sel]^*(v_0, v))$ $\rightarrow \Diamond \oslash v)$ |

**Table 3.** Sample ETL Specifications

### 2.2 Separable Specifications

It is interesting to consider subclasses of ETL for which the verification problem is somewhat easier. Two such classes are *spatially separable* and *temporally separable* specifications.

Spatially separable specifications do not place requirements on the relationships between individuals of a historical world. As implied by their name, such specifications are separable on the spatial dimension, i.e., they allow each individual to be considered separately. Therefore, they could be considered as a set of propositional verification problems — one for each individual. This set could be solved by a single analysis in parallel by separately tracking the state of each individual.

**Definition 8.** *[Spatial Separability] An ETL formula $\varphi$ is said to be spatially separable when temporal operators in $\varphi$ are only applied to first-order formulae with the same single free variable.*

In temporally separable specification there is no relation between individuals across worlds. Essentially, this corresponds to the extraction of propositional information from each world, and having temporal specifications over the extracted propositions. This class was addressed in [21].

**Definition 9.** *[Temporal Separability] An ETL formula $\varphi$ is said to be temporally separable when temporal operators in $\varphi$ are only applied to closed first-order formulae.*

## 3 Expressing Trace Semantics using First-Order Logic

In this section we use first-order logic to represent traces, we encode temporal operators using standard first-order quantifiers. This allows us to automatically

derive the abstract semantics of Section 4. It can be shown that this could be used to define different kinds of temporal logic such as the $\mu$-calculus. One limitation of our first-order logical structures, is that they do not explicitly represent infinite traces. However, we will show in Section 4 that this does not affect the soundness of our analysis although it may lead to overly conservative results. Our initial experience indicates that we may show some temporal properties for program with dynamically allocated storage.

### 3.1 Representing Finite Trace Prefixes

We encode a finite prefix of a trace via a first-order logical structure using the set of designated predicates specified in Table 4. Successive worlds are connected using the *succ* predicate. Each world of the trace may contain an arbitrary number of individuals. The predicate $exists(o, w)$ relates an individual $o$ to a world $w$ in which it exists. Each individual only exists in a single world. The $evolution(o_1, o_2)$ predicate relates an individual $o_1$ to its counterpart $o_2$ in a successor world. The predicates $isNew$ and $isFreed$ hold for newly created or deallocated individuals.

**Definition 10.** *[Concrete Trace] A concrete trace parameterized by a set of predicates $P$, each with a fixed arity, is a trace encoded as a first-order logical structure $T = \langle U_T, \iota_T \rangle$, where $U_T$ is the universe of the trace, and $\iota_T$ is the interpretation function mapping predicates to their truth value in the logical structure, that is, $p \in P$ of arity $k$, $\iota_T(p) \colon U_T^k \to \{0, 1\}$.*

*To exclude structures that cannot represent valid traces, we impose certain hygiene condition in the spirit of [17]. For example, we require that each world has at most one successor (predecessor) and that every configuration but the last has exactly one successor.*

*In the sequel, the set of traces is denoted by $CTraces[P]$. We will omit $P$ to mean the set of predicates in Table 4.*

| **Predicate** | **Intended Meaning** |
|---|---|
| $world(w)$ | $w$ is a world |
| $currWorld(w)$ | $w$ is the current world |
| $initialWorld(w)$ | $w$ is the initial world of the trace |
| $succ(w_1, w_2)$ | $w_2$ is the successor of $w_1$ |
| $exists(o, w)$ | object $o$ is in world $w$ |
| $evolution(o_1, o_2)$ | object $o_1$ evolves to $o_2$ |
| $isNew(o)$ | object $o$ is new |
| $isFreed(o)$ | object $o$ is freed |

**Table 4.** Trace predicates

The following definition converts a finite prefix of a trace as defined in Section 2 into a first-order structure.

**Definition 11.** *[Representation Mapping] We define the representation mapping rep of a finite prefix $\tau$ of infinite trace to the the first-order structure that corresponds to $\tau$. That is, every configuration is mapped to a world, with succ predicate holding for successor configurations.*

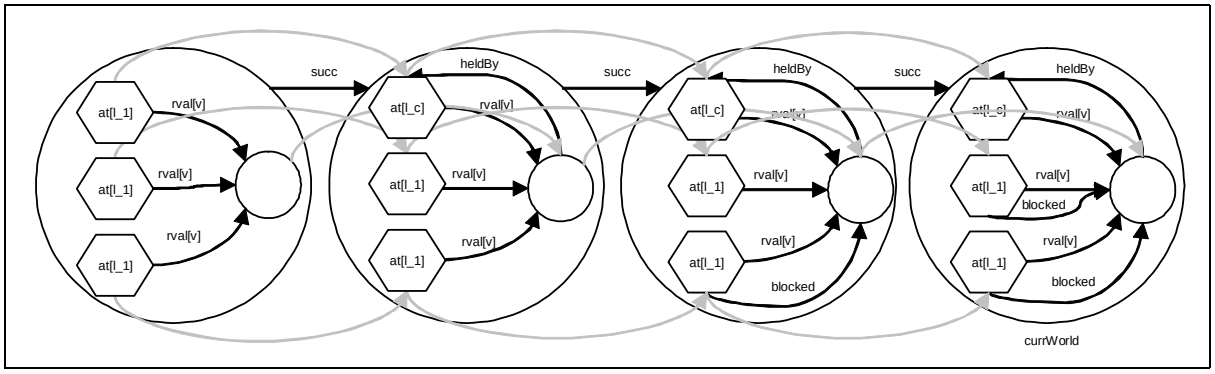


**Fig. 6.** A concrete trace $T_6$

*Example 4.* The trace $T_6{}^{\natural}$ shown in Figure 6 consists of 4 configurations. Grey edges, crossing world boundaries, are evolution edges, relating objects of different worlds. Note that these are the only edges which cross world boundaries.

### 3.2 Exact Extraction of Trace Properties

Once traces are represented via first-order logical structures, trace properties could be extracted by evaluating formulae of first-order logic with transitive closure.

**Definition 12.** *[Translation] We translate a given ETL formula $\varphi$ to an FOTC formula $FO(\varphi)$ by making the underlying trace structure explicit, and translating temporal operators to FOTC claims over worlds of the trace.*

*The translation procedure is straightforward, and given in Section A.*

*Example 5.* The property $\exists t\colon thread.\Diamond at[l_c](t)$ of Example 1 is translated to

$$\begin{aligned}&\exists w : world.\exists t : thread.initialWorld(w)\wedge\\ &\quad exists(t,w)\wedge\exists w'\exists t' : thread.succ^*(w,w')\wedge\\ &\quad exists(t',w')\wedge evolution^*(t,t')\wedge at[l_c](t')\end{aligned}$$

which evaluates to 1 for the trace prefix of Figure 6.

The property $\Box\forall t : thread.\Diamond at[l_c](t)$ could be translated to the following formula using the predicates of Table 4:

$$\begin{aligned}&\forall w : world.\forall t : thread.exists(t, w) \rightarrow \\ &\quad \exists w' : world.\exists t' : thread.succ^*(w, w') \\ &\quad \wedge exists(t', w') \wedge evolution^*(t, t') \wedge at[l_c](t')\end{aligned}$$

**Definition 13.** *The* **meaning** *of a formula $\varphi$, denoted by $[\![\varphi]\!]^\pi(Z)$, yields a truth value in $\{0, 1\}$. The meaning of $\varphi$ is defined inductively as follows:*

**Atomic Formulae** *For an atomic formula consisting of a logical literal $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}\}$, $[\![\boldsymbol{l}]\!]^\pi(Z) = l$ (where $l \in \{0, 1\}$).*
*For an atomic formula of the form $p(v_1, \ldots, v_k)$,*

$$[\![p(v_1, \ldots, v_k)]\!]^\pi(Z) = \iota^\pi(p)(Z(v_1), \ldots, Z(v_k))$$

*For an atomic formula of the form $(v_1 = v_2)$,*

$$[\![v_1 = v_2]\!]^\pi(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & Z(v_1) = Z(v_2) \end{cases}$$

**Logical Connectives** *When $\varphi$ is built from subformulae $\varphi_1$ and $\varphi_2$,*

$$\begin{aligned}[\![\varphi_1 \vee \varphi_2]\!]^\pi(Z) &= max([\![\varphi_1]\!]^\pi(Z), [\![\varphi_2]\!]^\pi(Z)) \\ [\![\neg\varphi_1]\!]^\pi(Z) &= 1 - [\![\varphi_1]\!]^\pi(Z)\end{aligned}$$

**Quantifiers** *When $\varphi$ has a quantifier as the outermost operator,*

$$[\![\exists v_1 : \varphi_1]\!]^\pi(Z) = max_{u \in U^\pi}[\![\varphi_1]\!]^\pi(Z[v_1 \mapsto u])$$

**Transitive Closure** *When $\varphi$ is of the form $(TC\ v_1 \colon v_2)(\varphi_1)v_3v_4$,*

$$\begin{aligned}&[\![(TC\ v_1 \colon v_2)(\varphi_1)v_3v_4]\!]^\pi(Z) = \\ &\quad max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \ min_{i=1}^n [\![\varphi_1]\!]^\pi(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])\end{aligned}$$

*We say that $\pi$ and $Z$* **satisfy** *$\varphi$ (denoted by $\pi, Z \models \varphi$) if $[\![\varphi]\!]^\pi(Z) = 1$. We write $\pi \models \varphi$ if for every $Z$ we have $\pi, Z \models \varphi$.*

### 3.3 Action Semantics

Informally, a program action $ac$ consists of a *precondition* $pre(ac)$ under which the action is enabled, which is expressed as logical formula, and a set of formulae updating the values of predicates according to the effect of the action. Enabled actions extend the trace with a newly created configuration where the interpretations of every predicate $p$ of arity $k$ is determined by evaluating a formula $\varphi_p(v_1, v_2, \ldots, v_k)$ which may use $v_1, v_2, \ldots, v_k$ and all predicates in $P$. (refer to [17]).

# 4 Exploring Finite Abstract Traces via Abstract Interpretation

In this section we give an algorithm for conservatively determining the validity of a program with respect to ETL temporal properties. Our initial experience, as reported in Section 5, is that this algorithm can be used to automatically verify temporal properties of heap manipulating programs.

There are four main ingredients to our approach: (i) the expressive power of ETL , which allows us to show that we can safely consider only finite prefixes; (ii) the need to delay the evaluation of temporal properties until the traces can no longer be extended, to make the result of the analysis useful; (iii) the finite representation of potentially unbounded finite prefixes of traces using 3-valued logical structures with the values $\{0, 1, 1/2\}$, where $1/2$ represents "unknown" value due to abstraction. (iv) The recording of intermediate values of temporal properties using special instrumentation predicates to avoid resulting $1/2$ values for many liveness properties due to abstraction. These instrumentation predicates accumulate temporal properties on-the-fly while the abstract trace is extended

The rest of this section is organized as follows: In Section 4.1 we show how to finitely represent finite trace prefixes via 3-valued logical structures. In Section 4.2, we describe our abstract interpretation algorithm. Section 4.3 then presents trace-instrumentation predicates needed to refine the precision of our analysis, and defines the crucial notion of transworld-equality via a designated instrumentation predicate. In addition, Section 4.3 shows how to derive some of the specifically required instrumentation predicates for a given ETL formula.

## 4.1 Finitely Representing Trace Prefixes

The following definition imposes an order on truth values of the 3-valued logic as in [17].

**Definition 14.** *[Information Order] For $l_1, l_2 \in \{0, 1, 1/2\}$, we define the information order on truth values as follows: $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = 1/2$.*

**Definition 15.** *[Representing traces via 3-valued logic] An abstract trace is a 3-valued first-order logical structure $T = \langle U_T, \iota_T \rangle$, where $U_T$ is the universe of the abstract trace, and $\iota_T$ is the interpretation mapping predicates to their truth values, that is, $p \in P$ of arity $k$, $\iota_T(p) \colon U_T^k \to \{0, 1, 1/2\}$.*

*We also use a designated predicate sm to express whether an individual may represent more than a single node. We assume that sm is 0 for concrete structures.*

*We denote by $[\![\varphi]\!]_3^T(Z)$ the meaning of $\varphi$ in a 3-valued logic [17]. Also, we say that a trace $T$ with an assignment $Z$* potentially satisfies *a formula $\varphi$ when $[\![\varphi]\!]_3^T(Z) \in \{1, 1/2\}$ and denote this by $T, Z \models_3 \varphi$.*

**Embedding into Bounded Traces**

**Definition 16.** *[Embedding Ordering of Abstract Traces]*

*Let $T = \langle U, \iota \rangle$ and $T' = \langle U', \iota' \rangle$ be abstract traces encoded as first-order structures. A function $f : T \to T'$ such that $f$ is surjective is said to* embed $T$ into $T'$ *if for each predicate $p$ of arity $k$, and for each $u_1, \ldots, u_k \in U$:*

$$\iota(p(u_1, u_2, \ldots, u_k)) \sqsubseteq \iota'(p(f(u_1), f(u_2), \ldots, f(u_k))) \tag{1}$$

*and*

$$|\{u | f(u) = u'\}| > 1 \sqsubseteq \iota'(sm(u')) \tag{2}$$

*This sets sm to a non-zero value for each abstract individual that may represent more than a single concrete individual.*

*We say that $T'$* represents *$T$ when there exists such an embedding $f$.*

One way of creating an embedding function $f$ is by using *canonical abstraction*. Canonical 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.

A *tight embedding* is a special kind of embedding—one in which information loss is minimized when multiple individuals of $T$ are mapped to the same individual in $T'$:

**Definition 17.** *A structure $T' = \langle U^{T'}, \iota^{T'} \rangle$ is a* **tight embedding** *of $T = \langle U^T, \iota^T \rangle$ if there exists a surjective function $t\_embed : U^T \to U^{T'}$ such that, for every predicate $p$ of arity $k$,*

$$\iota^{T'}(p)(u'_1, \ldots, u'_k) = \bigsqcup_{t\_embed(u_i) = u'_i, 1 \le i \le k} \iota^T(p)(u_1, \ldots, u_k) \tag{3}$$

*and for every $u' \in U^{S'}$,*

$$\iota^{T'}(sm)(u') = (|\{u | t\_embed(u) = u'\}| > 1) \sqcup \bigsqcup_{t\_embed(u) = u'} \iota^T(sm)(u) \tag{4}$$

*When a surjective function t_embed possesses both properties (3) and (4), we say that $T' = t\_embed(T)$.*

*Example 6.* Figure 8 shows an abstract trace with 3 abstract configurations. A node with double-line boundaries is a summary node representing possibly more than a single concrete node. Similarly, the world with a double-line boundaries, is a summary world representing possibly more than a single world. Dashed edges are 1/2 edges representing values that may or may not be true. For example, a 1/2 successor edge between two configurations represents the possible succession of configurations.

### 4.2 Abstract Interpretation

Our abstract semantics represents abstract states using 3-valued structures. When a trace is extended, we evaluate formula's precondition and its update formulae using 3-valued logic. The 3-valued interpretation is a reinterpretation of Definition 13 over a 3-valued domain.

We extend mappings on individuals to operate on assignments: If $f\colon U^T \to U^{T'}$ is a function and $Z\colon Var \to U^T$ is an assignment, $f \circ Z$ denotes the assignment $f \circ Z\colon Var \to U^{T'}$ such that $(f \circ Z)(v) = f(Z(v))$.

The following theorem is one of the components in showing the soundness of the analysis.

**Theorem 1.** *[Embedding Theorem, [17]]*
*Let $T = \langle U^T, \iota^T \rangle$ and $T' = \langle U^{T'}, \iota^{T'} \rangle$ be two concrete traces encoded as first-order structures, and let $f\colon U^T \to U^{T'}$ be a function such that $T \sqsubseteq^f T'$. Then, for every formula $\varphi$ and complete assignment $Z$ for $\varphi$, $[\![\varphi]\!]^T(Z) \sqsubseteq [\![\varphi]\!]^{T'}(f \circ Z)$.*

It often happens that the temporal properties evaluate to 1/2, due to an overly conservative approximation. In the next section we present a machinery for refining the abstraction to allow successful verification in interesting cases.

*Example 7.* Space precludes us from showing a real application, as in the web-server. Instead, we use an artificial example which is also used in the next section. Figure 7 shows an abstract trace in which the property $P(v)\ \mathcal{U}\ Q(v)$ holds for the single individual but the formula $P(v)\ \mathcal{U}\ Q(v)$ evaluates to 1/2 since the successor (and evolution edge have 1/2 value).

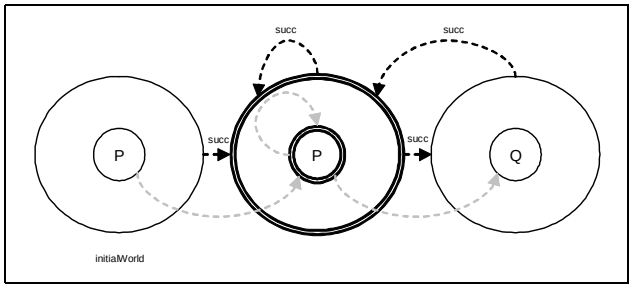


**Fig. 7.** An abstract trace $T_7$ with a single individual for which $P(v)\ \mathcal{U}\ Q(v)$ holds, but evaluates to 1/2

**Predicting Temporal Behavior using Abstraction** One of the most difficult issues in proving liveness properties, since a liveness property is sometimes only satisfied by an infinite trace. Still, the following theorem guarantees that

even though our abstract interpretation only explores the representation of finite prefixes, the solution that we obtain is sound.

**Theorem 2.** *For every ETL formula $\varphi$, every (infinite) trace $\pi$, and every assignment $Z\colon FV(\varphi) \to U_\pi$, there exists a finite prefix of $\pi$, $\tau$ such that:*

$$\pi, Z \models \varphi \iff g(\tau), t\_embed \circ Z \models_3 FO(\varphi) \tag{5}$$

*where $g = t\_embed \circ rep$.*

*Proof. appears in the full paper, by structural induction on the ETL formula.*

Unfortunately the above theorem does not lead to a useful procedure for verifying liveness properties, since these properties may not turn out to hold on short prefixes. Therefore, an oblivious analysis that considers all finite prefixes can only show correctness of safety properties. Our chaotic iteration algorithm ignores traces which are prefixes of other traces.

### 4.3 Property Guided Instrumentation

To refine the abstract interpretation throughout the analysis, we maintain more precise information about correctness of temporal formulae as traces are being constructed. Generally, this principle is referred to in [17] as the *Instrumentation Principle*. This work goes beyond what was mentioned there, by showing how one could actually obtain instrumentation predicates from the temporal specification. In Section 5, we report the result of verifying temporal properties when using this procedure.

The rest of this subsection is organized as follows. In Section 4.3 we introduce special kind of instrumentation predicates that are crucial in order to handle evolution of heap allocated objects. Then, in Section 4.3 we introduce predicates which are derived from the temporal specification.

**Trace Instrumentation** The predicates in Table 5 are required for preserving properties of interest under abstraction. For example, the instrumentation predicate $current(o)$ denotes that $o$ is a member of the current world, and should be distinguished from individuals of predecessor worlds.

| **Predicate** | **Intended Meaning** | **Formula** |
|---|---|---|
| $twe(o_1, o_2)$ | object $o_1$ is equal to object $o_2$ possibly across worlds | $(o_1 == o_2)$ $\vee evolution^*(o_1, o_2)$ $\vee evolution^*(o_2, o_1)$ |
| $current(o)$ | object $o$ is a member of the current world | $\exists w\colon world(o, w)$ $\wedge currWorld(w)$ |

**Table 5.** Trace instrumentation predicates

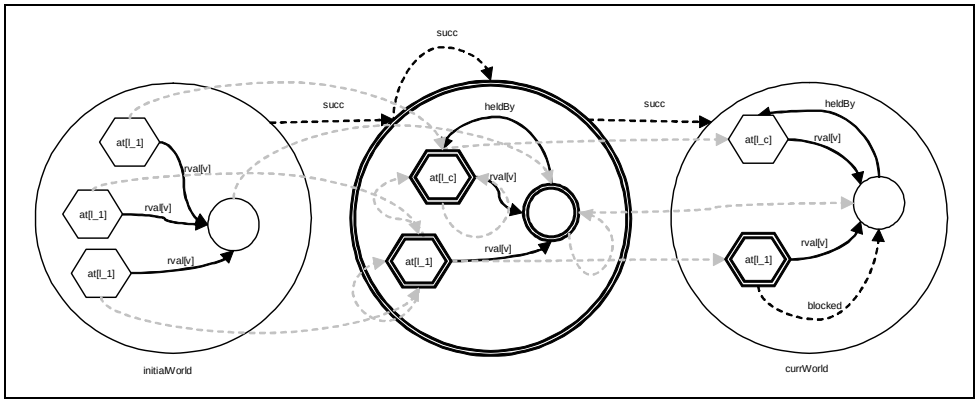


**Fig. 8.** An abstract trace $T_6$ representing the concrete trace $T_6{}^{\natural}$

*Transworld Equality*: In the evolution semantics, two individuals are considered different incarnations of the same individual when one may be transitively evolved to the other. We refer to this notion of equality as transworld equality and introduce an instrumentation predicate $twe(v_1, v_2)$ to capture this notion.

Since our abstraction operates on traces (and not only single configurations), individuals of different worlds may be abstracted together. Transworld equality is crucial for distinguishing a summary node that represents different incarnations of the same individual from a summary node that may represent a number of different individuals.

An example of transworld equality is given in Figure 9 in which the 1-valued *twe* self-loop to the summary thread-node at label $l_c$ records the fact that this summary node actually represents multiple incarnations of a single thread and not a number of different threads.

**Temporal Instrumentation** Given an ETL specification formula, we construct a corresponding set of instrumentation predicates for refining the abstraction of the trace according to the property of interest. The set of instrumentation predicates essentially corresponds to the sub-formulae of the original specification. In this paper, we only use sub-formulae that correspond to temporal operators, and evaluate the spatial ones as usual.

*Example 8.* Recall Example 7 in which the property $\exists v\colon P(v)\ \mathcal{U}\ Q(v)$ evaluated to 1/2 although being satisfied by the trace. We now add the temporal instrumentation predicates $I_p(v)$ and $I_q(v)$ to record the value of the temporal subformulae. The predicates are updated according to their value in the previous configuration. Note the use of transworld equality instrumentation to more precisely record transitive evolution of objects. In particular, this provides the information that the summary node of the second configuration is an abstraction of different incarnations of the same single object.

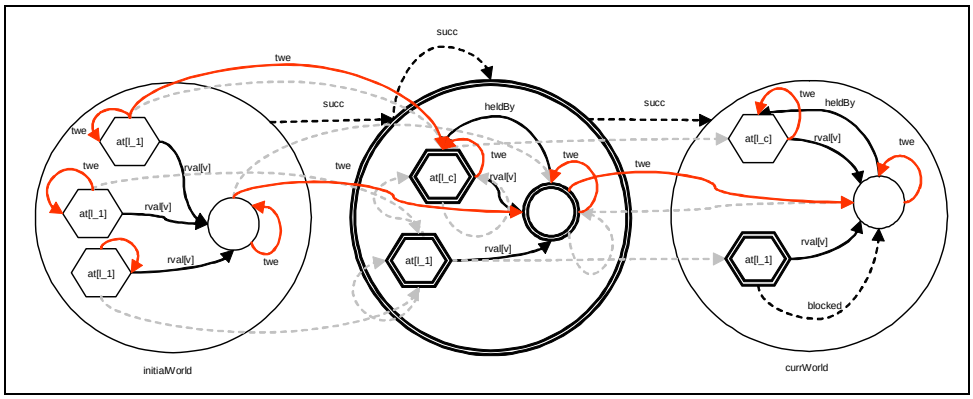


**Fig. 9.** Abstract trace with transworld equality instrumentation. Only 1-valued transworld equality edges are shown

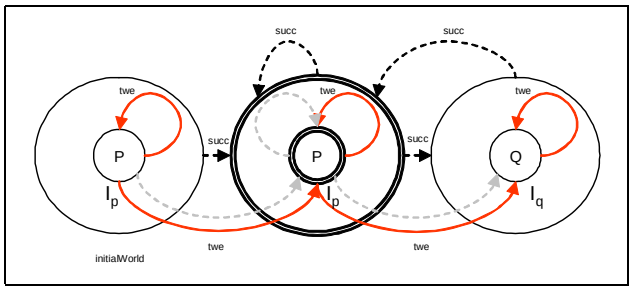


**Fig. 10.** An instrumented abstract trace $T_{10}$ with a single individual for which $P(v)\ \mathcal{U}\ Q(v)$ holds

## 5 Prototype Implementation

We have implemented a prototype of our framework based on the TVLA framework [12]. We have used the prototype implementation to verify the termination of sequential programs manipulating singly-linked lists and of concurrent heap manipulating programs, results are given in Table 6. Running times were measured using Sun's JVM1.3 for Windows 2000, running on a 900MHZ Pentium III. The prototype implementation used is a naive implementation, using TVLA, without applying any additional optimizations.

### 5.1 Analysis Cost

The cost of verifying general properties using this framework could be very high. This should not (and probably does not) come as a surprise considering the setting in which the framework operates. This setting is extremely challenging, it includes the most notorious features of a verification environment: concurrency, dynamic allocation of objects, dynamic allocation of threads, object references, and references to threads. As far as we know, there are no other analyses supporting this combination of features with the precision provided by our framework. In particular, many existing analyses assume a simplified program model with no support for at least one of dynamic allocation of objects, dynamic allocation of threads, references to object, and references to threads [16].

A nice feature of the framework is its adaptivity, the cost of verification reduces when programs verified do not use the complicated features (e.g., parametric systems which use no dynamic allocation), and when the verified properties are simple.

This framework sheds some light on the complications arising in verification with a varying domains model. The amount of information that should be recorded for verifying general properties of such programs is extremely high. Nevertheless, instances of this framework could be applied to subclasses of the general specifications. In particular, Section 2.2 identifies two specification classes for which more efficient verification could be applied.

## 6 Conclusion

We have presented a parametric framework for verifying temporal properties of concurrent Java-like programs. The framework uses a first-order temporal logic which is implemented using 3-valued first-order logic by exploring finite trace prefixes.

## A Translation of ETL to FOTC

We say that a ETL sub-formula is temporally-bound if it appears under a temporal operator. Translations for temporally-bound and non-temporally-bound formulae are different, since non-temporally-bound formulae should be bound to the initial world of the trace.

```
public void put(int value) {
     QueueItem x_i = new QueueItem(value);                    lp_0
     synchronize(this) {                                      lp_1
         if (tail == null) {                                  lp_2
            tail = x_i;                                       lp_3
            head = x_i;                                       lp_4
         } else {
            tail.next = x_i;                                  lp_5
            tail = x_i;                                       lp_6
         }
     }                                                        lp_7
}                                                             lp_8
public QueueItem take() {
     synchronized(this) {                                     lt_0
         QueueItem x_d;
         if (head != null) {                                  lt_1
            newHead = head.next;                              lt_2
            x_d = head;                                       lt_3
            x_d.next = null;                                  lt_4
            head = newHead;                                   lt_5
            if (newHead == null) {                            lt_6
               tail = null;                                   lt_7
            }
         }
     }                                                        lt_8
     return x_d;                                              lt_9
}
```

**Fig. 11.** Queue implementation used by producer / consumer. Note that the queue is unbounded

| Program | Property | Time | Configurations |
|---|---|---|---|
| tr_search | termination | 134 | 144 |
| tr_insert | termination | 2215 | 363 |
| tr_delete | termination | 2658 | 406 |
| tr_delall | termination | 268 | 165 |
| tr_swap | termination | 96 | 82 |
| tr_reverse | termination | 500 | 189 |
| tr_getlast | termination | 288 | 113 |
| one-shot mutual exclusion | absence of starvation | 472 | 1918 |
| web server | P2 | 3864 | 4048 |
| web server | P3 | 10472 | 9015 |
| web server | P4 | 7307 | 5411 |
| web server | P6 | 10431 | 8892 |

**Table 6.** The programs and properties verified with running times and number of configurations. Programs below the two horizontal line are concurrent programs.

**Definition 18.** *[ETL translation to FOTC] We denote by* $\langle\varphi\rangle^w$ *the bounded translation of a formula* $\varphi$ *in a world* $w$ *and by* $\langle\psi\rangle$ *the non-bounded translation. This is referred to as FO translation is Section 3.*

- $\langle\psi\rangle = \exists w\colon world.initialWorld(w) \wedge \langle\psi\rangle^w$
- *if* $\varphi$ *is an atomic formula other than* $\odot x$ *and* $\oslash x$ *then* $\langle\varphi\rangle^w = \langle\varphi\rangle = \varphi$. *If* $\varphi = \odot x$ *then* $\langle\varphi\rangle^w = \langle\varphi\rangle = isNew(x)$. *If* $\varphi = \oslash x$ *then* $\langle\varphi\rangle^w = \langle\varphi\rangle = isFreed(x)$.
- $\langle\varphi \wedge \psi\rangle^w = \langle\varphi\rangle^w \wedge \langle\psi\rangle^w$, $\langle\varphi \vee \psi\rangle^w = \langle\varphi\rangle^w \vee \langle\psi\rangle^w$, $\langle\neg\varphi\rangle^w = \neg\langle\varphi\rangle^w$
- $\langle\varphi(x_1, \ldots, x_n)\ \mathcal{U}\ \psi(y_1, \ldots, y_k)\rangle^w =$
  $\exists w'\colon world.\exists y'_1, \ldots, y'_k\colon succ^*(w, w') \wedge \langle\psi(y'_1, \ldots, y'_k)\rangle^{w'}$
  $\bigwedge_{1\le i\le k} evolution^*(y_i, y'_i)$
  $\wedge\forall\tilde{w}\colon world.\exists x'_1, \ldots, x'_n\colon (succ^*(w, \tilde{w})$
  $\wedge\ succ^*(\tilde{w}, w') \rightarrow \langle\varphi(x'_1, \ldots, x'_n)\rangle^{\tilde{w}}$
  $\bigwedge_{1\le j\le n} evolution^*(x_j, x'_j))$
- $\langle\chi\varphi(x_1, \ldots, x_n)\rangle^w = \exists w'\colon world.\exists x'_1, \ldots, x'_n\colon succ(w, w')$
  $\wedge\ \langle\varphi(x'_1, \ldots, x'_n)\rangle^{w'} \bigwedge_{1\le j\le n} evolution^*(x_j, x'_j)$
  $\wedge\ exists(x'_j, w')$
- $\langle\exists x\ \varphi\rangle^w = \exists x\colon exists(w, x) \wedge \langle\varphi\rangle^w$

Simplified translations may be used for the $\Diamond$ and $\Box$ temporal operators as shown by the following example.

## References


1. T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. *Lecture Notes in Computer Science*, 2102, 2001.
2. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

3. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN*, pages 205–223, 2000.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Symp. on Princ. of Prog. Lang.*, pages 238–252, New York, NY, 1977. ACM Press.
5. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. of 27th POPL*, pages 12–25, Jan. 2000.
6. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS'00, Static Analysis Symposium*. Springer, 2000. Available at "http://www.math.tau.ac.il/∼ nurr".
7. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 411–421, New York, May 1999. Association for Computing Machinery.
8. B. Eckel. *Thinking in Java*. Prentice-Hall, 2000.
9. M. Fitting and R. Mendelsohn. *First-Order Modal Logic*, volume 277 of *Synthese Library*. Kluwer Academic Publishers, Dordrecht, 1998.
10. G.E. Hughes and M.J. Creswel. *An Introduction to Modal Logic*. Methuen, London, 1982.
11. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, Massachusetts, 1997.
12. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer, 2000. Available at http://www.math.tau.ac.il/∼tla.
13. D. Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, LXV(5):113–126, 1968.
14. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995.
15. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In Proceedings of CAV 2002. To appear.
16. M. Rinard. Analysis of multithreaded programs. *Lecture Notes in Computer Science*, 2126:1–??, 2001.
17. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999.
18. M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 Nov. 1994.
19. P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80, Berlin, June 1990. Springer.
20. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. of 27th POPL*, pages 27–40, Mar. 2001. Available at http://www.cs.tau.ac.il/∼yahave/popl01.ps.
21. E. Yahav, T. Reps, and M. Sagiv. LTL model checking for systems with unbounded number of dynamically created threads and objects. Technical Report TR-1424, Computer Sciences Department, University of Wisconsin, Madison, WI, March 2001.