# Verifying Temporal Heap Properties Specified via Evolution Logic*


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

[1] School of Comp. Sci., Tel-Aviv Univ., Tel-Aviv, Israel, {yahave,msagiv}@post.tau.ac.il
[2] Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI 53706, USA, reps@cs.wisc.edu
[3] Informatik, Univ. des Saarlandes, 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. Unlike classical model checking, which uses propositional temporal logic, we use first-order temporal logic to specify temporal properties of heap evolutions; this logic allows domain changes to be expressed, which permits allocation and deallocation to be modelled naturally. 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 each request made to a web server, can be a quite difficult task.

The contributions of this paper can be summarized as follows:

1. We introduce a first-order modal (temporal) logic [9, 8] that allows specifications of temporal properties of programs with dynamically evolving heaps to be stated in a natural manner.
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 [11] and applied it to verify several temporal properties, including liveness properties of Java programs with evolving heaps.

We have used the framework to specify and verify the following:

*Specify 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.

*Verify 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].


* This research was supported by a grant from the Ministry of Science, Israel, a grant from the Academy of Science Israel, by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme, by ONR under contract N00014-01-1-0796, and by the A. von Humboldt Foundation.

*Verify 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 properties [13]. We have applied this analysis to programs with an unbounded number of threads.

Due to space limitations, the prototype implementation is only discussed in [17, 20].

The remainder of this paper is organized as follows: Section 2 gives an overview of the verification method and contrasts it with previous work. Section 3 introduces trace semantics based on first-order modal logic, and discusses how to state trace properties using the language of evolution logic. Section 4 defines an implementation of trace semantics via first-order logic. Section 5 shows how abstract traces are used to conservatively represent sets of concrete traces. Section 6 summarizes related work. Finally, Section 7 concludes the paper.

## 2 Overview

### 2.1 A Temporal Logic Supporting Evolution

The specification language, *Evolution Temporal Logic* (ETL), is a first-order linear temporal logic that allows specifying properties of the way program execution causes dynamically allocated memory ("the heap") to evolve.

It is natural to consider the concrete semantics of a program as the set of its execution traces [5, 16], where each trace is an infinite sequence of *worlds*. First-order logical structures provide a natural representation of worlds with an unbounded number of objects: an individual of the structure's domain (universe) corresponds to an anonymous, unique store location, and predicates represent properties of store locations. Such a representation allows properties of the heap contents to be maintained while abstracting away any information about the actual physical locations in the store.

This gives rise to traces in which worlds along the trace may have different domains. Such traces can be seen as models of a first-order modal logic with a varying-domain semantics [8]. This could be equivalently, but less naturally, modelled using constant-domain semantics.

This framework generalizes other specification methods that address dynamic allocation and deallocation of objects and threads. In particular, its descriptive power goes beyond Propositional LTL and finite-state machines (e.g., [1]).

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 [12] to cast modal models (and formula evaluation) in terms of classical predicate logic with transitive closure ($FO^{TC}$) [3].

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. Two abstractions are employed: (i) representing multiple concrete worlds by a single abstract world, and (ii) creating cycles when an abstract world reoccurs in the trace.

Because of these abstractions, we may fail to show the correctness of certain programs, even though they are correct. Fortunately, we can use reduction arguments and progress monitors as employed in other program-verification techniques (e.g., [10]).

As in finite-state model checking (e.g., [16]), 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 [15, 18], which only handle safety properties.

### 2.2 Overview of the Verification Procedure

First, the property $\varphi$ is specified in ETL. The formula is then translated in a straightforward manner into an $FO^{TC}$ logical formula, $(\varphi)^{\dagger}$, using a translation procedure described in Appendix A. An abstract-interpretation procedure is then applied to explore finite representations of the set of traces, using Kleene's 3-valued logic to conservatively interpret formulae. The abstract-interpretation procedure essentially computes a greatest fixed point over the set of traces, starting with an abstract trace that represents all possible infinite traces from an initial state, and gradually increasing the set of abstract traces and reducing the set of represented concrete traces. Finally, the formula $(\varphi)^{\dagger}$ is evaluated on all of the abstract traces in the fixed point. If $(\varphi)^{\dagger}$ is satisfied in all of them, then the original ETL formula $\varphi$ must be satisfied by all (infinite) traces of the program. However, it may be the case that for some programs that satisfy the ETL specification, our analysis only yields "maybe".

### 2.3 Running Example

Consider a web server in which a new thread is dynamically allocated to handle each `http` request received. Each thread handles a single request, then terminates and is subject to garbage collection. Assume that worker threads compete for some exclusively shared resource, such as exclusive access to a data file. Figure 1 shows fragments of a Java program that implements such a naive web server.

```
public class Worker implements Runnable {
 Request request;
 Resource resource; ...
 public void run() { ...
  synchronized(resource) {                lw1
   resource.processRequest(request);      lwc
  }                                       lw2
 }}
```

**Fig. 1.** Java fragment for worker thread in a web server with no explicit scheduling.

A number of properties for the naive web-server implementation are shown in Tab. 1 as properties P1–P4. For now you may ignore the formulae in the third column; these will become clear as ETL syntax is introduced in Sec. 3.

Due to the unbounded arrival of requests to the web server, and the fact that a thread is dynamically created for each request, absence of starvation (P2) does not hold in the naive implementation. To guarantee absence of starvation, 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 (for additional properties of interest see [17]). Figure 2 shows fragments of a web-server program in which threads use an explicit FIFO scheduler.

The ability of our framework to model explicit scheduling queues provides a mechanism for addressing issues of fairness in the presence of dynamic allocation of threads. (Further discussion of fairness is beyond the scope of this paper).

```
public class Scheduler
        implements Runnable {
 protected Queue schedQ;
 protected Resource resource; ...
 public void run() {
  while(true) { ...                   ls1
   synchronized(resource) {           ls2
    while(resource.isAcquired())
     resource.wait();                 ls3
     // may block until
     // queue not empty
    worker=schedQ.dequeue();          ls4
    worker.start();                   ls5
   }
  }
 }
}
```

```
public class Listener
        implements Runnable {
 protected Queue schedQ; ...
 public void run() {
  while(true) { ...                    la1
   req=rqStream.readObject();          la2
   worker=
    new Thread(new Worker(req));       la3
   schedQ.enqueue(worker);             la4
  ... }
 }}
public class Worker
        implements Runnable {
 Request req;
 Resource resource; ...
 public void run() {
  synchronized(resource) { ...         lw1
   resource.processRequest(req);       lwc
   resource.notifyAll();
  }                                    lw2
 }}
```

**Fig. 2.** Java code fragment for a web server with an explicit scheduler.

| Pr. | Description | Formula |
|---|---|---|
| P1 | mutual exclusion over the shared resource | $\Box\forall t_1, t_2\colon thread.(t_1 \neq t_2) \rightarrow \neg(at[lw_c](t_1) \wedge at[lw_c](t_2))$ |
| P2 | absence of starvation for worker threads | $\Box\forall t\colon thread.at[lw_1](t) \rightarrow \Diamond at[lw_c](t)$ |
| P3 | a thread only created when a request is received | $\Box(\forall t\colon thread.\neg \odot t) \vee (\forall t\colon thread.\neg \odot t)\ \mathcal{U}\ (\exists v\colon request. \odot v)$ |
| P4 | each request is followed by thread creation | $\Box\exists v\colon request. \odot v \rightarrow \Diamond\exists t\colon thread. \odot t$ |
| P5 | mutual exclusion of listener and scheduler over scheduling queue | $\Box\forall t_1, t_2\colon thread.(t_1 \neq t_2) \rightarrow \neg(at[ls_2](t_1) \wedge at[la_3](t_2))$ |
| P6 | each created thread is eventually inserted into the scheduling queue | $\Box\forall t\colon thread. \odot t \rightarrow \Diamond\exists q\colon queue.rval[head.next^*](q,t)$ |
| P7 | each scheduled worker thread was removed from the scheduling queue | $\Box\forall t\colon thread.at[lw_1](t) \rightarrow \neg\exists q\colon queue.rval[head.next^*](q,t)$ |
| P8 | each worker thread waiting in the queue eventually leaves the queue | $\exists q\colon queue.\Box\forall t\colon thread. (rval[head.next^*](q,t)) \rightarrow \Diamond\neg(rval[head.next^*](q,t))$ |

**Table 1.** Web server ETL specification using predicates of Tab. 2.

## 3 Trace-Based Evolution Semantics

In this section, we define a trace-based semantic domain for programs that manipulate unbounded amounts of dynamically allocated storage. To allow specifying temporal properties of such programs, we employ first-order modal logic [8]. Various such logics have been defined, and in general they can 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 can vary and domains of different worlds can overlap. In the most general setting, in both types of semantics an object can exist in more than a single world, and an equality relation is predefined to express global equality between individuals.

To model the semantics of languages such as Java, and to hide the implementation details of dynamic memory allocation, we use a semantics with varying domains. However, the semantics is deliberately restricted because of our intended application to program analysis. By design, our *evolution semantics* has a notion of equality in the presence of dynamic allocation and deallocation, without the need to update a predefined global-equality relation. Evolution semantics is adapted from Lewis's counterpart semantics [12]. In both evolution and counterpoint 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 unequal by definition.To relate individuals of different worlds, an evolution mapping is defined; however, unlike Lewis, we are interested in an evolution mapping that is reflexive, transitive, and symmetric, which models the fact that, during a computation, an allocated memory cell does not change its identity until deallocated. In Sec. 5.3, we show how to track statically, in the presence of abstraction, the equivalence relation induced by the evolution mapping.

As is often done, we add a skip action from the exit of the program to itself, so that all terminating traces are embedded in infinite traces. The semantics of the program is its set of infinite traces.

In the rest of this paper, we work with a fixed set of predicates (or vocabulary) $\mathcal{P} = \{eq, p_1, \ldots, p_k\}$. We denote by $\mathcal{P}^k$ the set of predicates from $\mathcal{P}$ with arity $k$.

**Definition 1.** *A **world** (program configuration) is represented via a first-order logical structure $W = \langle U_w, \iota_w \rangle$, where $U_w$ is the domain (universe) of the structure, and $\iota_w$ is the interpretation function mapping predicates to their truth values; that is, for each $p \in \mathcal{P}^k$, $\iota_w(p)\colon U_w^k \to \{0,1\}$, such that for all $u \in U_w$, $\iota_w(eq)(u,u) = 1$, and for all $u_1, u_2 \in U_w$ such that $u_1$ and $u_2$ are distinct individuals $\iota_w(eq)(u_1, u_2) = 0$.*

**Definition 2.** *A **trace** is an infinite sequence of worlds $\pi_1 \xrightarrow{D_{\pi_1}, e_{\pi_1}, A_{\pi_2}} \pi_2 \xrightarrow{D_{\pi_2}, e_{\pi_2}, A_{\pi_3}}$ $\ldots$, where: (i) each world represents a global state of the program, $\pi_1$ is an initial state, and for each $\pi_i$, its successor world $\pi_{i+1}$ 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 worlds $\pi_i, \pi_{i+1}$ is related by a stepwise **evolution function**, a bijective renaming function $e_{\pi_i}\colon U_{\pi_i} \setminus D_{\pi_i} \to U_{\pi_{i+1}} \setminus A_{\pi_{i+1}}$.*

**Extracting Trace Properties**

To extract trace properties, we need a language that can relate information from different worlds in a 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] An* **ETL formula** *is defined by*

$$\begin{aligned}\varphi ::= {} & 0 | 1 | p(v_1, \ldots, v_n) | \odot v_1 | \oslash v_1 | \varphi_1 \vee \varphi_2 | \neg\varphi_1 | \exists v_1 . \varphi_1 | (TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4) \\ & | \varphi_1\ \mathcal{U}\ \varphi_2 | \chi\varphi_1\end{aligned}$$

*where $v_i$ are logical variables.*

*The set of free variables in a formula $\varphi$ denoted by $FV(\varphi)$ is defined as usual. In a transitive closure formula, $FV((TC\ v_1, v_2 \colon \varphi_1)(v_3, v_4)) = (FV(\varphi_1) \setminus \{v_1, v_2\}) \cup \{v_3, v_4\}$.*

The operators $\odot$ and $\oslash$ allow the specification to refer to the exact moments of birth and death (respectively) of an individual.[4]

*Shorthand Formulae*: For convenience, we also allow formulae to contain the shorthand notations $(v_1 = v_2) \triangleq eq(v_1, v_2)$, $(v_1 \neq v_2) \triangleq \neg eq(v_1, v_2)$, $\varphi_1 \wedge \varphi_2 \triangleq \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\forall v.\varphi_1 \triangleq \neg(\exists v.\neg\varphi_1)$, $\Diamond\varphi_1 \triangleq 1\,\mathcal{U}\,\varphi_1$, and $\Box\varphi_1 \triangleq \neg(1\,\mathcal{U}\,\neg\varphi_1)$. We also use the shorthand $p^*(v_3, v_4)$ for $(TC\ v_1, v_2 : p(v_1, v_2))(v_3, v_4) \vee (v_3 = v_4)$, when $p$ is a binary predicate.

In our examples, the predicates that record information about a single world include the predicates of Tab. 2, plus additional predicates defined in later sections. The set of predicates $\{at[lab](t) \colon lab \in Labels\}$ is parameterized by the set of program labels. Similarly, the set of predicates $\{rval[fld](o_1, o_2) \colon fld \in Fields\}$ is parameterized by the set of selector fields. We use the shorthand notation $rval[x.fld^*](v_1, v_2) \triangleq \exists v'.rval[x](v_1, v') \wedge rval[fld]^*(v', v_2)$. The transitive closure allows specifying properties relating to unbounded length of heap-allocated data structures (e.g., in $rval[fld]^*(v', v_2)$).

We use unary predicates, such as $thread(t)$, to represent type information. This could have been expressed using a many-sorted logic, but we decided to avoid this for expository purposes. Instead, for convenience we define the shorthands $\exists v \colon type.\varphi \triangleq \exists v.type(v) \wedge \varphi$ and $\forall v \colon type.\varphi \triangleq \forall v.type(v) \rightarrow \varphi$.

| **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 world

*Example 1.* Property P2 of Tab. 1 specifies the absence of starvation for worker threads (Fig. 1). The formula $\exists t \colon thread.\Diamond at[lw_c](t)$ states that some thread eventually enters the critical section. The formula $\Box\exists t \colon thread.\Diamond at[lw_c](t)$ expresses the fact that globally some thread eventually enters the critical section.

The property $\Box(\forall v.\ \odot\ v \rightarrow \Diamond \oslash v)$ states that globally, each individual that is allocated during program execution is eventually deallocated. Note that the universal quantifier quantifies over individuals of the world in which it is evaluated. This property is an instance of the commonly used "Response structure" [13, 7], in which an allocation in a world has a deallocation response in some future world.

The properties

$$\forall t \colon thread.\Box(at[l_{lh}](t) \rightarrow \exists v.rval[i.next^*](t, v) \wedge \Diamond(at[l_{lh}](t) \wedge \neg rval[i.next^*](t, v)))$$
$$\forall t \colon thread.\Box(\forall v.at[l_{lh}](t) \wedge \neg rval[i.next^*](t, v) \rightarrow \Box\neg at[l_{lh}](t) \vee \neg rval[i.next^*](t, v))$$

establish a ranking function for linked data structures based on transitive reachability. These properties state that at the loop head $l_{lh}$, the set of individuals transitively reachable from program variable `i` decreases on each iteration of the loop. (Typically `i` is a pointer that traverses a linked data structure during the loop.) Note that these properties relate an unbounded number of individuals of one world to another.

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

The property $\Box(\forall v.\Diamond\Box\forall t\colon thread.\bigwedge_{\substack{x \in Var \\ fld \in Fields}} \neg rval[x.fld^*](t, v) \rightarrow \Diamond \oslash v)$ is a desired property of a garbage collector — that all non-reachable items are eventually collected.

**Evolution Semantics** In the following definitions, $head(\pi)$ denotes the first world in a trace $\pi$, $tail(\pi)$ denotes the suffix of $\pi$ without the first world, and $\pi^i$ denotes the suffix of $\pi$ starting at the $i$-th world. We also use $last(\tau)$ to denote the last world of a finite trace prefix $\tau$.

**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 worlds 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 of $\varphi$ to individuals of a domain $U_{head(\tau)}$, we say that $\pi \models_k Z \rightsquigarrow Z'$ ($Z$* ***evolves into*** *$Z'$ 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 define inductively when an ETL formula $\varphi$ is satisfied over a trace $\pi$ with an assignment $Z$ (denoted by $\pi, Z \models \varphi$) as follows:*

- $\pi, Z \models \mathbf{1}$, *and not* $\pi, Z \models \mathbf{0}$.
- $\pi, Z \models p(v_1, \ldots, v_k)$ *when* $\iota_{head(\pi)}(p)(Z(v_1), \ldots, Z(v_k)) = 1$
- $\pi, Z \models \neg\varphi$ *when not* $\pi, Z \models \varphi$
- $\pi, Z \models \varphi \vee \psi$ *when* $\pi, Z \models \varphi$ *or* $\pi, Z \models \psi$
- $\pi, Z \models \exists v.\varphi(v)$ *when there exists* $u \in U_{head(\pi)}$ *s.t.* $\pi, Z[v \mapsto u] \models \varphi(v)$
- $\pi, Z \models (TC\ v_1, v_2\colon \varphi)(v_3, v_4)$ *when there exists* $u_1, \ldots, u_{n+1} \in U_{head(\pi)}$*, such that* $Z(v_3) = u_1, Z(v_4) = u_{n+1}$*, and for all* $1 \leq i \leq n$,
  $\pi, Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}] \models \varphi$.
- $\pi, Z \models \odot v$ *when* $Z(v) \in A_{head(tail(\pi))}$.
- $\pi, Z \models \oslash v$ *when* $Z(v) \in D_{head(\pi)}$.
- $\pi, Z \models \chi\varphi$ *when there exists* $Z'$ *such that* $tail(\pi), Z' \models \varphi$ *and* $\pi \models_1 Z \rightsquigarrow Z'$.
- $\pi, Z \models \varphi\, \mathcal{U}\, \psi$ *when there exists* $k \geq 1$, $Z'$, *and* $Z''$ *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* $\pi \models_j Z \rightsquigarrow Z''$,

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

It is worth noting that the first-order quantifiers in this definition only range 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 $Z[v \mapsto u]$ binds $v$ to (the evolution of) an individual from the domain of the world over which the quantifier was evaluated (cf. the semantics of $\chi$ and $\mathcal{U}$ ).

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 domain of underlying worlds varies. Verification of ETL properties therefore requires a mechanism for recording the domain related to

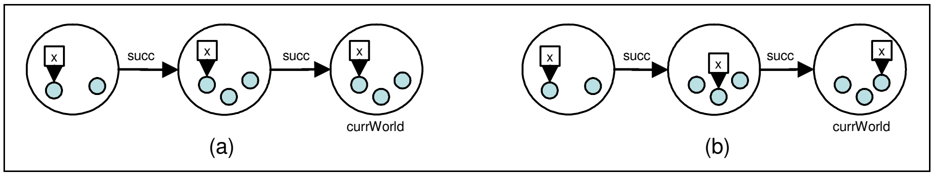


**Fig. 3.** Interaction of first-order quantifiers and temporal operators

each quantifier, and for relating members of quantification domains to individuals of future worlds. For ETL, this mechanism is provided by evolution-mappings, which relate individuals of a world to the individuals of its successor world. Transitively composing evolution-mappings captures the evolution of individuals along a trace.

*Example 2.* The formula $\exists v.\Box x(v)$ states that the pointer variable `x` remains constant throughout program execution, and points to an object that existed in the program's initial world. On the other hand, the formula $\Box\exists v.x(v)$ merely states that `x` never has the value `null`; however, $x$ is allowed to point to different objects at different times in the program's execution, and in particular $x$ can point to objects that did not exist in the initial world. Examples illustrating the two situations are shown in Fig. 3, where in $(a)$ $x$ points to the same object in all worlds, and in $(b)$ it points to different objects in different worlds.

**Definition 7.** *We say that a 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 domain, thus conceptually representing a varying-domain semantics, which allows dynamic allocation and deallocation of objects and threads. In Section 4, we give a possible implementation of this semantics in terms of evolving first-order logical structures.

**Separable Specifications** It is interesting to consider subclasses of ETL for which the verification problem is somewhat easier. Two such classes are: (i) *spatially separable specifications* — do not place requirements on the relationships between individuals of one world; this allows each individual to be considered separately, and the verification problem can be handled as a set of propositional verification problems; (ii) *temporally separable specifications* — do not relate 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 [2, 19].

## 4 Expressing Trace Semantics using First-Order Logic

In this section, we use first-order logic to express a trace semantics; we encode temporal operators using standard first-order quantifiers. This allows us to automatically derive an abstract semantics in Section 5. This approach also extends to other kinds of temporal logic, such as the $\mu$-calculus. Our initial experience is that we are able to demonstrate that some temporal properties, including liveness properties, hold for programs with dynamically allocated storage.

### 4.1 Representing Infinite Traces via First-Order Structures

We encode a trace via an infinite first-order logical structure using the set of designated predicates specified in Tab. 3. 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 and are used to model the allocation and deallocation operators.

**Definition 8.** *A **concrete trace** is a trace encoded as an infinite first-order logical structure $T = \langle U_T, \iota_T \rangle$, where $U_T$ is the domain of the trace, and $\iota_T$ is the interpretation function mapping predicates to their truth value in the logical structure, i.e., for each $p \in \mathcal{P}^k$, $\iota_T(p)\colon U_T^k \to \{0, 1\}$. To exclude structures that cannot represent valid traces, we impose certain integrity constraints [15]. For example, we require that each world has at most one successor (predecessor), and that equality ($eq$) is reflexive.*

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

**Table 3.** Trace predicates.

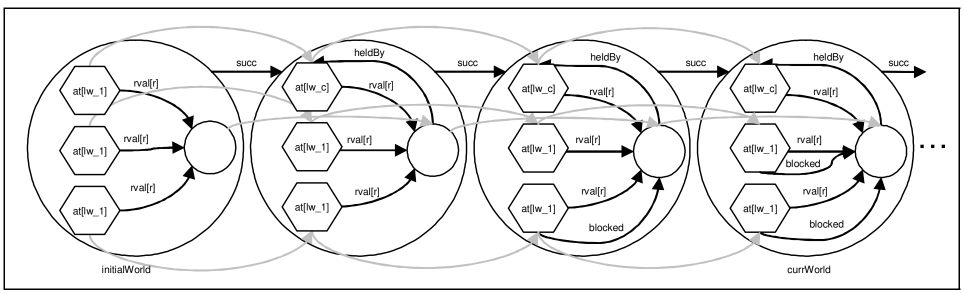


**Fig. 4.** A concrete trace $T_4^\natural$.

*Example 3.* Figure 4 shows four worlds of the trace $T_4^\natural$ where each world is depicted as a large node containing other nodes and worlds along the trace are related by successor edges. Information in a single world is represented by a first-order logical structure, which is shown as a directed graph. Each node of the graph corresponds to a heap-allocated object. Hexagon nodes correspond to thread objects, and small round nodes to other types of heap-allocated objects. Predicates holding for an object are shown inside the object node, and binary predicates are shown as edges. For brevity, we use the label $rval[r]$ to stand for $rval[resource]$. Grey edges, crossing world boundaries, are evolution edges, which relate objects of different worlds. Note that these are the only edges that cross world boundaries.

### 4.2 Exact Extraction of Trace Properties

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

We translate a given ETL formula $\varphi$ to an $FO^{TC}$ formula $(\varphi)^\dagger$ by making the underlying trace structure explicit, and translating temporal operators to $FO^{TC}$ claims over worlds of the trace. The translation procedure is straightforward, and given in Appendix A.

*Example 4.* The property $\exists t\colon thread.\Diamond at[lw_c](t)$ of Example 1 is translated to

$$\exists w : world.\exists t : thread.initialWorld(w) \wedge exists(t, w) \wedge \exists w' \exists t' : thread.succ^*(w, w') \wedge \\ exists(t', w') \wedge evolution^*(t, t') \wedge at[lw_c](t')$$

which evaluates to 1 for the trace prefix of Fig. 4.

**Definition 9.** *The* ***meaning*** *of a formula $\varphi$ over a concrete trace $T$, with respect to an assignment $Z$, denoted by $[\![\varphi]\!]_2^T(Z)$, yields a truth value in $\{0, 1\}$. The meaning of $\varphi$ is defined inductively as follows:*

$$[\![l]\!]_2^T(Z) = l \text{ (where } l \in \{0, 1\}) \qquad [\![p(v_1, \ldots, v_k)]\!]_2^T(Z) = \iota^T(p)(Z(v_1), \ldots, Z(v_k))$$
$$[\![\varphi_1 \vee \varphi_2]\!]_2^T(Z) = \max([\![\varphi_1]\!]_2^T(Z), [\![\varphi_2]\!]_2^T(Z)) \qquad [\![\neg\varphi_1]\!]_2^T(Z) = 1 - [\![\varphi_1]\!]_2^T(Z)$$
$$[\![\exists v_1.\varphi_1]\!]_2^T(Z) = \max_{u \in U^T} [\![\varphi_1]\!]_2^T(Z[v_1 \mapsto u])$$
$$[\![(TC\ v_1, v_2\colon \varphi_1)(v_3, v_4)]\!]_2^T(Z) = \\ max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} min_{i=1}^{n} [\![\varphi_1]\!]_2^T(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

*We say that $T$ and $Z$* ***satisfy*** *$\varphi$ (denoted by $T, Z \models \varphi$) if $[\![\varphi]\!]_2^T(Z) = 1$. We write $T \models \varphi$ if for every $Z$ we have $T, Z \models \varphi$.*

The correctness of the translation is established by the following theorem:

**Theorem 1.** *For every closed ETL formula $\varphi$ and a trace $\pi$, $\pi \models \varphi$ if and only if $rep(\pi) \models (\varphi)^\dagger$, where $rep(\pi)$ is the first-order representation of $\pi$, i.e., the first-order structure that corresponds to $\pi$, in which every world in $\pi$ is mapped to a world in $rep(\pi)$, with the $succ$ predicate holding for consecutive worlds.*

### 4.3 Semantics of Actions

Informally, a program action $ac$ consists of a *precondition* $ac_{pre}$ under which the action is *enabled*, which is expressed as a logical formula, and a set of formulae for updating the values of predicates according to the effect of the action. An enabled action specifies that a possible next world in the trace is one in which 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 $\mathcal{P}$ (see [15]).

## 5 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 an ETL property. A key difficulty in proving liveness properties is the fact that a liveness property might be violated only by an infinite trace. Therefore, our procedure for verifying liveness properties is a greatest fixed-point computation, which works down from an initial approximation that represents all infinite traces. In this section, we present our abstract-interpretation algorithm; procedure `explore` of Figure 8.

Our approach uses finite representations of infinite traces. Finite representations are obtained by abstraction to three-valued logical structures. The third logical value, 1/2, represents "unknown" and may result from abstraction. The abstract semantics conservatively models the effect of actions on abstract representations.

### 5.1 A Finite Representation of Infinite Traces

The first step in making the algorithm of Figure 8 feasible is to define a finite representation of sets of infinite traces. Technically, we use 3-valued logical structures to finitely represent sets of infinite traces.

**Definition 10.** *An* ***abstract trace*** *is a* 3*-valued first-order logical structure* $T = \langle U_T, \iota_T \rangle$*, where* $U_T$ *is the domain of the abstract trace, and* $\iota_T$ *is the interpretation, mapping predicates to their truth values, i.e., for each* $p \in \mathcal{P}^k$*,* $\iota_T(p)\colon U_T^k \to \{0, 1, 1/2\}$*. We refer to the values* 0 *and* 1 *as* ***definite values****, and to* $1/2$ *as a* ***non definite value****.*

*An individual* $u$ *for which* $\iota_T(eq)(u, u) = 1/2$ *is called a* ***summary individual****;*[5] *a summary individual may represent more than one concrete individual.*

*The* ***meaning*** *of a formula* $\varphi$ *over a* 3*-valued abstract trace* $T$*, with respect to an assignment* $Z$*, denoted by* $[\![\varphi]\!]_3^T(Z)$*, is defined exactly as in Def. 9, but interpreted over* $\{0, 1, 1/2\}$*.*

*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* $T, Z \models_3 \varphi$*.*

We now define how concrete traces are represented by abstract traces. The idea is that each individual of a concrete trace is mapped by the abstraction into an individual of an abstract trace. The new two definitions permit an (abstract or concrete) trace to be related to a less-precise abstract trace. Abstraction is a special case of this in which the first trace is a concrete trace. First, the following definition imposes an order on truth values of the 3-valued logic:

**Definition 11.** *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* $l_2 = 1/2$*.*

The embedding ordering of abstract traces is then defined as follows:

**Definition 12.** *Let* $T = \langle U, \iota \rangle$ *and* $T' = \langle U', \iota' \rangle$ *be abstract traces encoded as first-order structures. A function* $f\colon T \to T'$ *such that* $f$ *is surjective is said to* ***embed*** $T$ ***into*** $T'$ *if for each predicate* $p \in \mathcal{P}^k$*, and for each* $u_1, \ldots, u_k \in U$*:*

$$\iota(p(u_1, u_2, \ldots, u_k)) \sqsubseteq \iota'(p(f(u_1), f(u_2), \ldots, f(u_k)))$$

*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 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. We denote the canonical abstraction of a trace $T$ by $t\_embed(T)$. Canonical abstraction guarantees that each abstract trace is no larger than some fixed size, known *a priori*.

*Example 5.* Figure 5 shows an abstract trace, with four abstract worlds, that represents the concrete trace of Fig. 4. An individual with double-line boundaries is a summary individual representing possibly more than a single concrete individual. Similarly, the worlds with double-line boundaries are summary worlds that possibly represent more than a single world. Dashed edges are $1/2$ edges, that represent relations that may or may not hold. For example, a $1/2$ successor edge between two worlds represents the

[5] Note that for all $u \in U_T$, $\iota_T(eq)(u, u) = 1$ or $\iota_T(eq)(u, u) = 1/2$.

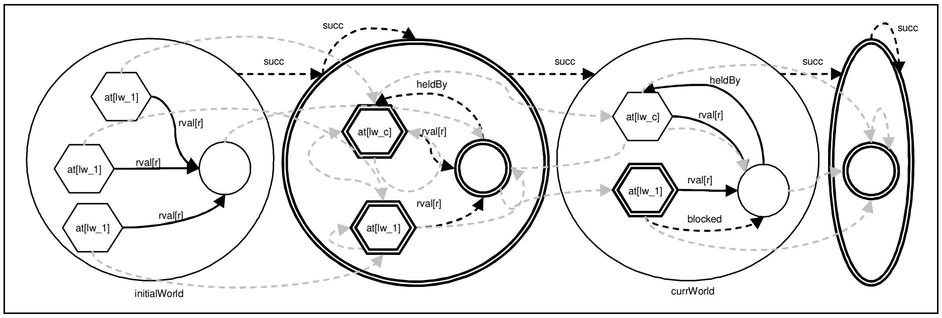


**Fig. 5.** An abstract trace $T_4$ that represents the concrete trace $T_4^\natural$.

possible succession of worlds. The summary world following the initial world represents the two concrete worlds between the initial and the current world of $T_4^\natural$, which have the same values for their unary predicates. Similarly, the summary node labeled $at[lw_1]$ represents all thread individuals in these worlds that reside at label $lw_1$.

Note that this abstract trace also represents other concrete traces besides $T_4^\natural$, for example, concrete traces in which in the current world some threads are blocked on the lock and some are not blocked.

### 5.2 Abstract Interpretation

The abstract semantics represents abstract traces using 3-valued structures. Intuitively, applying an action to an abstract trace unravels the set of possible next successor worlds in the trace. That is, an abstract action elaborates an abstract trace by materializing a world $w$ from the summary world at the tail of the trace; $w$ becomes the definite successor of the current world $currWorld$, and $w$'s (indefinite) successor is the summary world at the tail of the trace. $currWorld$ is then advanced to $w$, which often causes the former $currWorld$ to be merged with its predecessor. When a trace is extended, we evaluate the formula's precondition and its update formulae using 3-valued logic (as in Def. 10).

*Example 6.* Figures 5, 6, and 7 illustrate the application of the action that releases a lock. Figure 6 shows the materialization of the next successor world for the trace $T_4$ of Figure 5. In the successor world, the thread that was at label $lw_c$ no longer holds the lock and has advanced to label $lw_2$. The $currWorld$ predicate is then advanced, and the former $currWorld$ is merged with its predecessor, resulting in the abstract trace shown in Figure 7.

The abstract-interpretation procedure `explore` is shown in Figure 8. It computes a greatest fixed point starting with the set $\{T_1^\top, T_2^\top\}$; these two abstract traces represent all possible concrete (infinite) traces that start at a given initial state. $T_1^\top$ and $T_2^\top$ each have two worlds: an initial world that represents the initial program configuration connected by a 1/2-valued successor edge to a summary world that represents the unknown possible suffixes. The summary world $w_{s1}$ of $T_1^\top$ has a summary individual $u_{s1}$ related to it. The summary individual $u_{s1}$ has $1/2$ values for all of its predicates, including $exists(u_{s1}, w_{s1}) = 1/2$, meaning that future worlds of the trace do not necessarily contain any individuals. The summary world of $T_2^\top$ has no summary individual related to it and represents suffixes in which all future worlds are empty. Figure 9 shows an initial abstract trace (corresponding to $T_1^\top$) representing all traces starting with an arbitrary number of worker threads at label $lw_1$ sharing a single lock.

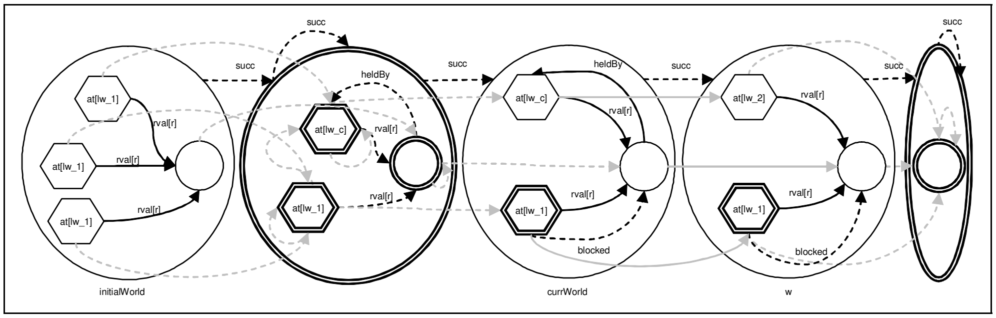


**Fig. 6.** An intermediate abstract trace, which represents the first stage of applying an action to $T_4$.

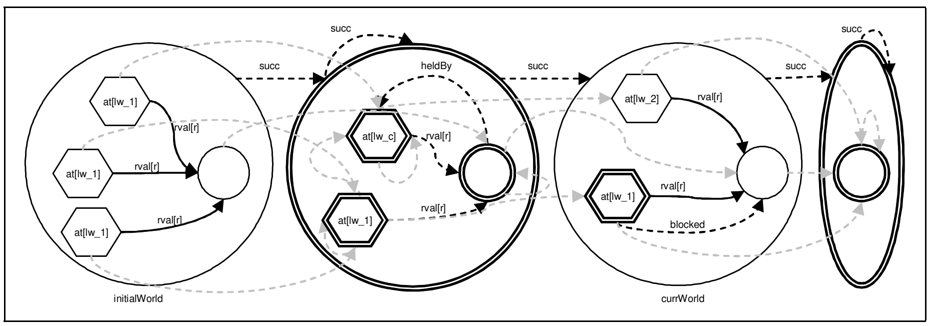


**Fig. 7.** The resulting abstract trace after applying an action over $T_4$ (after advancing $currWorld$).

The procedure `explore` accumulates abstract traces in the set $Traces$ until a fixed point is reached. Throughout this process, however, the set of concrete traces represented by the abstract traces in $Traces$ is actually decreasing. It is in this sense that `explore` is computing a greatest fixed point.

Once a fixed point has been reached, the property of interest is evaluated over abstract traces in the fixed point. Formula evaluation over an abstract trace exploits values of instrumentation predicates when possible (this is explained in the following sec-

`explore() {`
 $Traces = \{T_1^\top, T_2^\top\}$
 `while changes occur {`
  `select and remove` $\tau$ `from` $Traces$
  `for each action` $ac$ `enabled for` $\tau$
   $Traces = Traces \bigcup \{ac(\tau)\}$
 `}`
 `for each` $\tau \in Traces$
  `if` $\tau \not\models_3 (\varphi)^\dagger$ `report possible error`
`}`

**Fig. 8.** Computing the set of abstract traces and evaluating the property $(\varphi)^\dagger$.

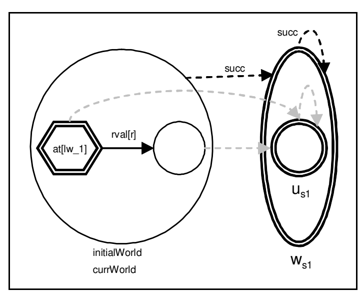


**Fig. 9.** An initial abstract trace $T_1^\top$.

tion). This allows the use of recorded definite values, whereas re-evaluation might have yielded $1/2$.

We now show the soundness of the approach. We extend mappings on individuals to operate on assignments: If $f: U^T \rightarrow U^{T'}$ is a function and $Z: Var \rightarrow U^T$ is an assignment, $f \circ Z$ denotes the assignment $f \circ Z: Var \rightarrow U^{T'}$ such that $(f \circ Z)(v) = f(Z(v))$. One of the nice features of 3-valued logic is that the soundness of the analysis is established by the following theorem (which generalizes [15] for the infinite case):

**Theorem 2.** *[Embedding Theorem] Let $T = \langle U^T, \iota^T \rangle$ and $T' = \langle U^{T'}, \iota^{T'} \rangle$ be two traces encoded as first-order structures, and let $f: U^T \rightarrow U^{T'}$ be a function such that $T \sqsubseteq^f T'$. Then, for every formula $\varphi$ and complete assignment $Z$ for $\varphi$, $[\![\varphi]\!]_3^T(Z) \sqsubseteq [\![\varphi]\!]_3^{T'}(f \circ Z)$.*

The algorithm in Figure 8 must terminate. Furthermore, whenever it does not report an error, the program satisfies the original ETL formula $\varphi$.

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

*Example 7.* Space precludes us from showing a real application, such as the web server. Instead, we use an artificial example, which is also used in the next section. Figure 10 shows an abstract trace in which the property $\exists v.P(v) \,\mathcal{U}\, Q(v)$ holds for all the concrete traces represented by the abstract trace, but the formula $\exists v.P(v) \,\mathcal{U}\, Q(v)$ evaluates to $1/2$ because the successor and evolution edges have value $1/2$.

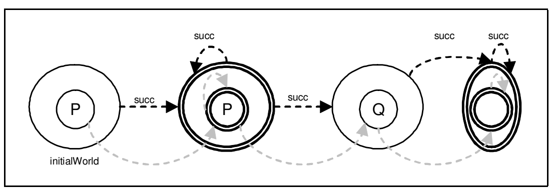


**Fig. 10.** $\exists v.P(v) \,\mathcal{U}\, Q(v)$ holds in all concrete traces that the abstract trace $T_{10}$ represents, yet $\exists v.P(v) \,\mathcal{U}\, Q(v)$ evaluates to $1/2$ on $T_{10}$ itself.

### 5.3 Property-Guided Instrumentation

To refine the abstraction, we can maintain more precise information about the correctness of temporal formulae as traces are being constructed. This principle is referred to in [15] 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.

**Trace Instrumentation** The predicates in Tab. 4 are required for preserving properties of interest under abstraction. The instrumentation predicate $current(o)$ denotes that $o$ is a member of the current world and should be distinguished from individuals of predecessor worlds. This predicate is required due to limitations of canonical embedding. The predicate $twe(o_1, o_2)$ records equality across worlds and is required due to the loss of information about concrete locations caused by abstraction.

*Transworld Equality*: In the evolution semantics, two individuals are considered to be different incarnations of the same individual when one may transitively evolve into

| 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 current world | $\exists w\colon world(o, w) \wedge currWorld(w)$ |

**Table 4.** Trace instrumentation predicates.

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.

Because the abstraction operates on traces (and not only single worlds), 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 in different worlds from a summary node that may represent a number of different individuals.

Transworld equality is illustrated in Fig. 11; the 1-valued $twe$ self-loop to the summary thread-node at label $lw_c$ records the fact that this summary node actually represents multiple incarnations of a single thread, and not a number of different threads.

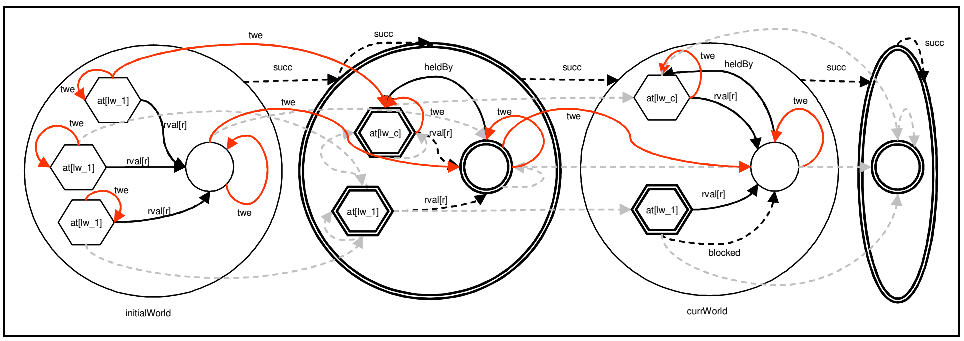


**Fig. 11.** Abstract trace with transworld equality instrumentation (Only 1-valued transworld equality edges are shown).

**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 corresponds to the sub-formulae of the original specification.

*Example 8.* In Example 7, the property $\exists v.P(v)\,\mathcal{U}\,Q(v)$ evaluated to $1/2$ although it is satisfied by all concrete traces that $T_{10}$ represents. We now add the temporal instrumentation predicates $I_p(v)$ and $I_q(v)$ to record the values of the temporal subformulae $P(v)$ and $Q(v)$. The predicates are updated according to their value in the previous worlds. 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 world is an abstraction of different incarnations of the same single object. This is shown in Fig. 12.

## 6 Related Work

The Bandera Specification Language (BSL) [2] allows writing specifications via common high-level patterns. In BSL, it is impossible to relate individuals of different worlds, and impossible to refer to the exact moments of allocation and deallocation of an object.

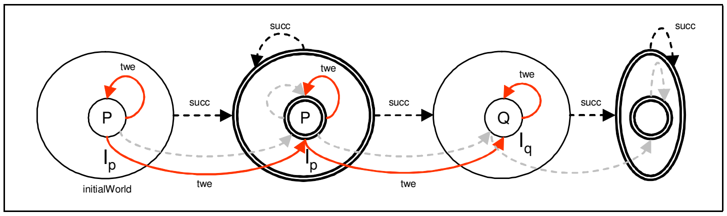


**Fig. 12.** In the abstract trace $T_{12}$, $\exists v.P(v)\,\mathcal{U}\,Q(v)$ evaluates to 1.

In [14], a special case of the abstraction from [18, 19], named "counter abstraction", is used to abstract an infinite-state parametric system into a finite-state one. They use static abstraction, i.e., they have a preceding model-extraction phase. In contrast, in our work abstraction is applied dynamically on every step of state-space exploration, which enables us to handle dynamic allocation and deallocation of objects and threads.

In [19], we have used observing-propositions defined over a first-order configuration to extract a propositional Kripke structure from a first-order one. The extracted structure was then subject to PLTL model-checking techniques. This approach is rather limited, because individuals of different worlds could not be specifically related.

## 7 Conclusion

We believe this work provides a foundation for specifying and verifying properties of programs manipulating the heap with dynamic allocation and deallocation of objects and threads. In the future, we plan to develop more scalable approaches, and in particular abstract-interpretation algorithms that are tailored for ETL.

## Acknowledgments

We would like to thank Patrick Cousot, Nissim Francez, and Amir Pnueli for helpful discussions and insightful comments. We would also like to thank the anonymous referees for providing useful comments on this paper.

## References

1. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
2. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. A language framework for expressing checkable properties of dynamic software. In *SPIN*, 2000.
3. B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In N. Immerman and P.G. Kolaitis, editors, *Descriptive Complexity and Finite Models: Proceedings of a DIAMCS Workshop*, chapter 2, pages 33–57. American Mathematical Society, 1996.
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 *POPL*, 1977.
5. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. of 27th POPL*, pages 12–25, January 2000.
6. N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS*. Springer, 2000.
7. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of Int. Conf. on Software Engineering*, pages 411–421, May 1999.
8. M. Fitting and R.L. Mendelsohn. *First-Order Modal Logic*, volume 277 of *Synthese Library*. Kluwer Academic Publishers, Dordrecht, 1998.
9. G.E. Hughes and M.J. Creswel. *An Introduction to Modal Logic*. Methuen, London, 1982.
10. Y. Kesten, A. Pnueli, and M. Vardi. Verification by augmented abstraction: The automata-theoretic view. *JCSS: J. of Comp. Sys. Sci.*, 62, 2001.

11. T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In *Static Analysis Symposium*. Springer, 2000.
12. D. Lewis. Counterpart theory and quantified modal logic. *Journal of Philosophy*, LXV(5):113–126, 1968.
13. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, 1995.
14. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. CAV 2002.
15. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217.
16. M.Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.
17. E. Yahav. http://www.cs.tau.ac.il/∼yahave.
18. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. of 27th POPL*, pages 27–40, March 2001.
19. 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, CS Dept., Univ. of Wisconsin, Madison, WI, March 2001.
20. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Automatic verification of temporal properties of concurrent heap-manipulating programs using evolution logic. Technical Report 338/02, School of CS, Tel Aviv University, Israel, July 2002.

# A Translation of ETL to $FO^{TC}$

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.

**Definition 13.** *[ETL translation to $FO^{TC}$] We denote by $(\varphi)^{\dagger w}$ the bounded translation of a formula $\varphi$ in a world $w$ and by $(\varphi)^{\dagger}$ the non-bounded translation.*

- $(\varphi)^{\dagger} = \exists w\colon world.initialWorld(w) \wedge (\varphi)^{\dagger w}$
- *if $\varphi$ is an atomic formula other than $\odot x$ and $\oslash x$ then $(\varphi)^{\dagger w} = \varphi$. If $\varphi = \odot x$ then $(\varphi)^{\dagger w} = isNew(x)$. If $\varphi = \oslash x$ then $(\varphi)^{\dagger w} = isFreed(x)$.*
- $(\varphi \wedge \psi)^{\dagger w} = (\varphi)^{\dagger w} \wedge (\psi)^{\dagger w}$, $(\varphi \vee \psi)^{\dagger w} = (\varphi)^{\dagger w} \vee (\psi)^{\dagger w}$, $(\neg\varphi)^{\dagger w} = \neg(\varphi)^{\dagger w}$
- $(\exists x\, \varphi)^{\dagger w} = \exists x.exists(w, x) \wedge (\varphi)^{\dagger w}$
- $((TC\, x_1, x_2\colon \varphi)(x_3, x_4))^{\dagger w} = (TC\, x_1, x_2\colon (\varphi)^{\dagger w} \wedge exists(w, x_1) \wedge exists(w, x_2))(x_3, x_4)$
- $(\varphi(x_1, \ldots, x_n)\, \mathcal{U}\, \psi(y_1, \ldots, y_k))^{\dagger w} =$
$$\begin{array}{l}\exists w'\colon world.\exists y'_1, \ldots, y'_k.succ^*(w, w') \wedge (\psi(y'_1, \ldots, y'_k))^{\dagger w'} \\ \quad \wedge \bigwedge_{1 \le i \le k} evolution^*(y_i, y'_i) \wedge \forall \tilde{w}\colon world.\exists x'_1, \ldots, x'_n.(succ^*(w, \tilde{w}) \\ \quad \wedge succ^*(\tilde{w}, w') \rightarrow (\varphi(x'_1, \ldots, x'_n))^{\dagger \tilde{w}} \wedge \bigwedge_{1 \le j \le n} evolution^*(x_j, x'_j))\end{array}$$
- $(\chi\varphi(x_1, \ldots, x_n))^{\dagger w} =$
$$\begin{array}{l}\exists w'\colon world.\exists x'_1, \ldots, x'_n.succ(w, w') \\ \quad \wedge (\varphi(x'_1, \ldots, x'_n)^{\dagger w'} \wedge \bigwedge_{1 \le j \le n} evolution(x_j, x'_j) \wedge exists(x'_j, w')\end{array}$$

Note that $x_i$ and $y_i$ are not necessarily distinct. Simplified translations may be used for the $\Diamond$ and $\Box$ temporal operators.