LTL Model Checking for Systems with Unbounded Number of Dynamically Created Threads and Objects

Eran Yahav
Thomas Reps
Mooly Sagiv

Technical Report #1424

March 2001
LTL Model Checking for Systems with Unbounded Number of Dynamically Created Threads and Objects

Eran Yahav\textsuperscript{1}, Thomas Reps\textsuperscript{2}, and Mooly Sagiv\textsuperscript{1}

\textsuperscript{1} School of Comp. Sci., Tel-Aviv Univ., Tel-Aviv 69978, \{yahav,sagiv\}@math.tau.ac.il
\textsuperscript{2} CNR-IEI, Pisa, Italy and Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI 53706, USA, reps@cs.wisc.edu

Abstract. One of the stumbling blocks to applying model checking to a concurrent language such as Java is that a program’s data structures (as well as the number of threads) can grow and shrink dynamically, with no fixed upper bound on their size or number. This paper presents a method for verifying LTL properties of programs written in such a language. It uses a powerful abstraction mechanism based on 3-valued logic, and handles dynamic allocation of objects (including thread objects) and references to objects. This allows us to verify many programs that dynamically allocate thread objects, and even programs that create an unbounded number of threads.

1 Introduction

Our goal is to apply temporal-logic model checking to languages such as Java, which allow (i) dynamic creation and destruction of an unbounded number of threads, (ii) dynamic allocation and freeing of an unbounded number of storage cells from the heap, and (iii) destructive updating of structure fields. This combination of features creates considerable difficulties for any method that tries to check program properties:

- Dynamic storage allocation and dynamic thread creation mean that there is no a priori upper bound on either the size of a program’s data structures or the number of threads that arise in the system at execution time.
- Obtaining useful information about linked data structures that can be destructively updated is generally very difficult [30,5,35].

In such a situation, the challenge becomes one of how to obtain a model that abstracts away from the details of the system to be checked, but retains information relevant to the properties to be checked.

This paper shows how to obtain such a model of a concurrent Java program, and how to perform LTL model checking on this model. The primary technical tools used in the paper are as follows: (i) Program configurations are represented as first-order logical structures. The semantics of the program is expressed in
terms of transitions that operate on first-order logical structures. A program’s behavior is modeled by a (potentially infinite) transition system; the use of first-order formulae allows the semantics of pointer manipulations and the dynamic allocation and deallocation of threads and objects to be described in a natural way. (ii) A powerful abstraction mechanism based on 3-valued logic is used, together with methods for performing model checking on 3-valued Kripke structures. (iii) Given a description of the property to be checked as an LTL formula (together with a description of the abstraction to be applied), we automatically construct a finite, conservative, 3-valued transition system that approximates the program’s behavior. (iv) In the course of the latter construction, the abstraction is augmented in a way that “tunes” the analysis algorithm for the property being checked.

In high-level terms, the abstraction principles that we use to obtain a model of a concurrent Java program that may have unbounded numbers of dynamically created threads and objects can be characterized as follows:

**Observation 1. [Model-Extraction Principles]:**

(a) **Abstraction of system configurations to atomic propositions:** As is standard in model checking, path properties to be checked will be formulated in propositional temporal logic. The **vocabulary** of atomic propositions over which the temporal-logic formulae are formulated is defined, in turn, by formula in first-order logic; the values of atomic propositions are obtained by evaluating the first-order formulae on individual system configurations. The first-order formulae provide formal definitions of the distinctions among system configurations that it is possible to talk about; temporal-logic formulae over this vocabulary state desired path properties of a transition system.

(b) **Abstraction of configurations:** Finite—albeit 3-valued—abstractions of a system configuration can be obtained by partitioning the set of individuals (storage cells, threads, etc.) according to the values the individuals’ have for certain unary predicates. The configuration is then represented conservatively by a condensed (3-valued) configuration in which each abstract individual represents an equivalence class of concrete individuals [35].

(c) **Abstraction of system transitions:** A finite—albeit 3-valued—abstraction of a system’s transition system can be obtained by partitioning the set of system configurations according to the values of certain nullary predicates. The transition system is then represented conservatively by a condensed (3-valued) transition system in which each abstract configuration represents an equivalence class of concrete configurations [39].

Obs. 1(b) and (c) are related to the notion of **predicate abstraction** introduced by Graf and Saidi [26], and used subsequently by others [18, 8]. In many ways, the models obtained via Obs. 1 closely resemble the standard ones used for temporal-logic model checking both for the case of finite-state systems [10] and—via predicate abstraction—for infinite-state systems. However, in the models that we obtain, the “truth-value” associated with atomic propositions, transitions, etc. may be the indefinite truth value “unknown”.
We also make use of the following observation, which ties in with Obs. 1 to make model extraction "property-guided":

**Observation 2. [Property-Guidedness]**: *It is possible to arrange for the abstraction steps of Obs. 1 to be influenced by the characteristics of the algorithm applied to check the property of interest. This can allow the extracted models to maintain distinctions that are needed for better precision (i.e., for obtaining a definite answer — true or false, rather than “unknown”) when performing model checking on 3-valued structures.*

In the case of properties stated in LTL, the "characteristics of the algorithm" that we exploit are the states of the Büchi automaton built to check the property of interest.

The contributions of our work can be summarized as follows: (i) We present a method for model-checking of LTL properties of concurrent Java programs. Our method uses an abstraction technique based on 3-valued logic. Previous work [39] only addressed safety properties. (ii) Our abstraction mechanisms are guided by the vocabulary of properties provided by the user. (iii) Our method applies abstraction to the product-automaton constructed from the program and the property being checked. This enables the abstraction to be influenced by the property being checked. (iv) We have implemented a prototype of our framework. It currently supports verification of LTL properties given directly as Büchi automata [40]. This system, and applications of it, will be the subject of a subsequent paper.

![Diagram of model checking and abstraction](image)

*Fig. 1. Model checking and abstraction.*
1.1 Related Work

In many model-checking techniques, the number of threads is fixed, and the
global state of a system is usually described as a fixed-size tuple containing
the program counters of individual threads, and value assignments for shared
variables [10, 20, 9]. In contrast, in our approach, the number of heap-allocated
objects and threads is unbounded and can vary dynamically.

Many related works address model-checking of Java programs without the
use of abstraction, and therefore usually impose an a priori bound on the number
of allocated objects and threads (e.g., [27, 36, 33, 13]).

The popular LTL model-checker SPIN [28], imposes an a priori bound on the
number of allocated objects and threads, and does not support dynamic alloca-
tion of objects. A variant of SPIN, named dSPIN [19], does support dynamic
distribution of objects; however, it can only handle bounded data structures and
a bounded number of threads.

Many approaches have been proposed for the verification of infinite-state
systems [38, 1, 7], and particularly systems with an unbounded number of threads
[4, 31]. Most approaches do not address dynamic allocation of objects.

Most previous work based on predicate abstraction, e.g., [26, 8], does not
address languages that support dynamic allocation and deallocation of objects
and threads. One exception is the work by Das et al. [18], which uses predicate
abstraction to verify properties of a concurrent garbage-collection algorithm.

In [20], the state space is reduced by exploiting symmetries between process
indices. However, the representation used by [20] imposes an a priori bound on
the number of threads, and does not handle dynamic allocation of objects and
threads. In contrast, in our approach, threads are named by so-called canonical
names, which are a collection of thread properties that hold for the thread (see
Obs. 1(b) and Section 5).1 The use of this naming scheme automatically discov-
ers commonalities in the state space, but without relying on explicitly supplied
symmetry properties; there is no need to define a process-naming scheme that
incorporates permutation-equivalences (which may be destroyed anyway in the
presence of dynamic process creation).

Construction of a "product" program from a given program and property was
previously introduced by [11]. However, [11] only addresses safety properties of
sequential programs.

The work described in this paper is an outgrowth of previous work on shape
analysis [30, 5, 35], and specifically the approach of Sagiv, Reps, and Wilhelm,
which is based on 3-valued logical structures [35]. Our work uses abstraction
techniques that handle an unbounded number of thread objects in a manner
similar to the way [35] handles an unbounded number of heap-allocated objects.
This approach was pioneered in [39]; however, the present paper addresses full
LTL, whereas [39] addressed only safety properties.

---

1 One can still express static thread naming in our framework by using unary predi-
cates to denote thread names.
Corbett has applied the results of shape analysis of concurrent programs to reduce the size of finite-state models of concurrent Java programs [12]. In Corbett's work, however, the number of threads is bounded.

Model-checking using multi-valued logics was addressed by [6, 2, 3] in order to reason about partial or inconsistent systems. However, all of the above put an a priori bound on the number of objects and threads, and do not support dynamic allocation and deallocation.

Theoretical aspects of many-valued modal logics were investigated in the past by Fitting (e.g., [21], [22]). In particular, [21] presents a family of many-valued modal logics in which formulae may take values in a many-valued logic, and the accessibility relation between worlds can also take values in the many-valued logic. In this paper we use first-order 3-valued models corresponding to the propositional models discussed in [21].

1.2 Model Checking and Abstraction

Figure 1 gives an overview of the various families of Kripke structures that arise in our work, and the inter-relationships among them. Boxes labeled with $S$ stand for Kripke structures of the system; boxes labeled with $SP$ stand for the product-automaton of the system and the property. Every box in the diagram is labeled with the kind of logical structures used to label nodes in the Kripke structure, and with the number of truth values used. We use four different types of generalized Kripke structures:

- $S_2^P, SP_2^P \in K_2^P$: standard Kripke structures, in which each state is labeled with a set of 2-valued propositions.
- $S_3^P, SP_3^P \in K_3^P$: each state is labeled with a set of 3-valued propositions.
- $S_2^{FO}, SP_2^{FO} \in K_2^{FO}$: each state is labeled with 2-valued first-order structures.
- $S_3^{FO}, SP_3^{FO} \in K_3^{FO}$: each state is labeled with 3-valued first-order structures.

The relationship between $K_2^P$ and $K_3^P$ was previously investigated in [2] to allow reasoning about partial state spaces. In [2], a partial state space is represented using a 3-valued propositional Kripke structure.

The SPIN model-checker [28] follows the path $S_2^P \rightarrow SP_2^P \rightarrow DDFS_2$, which corresponds to model checking with no abstraction. SPIN only uses 2-valued propositional logic (the edge labeled “$\times BA \neg \phi$" stands for the step in which $S_2^P$ is combined with the automaton that represents the negation of the property of interest). SPIN uses the double-DFS algorithm [25, 14] for state-space exploration.

Previous work by the first author [39] corresponds to the path $S_2^{FO} \rightarrow S_2^{FO}$, where the resulting model is later explored for configurations violating a specified safety property.

In this paper, we concentrate on the following aspects of Figure 1:

- Extraction of a 2-valued model: A 2-valued propositional Kripke structure is extracted from a program that may contain dynamic object (and thread) allocations. This corresponds to the path $S_2^{FO} \rightarrow S_2^P$ in Figure 1.
extracted Kripke structure may be infinite, since no abstraction has been applied. (See Section 4.1.)
- Extraction of a 3-valued model that incorporates property-guided abstraction. This corresponds to the path $S_2^{FO} \to S_2^{FO} \to S_3^{FO} \to S_3^{P}$ in Figure 1. (See Section 4.2 and Section 5.)

The remainder of the paper is organized as follows: Section 2 provides an overview of automata-based model checking, and gives some background on Java. Section 3 presents a technique for modelling program behavior using 2-valued logical structures. Section 4 presents a method for model extraction that lays the groundwork for property-guided abstraction. Section 5 describes how 3-valued logical structures are used to perform model checking with abstraction. Section 6 describes a few details of our prototype implementation.

## 2 Preliminaries

### Verification using Linear Temporal Logic

Verification of an LTL property consists of verifying that all of the infinite execution sequences of a program satisfy the property-formula. It is common to use automata-based verification techniques to verify LTL properties [24,14, 37]. Automata-based verification represents the verified system and the desired LTL property using Büchi automata. A Büchi automaton is a five-tuple $A = (\Sigma, S, \Delta_s, S_0, F)$, where $\Sigma$ is the finite alphabet of the automaton, $S$ is the finite set of states, $\Delta_s \subseteq S \times \Sigma \times S$ is the relation of labelled transitions, $S_0 \subseteq S$ is the set of initial states, and $F \subseteq S$ is the set of accepting states.

An execution of $A$ over a word $w$ is an infinite sequence $\rho = s_0, s_1, \ldots$ such that $s_0 \in S_0$ and for each $i \geq 0 : (s_i, w(i), s_{i+1}) \in \Delta_s$. An execution is said to be accepting iff some accepting state $f \in F$ appears in $\rho$ infinitely often. The language $L(A)$ is the set of possible behaviors of the modeled system. Traditionally, verification of an LTL property $\Phi$ consists of the following stages:

- Building a Büchi automaton $A_\varphi = (\Sigma, S_\varphi, \Delta_\varphi, S_0, F_\varphi)$ for $\varphi = \neg \Phi$ (the negation of the property $\Phi$ being checked). In this paper, we assume that the construction of a Büchi automaton for $\varphi$ is performed by an existing algorithm such as [24,17].
- Building a representation of the system as a Büchi automaton $A_s = (\Sigma, S_s, \Delta_s, S_0, S_s)$, where $\Sigma$ is the automaton alphabet, $S_s$ is the set of states, $\Delta_s \subseteq S_s \times \Sigma \times S_s$ is a set of labeled transitions, and $S_0$ is the set of initial states. Note that all states of the system automaton are accepting states.
- Constructing the product of the two automata. The product automaton is denoted by $M = (\Sigma, S_0 \times S_\varphi, \Delta', S_0, S_0 \times F_\varphi)$, where $(s_i, p_j), (l, (s_x, p_y)) \in \Delta'$ iff $(s_i, l, s_x) \in \Delta_s$ and $(p_j, l, p_y) \in \Delta_\varphi$.
- Checking for an accepting cycle that is reachable from one of the initial states in the product automaton. If an accepting cycle is found, it is a counter-example for the specification. Otherwise, the property is proven correct.
The double-DFS algorithm [25, 14] efficiently finds accepting cycles in the constructed product automaton.

Java Concurrency

Our work addresses the problem of verifying properties of concurrent Java programs. This section gives a brief and partial overview of the concurrency model used by Java. Only details necessary for our presentation are given. More details can be found in [32].

Java supports concurrency using a specially designed class java.lang.Thread. Objects of class Thread are concurrently executing activities. Note, however, that from an allocation perspective, a thread object behaves just like any other object.

The constructor for class Thread takes as a parameter an object that implements the Runnable interface, which requires that the object implement the run() method. A thread is created by executing a new Thread() allocation statement. A thread is started by invoking the start() method and starts executing the run() method of the object implementing the Runnable interface.

Initially, a program starts by starting up a single thread, which executes the main() method of a user-specified class. Java assumes that threads are scheduled arbitrarily.

Each Java object has a unique implicit lock associated with it. In addition, each object has an associated block-set and wait-set for managing threads that are blocked on the object’s lock or waiting on the object’s lock. When a synchronized(expr) statement is executed by a thread t, the object expression expr is evaluated, and the resulting object’s lock is checked for availability. If the lock has not been acquired by any other thread, t successfully acquires the lock. If the lock has already been acquired by another thread t’, the thread t becomes blocked and is inserted into the lock’s block-set. A thread may acquire more than one lock, and may acquire a lock more than once. When a thread leaves the synchronized block, it unlocks the lock associated with it.

The example program shown in Figure 2 consists of a request queue holding incoming requests, and a main loop creating a new RequestHandler thread for each request. Each RequestHandler thread repeatedly tries to enter the critical section, perform some operations on the exclusive shared resource (e.g., a logfile), and leave the critical section.

3 Modeling Program Behavior via Logical Structures

This section shows how to construct a model of the analyzed program using logical structures. The section mostly summarizes the work of Yahav [39]. The formal aspects are described in more detail in [39].
public class RequestHandler
implements Runnable {
    public void run() {
        while (true) {
            synchronized(l) {
                // do some critical stuff
            }
        }
    }
}

public class Main {
    public static void main() {
        while (true) {
            if (curr != tail)
                r = curr;
            curr = curr.next;
            t = new Thread(new RequestHandler(r));
            t.start();
        }
    }
}

Fig. 2. Request handler program.

3.1 Representing Program Configurations via Logical Structures

A program configuration encodes a program's global state, which consists of (i) a
global store, (ii) the program-location of every thread, and (iii) the status of locks
and threads, e.g., if a thread is waiting for a lock. First-order logic is used in
this paper to express configurations and their properties. For every analyzed
program, we assume that there is a set of predicate symbols \( P \), each with fixed
arity. A program configuration\(^2\) is a 2-valued logical structure \( C^k = \langle U^k, \iota^k \rangle \),
where

\[- U^k \text{ is the universe of individuals. Each individual in } U^k \text{ represents a heap-}
\text{allocated object (some of which may represent the threads of the program).}
\-
\[- \iota^k \text{ is the interpretation function mapping predicates to their truth-value in}
\text{the structure, i.e., for every predicate } p \in P \text{ of arity } k, \iota^k(p): U^k \rightarrow \{0, 1\}.
\]

Table 1 contains the predicates used to analyze the example program. Note
that predicates in Table 1 are written in a generic way and can be applied to
analyze different Java programs by modifying the set of labels and fields.

In this paper, configurations are presented as directed graphs. Each individual
from the universe is displayed as a node. A unary predicate \( p \) which holds for
a node \( u \) is drawn inside the node. The name of a node is written inside angle
brackets and only used for ease of presentation. A true binary predicate \( p(u_1, u_2) \)
is drawn as directed edge from \( u_1 \) to \( u_2 \) labelled with the predicate symbol.

**Example 31** In the program in Figure 2, handler threads compete for a shared
lock \( l \). Figure 3 shows a possible configuration arising with this program. The
configuration consists of five requests, \( r_1, \ldots, r_5 \) arranged in a queue. Three of
the requests, \( r_1, r_2, \) and \( r_3, \) are already handled by three created threads, \( t_1, t_2, \)
and \( t_3 \). In this configuration, the lock \( l \) has been acquired by the thread \( t_4 \).

Properties of a configuration can be extracted by evaluating logical formulae
with respect to the configuration. For example, the formula \( \exists t: is\_thread(t) \land
held\_by(l, t) \) describes the fact that the lock \( l \) has been acquired by some thread.

\(^2\) In this paper, we use the natural symbol (\( \exists \)) to denote entities of the concrete domain.
**3.2 System Transitions**

Informally, a *transition* is characterized by the following kinds of information:

- A *source label* $l_{\text{src}}$ which is the source label for the transition.
- A *precondition* under which the action is *enabled*. A precondition is expressed as a logical formula. This formula may make use of free variables, including a designated free variable $t_s$ to denote the “scheduled” thread on which the action is performed. Our operational semantics is non-deterministic in the sense that many actions can be enabled simultaneously, and one of them is chosen for execution. In particular, it selects the scheduled thread by an assignment to $t_s$. This implements the interleaving model of concurrency.
- A collection of *predicate-update formulae* (for example, see Table 2). An enabled transition creates a new configuration, where the interpretation of every predicate $p$ of arity $k$ is determined by evaluating a formula $\varphi_p(v_1, \ldots, v_k)$, which may use $v_1, \ldots, v_k$ and $t_s$, as well as other predicates in $P$.
- A *target label* $l_{\text{tgt}}$ which defines the target label for the transition.

We refer to the pair of an enabling-condition and predicate-update-formulae as an *action*. A transition is enabled under an assignment $Z$ in a source configuration $C_s$ only when $at[l_{\text{src}}](t_s)$ holds under $Z$, and $Z$ satisfies the action’s precondition.
Note that the predicates $at[lab](t)$ are just regular unary predicates, and are
encoded as part of the program configuration.

We use special actions for the creation and removal of individuals. The
special action $\text{new}$ creates a new individual $v_{\text{new}}$ and results in a structure
$C^N = (U \cup \{v_{\text{new}}\}, \ell')$. The special action $\text{free}$ removes an individual from
the universe.

Figure 4 shows the transitions for a single RequestHandler thread.
We write transitions in the form $l_{\text{src}} \text{ ac} (\text{args}) l_{\text{tgt}}$ where $ac$ is used to
denote a pair consisting of a precondition and update-formulae (whose de-
tails are given in Table 2). A program transition system (PTS) is a col-
lection of such transitions.

Procedure calls are handled as in [34], and exceptions are modelled
with appropriate transitions. Due to space limitations, we omit addi-
tional discussion of these language features.

Example 32 The transition-system given in Figure 4 corresponds to the state-
ments executed by a single RequestHandler thread. Actions used in the transi-
sion system are defined in Table 2. Note that a single statement (or condition)
may be represented by more than a single transition.

<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Predicate-Update Formulae</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(v)</td>
<td>$\exists t_1 : rvalue(v)(t_1, l) \land held.by(l, l)$</td>
<td>$\text{held.by}(t_1, t_1) = \text{held.by}(l_1, t_1) \lor (t_1 = t_1 \land l_1 = l)$</td>
</tr>
<tr>
<td>unlock(v)</td>
<td>$rvalue(v)(t_2, l)$</td>
<td>$\text{blocked}(t_1, l_1) = \text{blocked}(t_2, l_1) \land ((t_2 \neq t_1) \lor (l_2 \neq l_1))$</td>
</tr>
<tr>
<td>blockLock(v)</td>
<td>$\exists t_3 : rvalue(v)(t_3, l) \land held.by(l, l)$</td>
<td>$\text{blocked}(t_1, l_1) = \text{blocked}(t_3, l_1) \lor (t_1 = t_3 \land l_1 = l)$</td>
</tr>
</tbody>
</table>

Table 2. Operational semantics for lock actions.

We divide our predicates into two (disjoint) sets: core predicates and in-
strumentation predicates. Core predicates serve as the building blocks used in for-
maulae to model the semantics of actions (i.e., the predicates in Table 1 are used
by actions in Table 2). Instrumentation predicates are used to record derived
properties of individuals. Instrumentation predicates are defined using logical
formulae over core predicates. Table 3 lists the instrumentation predicates used
in our example program.
<table>
<thead>
<tr>
<th>Predicate</th>
<th>Intended Meaning</th>
<th>Defining Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>r<a href="l">fld</a></td>
<td>l is referenced by the field fld of some object</td>
<td>$\exists o: rvalue[fld](o, l)$</td>
</tr>
<tr>
<td>is_acquired(l)</td>
<td>l is acquired by some thread</td>
<td>$\exists t: held-by(l, t)$</td>
</tr>
<tr>
<td>rglb, fld(o)</td>
<td>object o is reachable from the object referenced by global[glb] using a path of fld edges</td>
<td>$\exists o_1: global<a href="o_1">glb</a> \land rvalue^*[fld](o_1, o)$</td>
</tr>
</tbody>
</table>

Table 3. Instrumentation predicates for the semantics of a Java fragment.

What has been described above can be thought of as a specification for a variant version of a Kripke structure for the program. In this Kripke structure, a node is labelled with a 2-valued logical structure rather than with a subset of atomic propositions, as is done in most work on model checking. More formally, a $K_5^{FO}$ structure is a four-tuple $k_5^{FO} = \langle S, R, S_0, I \rangle$, where $S$ is the (potentially infinite) set of states, $R \subseteq S \times S$ is the transition relation, $S_0$ is the set of initial states, and $I : S \rightarrow 2\text{-}\text{STRUCT}[P]$ where $2\text{-}\text{STRUCT}[P]$ is the set of general two-valued logical structures over the set of predicates $P$.

4 Model Extraction

This section describes a non-standard approach to the construction of a product automaton. This approach plays a key role in Section 5, which presents an abstraction of the product automaton that (i) is more precise than if we had first built an abstracted version of the system-automaton, and computed the product-automaton afterwards, and (ii) is property-guided in the sense that the abstraction is specific to the property to be verified.

4.1 Extracting Atomic Propositions

In standard model-checking [10], the atomic propositions of the problem are usually presented by flat, and represent the lowest level of information that is made available about the actual actions of the system. In this paper, the atomic propositions are obtained by evaluation of logical formulae in first-order logic. Such formulae are expressed in terms of the core predicates and are evaluated with respect to system configurations.

We call the extracted nullary predicates the vocabulary of the model-checking problem, since the set of such predicates forms the language in which one can express the properties to be verified. The (finite) set of vocabulary predicates is denoted by $I_{\text{voc}}$.

This approach is the embodiment of Obs. 1(a). It explains where atomic propositions come from in the complicated situation that we are dealing with, where objects and threads can be dynamically allocated and deallocated.

We assume that the defining formulae for the vocabulary predicates are provided by user. These formulae provide formal definitions for the distinctions that
the user wishes to be able to make about execution states. Using LTL formulae over this vocabulary then allows the user to state desired path properties.

Given a 2-valued logical structure $T$, and a set of nullary predicates $Q$, $\eta(T, Q) = \{p | p \in Q, \nu_T(p) = 1\}$ is the set of all nullary predicates in $Q$ that hold in $T$. Given a $K_2^F$ structure $k_2^F = (S, R, S_0, I_2)$, the extraction of a Kripke structure from $k_2^F$ is $k = (S, R, S_0, I_1)$, where for all $s$ in $S$, $I_1(s) = \eta(I_2(s), I_{vac})$.

4.2 Representing the Product Automaton

In this section, we use a single $K_2^F$ structure to represent the product of the $K_2^F$ structure for the program, and the Büchi automaton for the negation of the property of interest. This creates a kind of infinite Büchi automaton, but where acceptance is now defined with respect to the finite-cardinality partition of the state space induced by the finite number of accepting states of the property automaton. That is, a path through the constructed $K_2^F$ structure is accepting if infinitely often the path encounters a state whose property-automaton-state component is accepting.

Instead of constructing a system-automaton and property-automaton separately, and then combining them by a product construction, we present a technique in which the product-automaton itself is constructed directly from an instrumented version of the system actions.

Given an LTL property and a concurrent program represented as a transition system, we first construct a property-automaton for the negation of the desired property using a standard construction algorithm. We then create an instrumented version of system-actions to track the state of the desired property for each program configuration. This construction produces the product of the system and the property: every configuration of the constructed $K_2^F$ structure incorporates information about both the state of the system, and the state of the property automaton.

Our approach is to incorporate the state of the property automaton in the configuration as a set of nullary predicates. We do this by adding nullary predicates corresponding to property-automaton states — one predicate for each automaton state. (A more efficient approach is to encode the states of the property-automaton using $\log_2(|\text{states}|)$ predicates. To simplify the presentation, we present only the first approach).

Let $A_\phi = (\Sigma, S_\phi, \Delta_\phi, S_0^\phi, F_\phi)$ be a property automaton where $\Sigma$ consists of subsets of instrumentation predicates from the set $I_{vac}$ (i.e., $\Sigma = 2^{I_{vac}}$). We define

- A set of nullary predicates $\{n | n \in S_\phi\}$ corresponding to states of the automaton. In addition, a designated nullary predicate $\text{accepting}$ is defined to denote accepting states. We denote the set of nullary predicates as $N = \{n | n \in S_\phi\} \cup \{\text{accepting}\}$.
- A set of actions representing automaton transitions. A transition $(s_i, l, s_j) \in \Delta_\phi$ is represented by an action with the precondition $s_i \wedge l$, and a predicate-update formula to update $s_i$, $s_j$, and possibly $\text{accepting}$.
Example 41 Consider the example program given in Figure 2. For the purpose of illustrating the machinery, suppose that we would like to check whether the system has the property “no thread ever acquires the lock referenced by $l$”. An appropriate atomic proposition could be formulated as $i = \neg \exists t, u: \text{is\_thread}(t) \land \text{rv_value}[l](t, u) \land \text{held\_by}(u, t)$. The LTL formula is therefore $Gi$.

We first compute the negation of the property, and generate instrumentation predicates to encapsulate first-order formulae. The negation of the property is $F\neg i$ — i.e., eventually a thread will acquire the lock $l$. An automaton for the negated property is now constructed. The resulting property automaton is given in Figure 5.

The automaton has two states: $S_{out}$ and $S_{in}$. The initial state is $S_0 = S_{out}$. The accepting state is $F_p = S_{in}$.

The generated set of property-nullary predicates is $N = \{n|n \in S_n\} \cup \{\text{accepting}\} = \{S_{out}, S_{in}, \text{accepting}\}$. The corresponding set of actions is given in Table 4.

<table>
<thead>
<tr>
<th>Action</th>
<th>Precondition</th>
<th>Predicate-Update Formulae</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{out} \rightarrow S_{in}$</td>
<td>$S_{out} \land \neg i$</td>
<td>$S_{out} = 0, S_{in} = 1, \text{accepting} = 1$</td>
</tr>
<tr>
<td>$S_{out} \rightarrow S_{out}$</td>
<td>$S_{out} \land \neg i$</td>
<td>$\text{accepting} = 0$</td>
</tr>
<tr>
<td>$S_{in} \rightarrow S_{in}$</td>
<td>$S_{in}$</td>
<td>$\text{accepting} = 1$</td>
</tr>
</tbody>
</table>

Table 4. Actions for the property automaton of Figure 5.

A configuration of the product-automaton is defined as a product of the system-configuration and the property-configuration as follows:

A product configuration is a 2-valued logical structure $C^0 = (U^k, i^k)$ over the predicates $P \cup N$, where each individual in the universe $U^k$ represents a heap-allocated object, for every predicate $p \in P$ of arity $k$, $i^k(p): U^k \rightarrow \{0, 1\}$, and for every predicate $n \in N$, $i^k: n \rightarrow \{0, 1\}$.

Because the precondition for the property-action is formulated in terms of current predicate values, the system-action and the property-action can be composed into a single product-action. Table 5 shows the product of the property-actions with the system’s lock($v$) action. The prime notation denotes the value of a predicate after the action has been applied, and is simply a shorthand for the corresponding predicate-update formula. For example, $\dot{i}'$ denotes the value of the instrumentation predicate $i$ after the system-action has been applied.
<table>
<thead>
<tr>
<th>Product Action</th>
<th>Product Precondition</th>
<th>Product Predicate-Update Formulae</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock(v)×</td>
<td>S_out → S_in</td>
<td>$\neg \exists t \neq t_2 : rvalue[p](t_1, t) \land \neg held_by(l_1, l) \land S_out \land \neg \neg i'$ $\land \neg bounded(l_1, t) \land \neg (t_1 \neq t_2 \lor (l_1 \neq l_2))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$S_{out} = 0, S_{in} = 1, accepting = 1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$i' = \neg rvalue[lock](t_1, l)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>held_by($l_1, t_1$) = held_by($l_1, t_1$) $\land$ ($t_1 = t_2 \land l_1 = l$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>blocked'($t_1, l_1$) = blocked($t_1, l_1$) $\land$ ($t_1 \neq t_2 \lor (l_1 \neq l_2)$)</td>
</tr>
<tr>
<td>lock(v)×</td>
<td>S_out → S_out</td>
<td>$\neg \exists t \neq t_2 : rvalue[p](t_1, t) \land \neg held_by(l_1, l) \land S_out \land \neg \neg i'$ $\land \neg bounded(l_1, t) \land \neg (t_1 \neq t_2 \lor (l_1 \neq l_2))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$i' = \neg rvalue[lock](t_1, l)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>held_by($l_1, t_1$) = held_by($l_1, t_1$) $\land$ ($t_1 = t_2 \land l_1 = l$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>blocked'($t_1, l_1$) = blocked($t_1, l_1$) $\land$ ($t_1 \neq t_2 \lor (l_1 \neq l_2)$)</td>
</tr>
<tr>
<td>lock(v)×</td>
<td>S_in → S_in</td>
<td>$S_{in}$ $\land \neg \neg i'$ $\land \neg bounded(l_1, t) \land \neg (t_1 \neq t_2 \lor (l_1 \neq l_2))$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$i' = \neg rvalue[lock](t_1, l)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>held_by($l_1, t_1$) = held_by($l_1, t_1$) $\land$ ($t_1 = t_2 \land l_1 = l$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>blocked'($t_1, l_1$) = blocked($t_1, l_1$) $\land$ ($t_1 \neq t_2 \lor (l_1 \neq l_2)$)</td>
</tr>
</tbody>
</table>

Table 5. Product actions for the property automaton of Figure 5 and the action lock(v). Note the use of $i'$ in the precondition.

5 Verification with Automatic Abstraction

In this section, we describe how to create a conservative representation of the concrete model presented in Section 3 in a way that provides both finiteness and high precision.

5.1 Representing Abstract Program Configurations via 3-Valued Logical Structures

This section presents the abstraction mechanism of Obs. 1(b). To make it feasible to perform model checking, we conservatively represent multiple configurations using a single logical structure that is finite, but uses an extra truth-value, 1/2, which denotes values that may be 1 or may be 0. The values 0 and 1 are called definite values, the value 1/2 is called an indefinite value. The commutative join operator denoted by $\lor$ is defined as follows: $x \lor x = x, 1/2 \lor x = 1/2, 0 \lor 1/2 = 1/2$.

Formally, an abstract configuration is a 3-valued logical structure $C = (U, \nu)$, where each individual in the universe $U$ represents possibly many allocated heap objects, and for every predicate $p \in P$ of arity $k$, $\nu(p) : U^k \rightarrow \{0, 1, 1/2\}$. An individual $u \in U$ that represents more than a single object is called a summary node.

Example 51 The configuration $C_6$ represents the concrete configuration $C_3^1$ of Figure 3. The summary node labeled $s_{r_1}$ represents the threads $t_2$ and $t_3$, both of which are at the same program label $l_0$. The summary node labeled $s_{r_2}$ represents the requests $r_2$ and $r_3$, both already handled by a handler thread. Note that the abstract configuration $C_6$ essentially represents all concrete configurations that have one or more threads residing at program label $l_0$, when one or more requests are already (possibly) handled. Thus, configuration $C_6$ represents infinitely many concrete configurations. (Note, however, that $C_6$ also represents configurations in which only some of the requests preceding curr are handled).
**Configuration Embedding** We now formally define how concrete configurations are represented by abstract configurations. The idea is that each individual from the concrete configuration is mapped into an individual in the abstract configuration. More generally, it is possible to map individuals from an abstract configuration into an individual in another less precise abstract configuration (note that the concrete configurations are a subset of the abstract configurations).

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

\[
i(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \quad \text{or} \quad \iota'(p(f(u_1), \ldots, f(u_k))) = 1/2
\]

We say that $C'$ represents $C$ when there exists such an embedding function $f$. We denote the fact that $C$ can be embedded in $C'$ as $C \sqsubseteq C'$.

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 a subset $A$ of the unary predicates, called the abstraction predicates. All individuals having the same values for predicate symbols in $A$ are mapped by $f$ to the same abstract individual. This leads to a bounded-size logical structure because there can be no more than $3^{|A|}$ individuals in the resulting structure.

Given a first-order structure $C^3$, we denote the abstraction of $C^3$ over the set of abstraction predicates $A$ by $\text{abs}_A(C^3)$.

Note that canonical abstraction captures commonalities among the different individuals that populate concrete configurations. In particular, for any given set of properties $T$ that characterize threads, there are only $2^m$ possible names of thread equivalence classes. When a transition in the program involves dynamic
allocation (or deallocation), the new abstract configuration automatically adjusts
to take into account the presence (or absence) of the new (or deleted) entity.
By these means, canonical abstraction clusters processes into a finite number
of equivalence classes according to property patterns that actually arise in the
reachable configurations.

Implementing an algorithm for applying actions on abstract configurations is
non-trivial because one has to consider all possible relations on the (possibly infinite)
set of represented concrete configurations. Roughly speaking, this
corresponds to evaluation of the precondition and predicate-update formulae in
3-valued logic. [39] describes how the focus and coerce operations of [35] can be
used to perform rewrites directly on abstract configurations in a conservative
but quite precise way.

5.2 $K_3^{FO}$ Structures

A $K_3^{FO}$ structure is a four-tuple $K_3^{FO} = \langle S, R, S_0, I \rangle$, where $S$ is the set of states,
$R \subseteq S \times S$ is the transition relation, $S_0$ is the set of initial states, and $I : S \rightarrow$
$3\text{-}\text{STRUCT}[P]$ where $3\text{-}\text{STRUCT}[P]$ is the set of 3-valued logical structures over the
set of predicates $P$.

Since every state of a $K_3^{FO}$ structure is labeled with a 3-valued logical structure,
every state may actually represent a (possibly infinite) number of states of the
corresponding $K_3^{FO}$ structure.

Embedding into $K_3^{FO}$ Structures This section presents the abstraction
mechanism from Obs. 1(c), which creates a $K_3^{FO}$ structure $k_3^{FO}$ from a $K_3^{FO}$
structure $k_3^{FO}$. In particular, the structure $k_3^{FO}$ is a finite structure whenever
the FO structures labeling $k_3^{FO}$ have been abstracted via canonical abstraction.

Suppose that $k_2^{FO} = \langle S^2, R^2, S_0^2, I^2 \rangle \in K_2^{FO}$. It is convenient for us to be
able to consider $R^3$ and $S_0^3$ as mappings into Boolean values. That is, the 2-valued
transition relation $R^3$ is now considered to be the characteristic mapping
$R^3 = S^3 \times S^3 \rightarrow \{0, 1\}$. Similarly, $S_0^3$ is now considered to be a characteristic
function that identifies the set of initial states, $S_0^3 = S^3 \rightarrow \{0, 1\}$.

Given a structure $k_3 = \langle S^3, R^3, S_0^3, I^3 \rangle$ in $K_2^{FO}$ and a set of abstraction
predicates $A$ for the FO structures of $k_3$'s node labels, $k$ is the induced $K_3^{FO}$
structure, where

\[ S = \{ abs_A(s) | s \in S^3 \} \]
\[ R = \lambda s_1, s_2. \bigcup_{\{ abs_A(s_1) = s_1, abs_A(s_2) = s_2 \}} R^3(s_1, s_2) \]
\[ S_0 = \lambda s_0. \bigcup_{\{ abs_A(s_0) = s_0 \}} S_0^3(s_0) \]  \hspace{1cm} (1)

Note that $R$ is a 3-valued transition relation, mapping into the values
$\{0, 1, 1/2\}$. Similarly, the 3-valued set of initial structures $S_0$ maps elements
of $S$ into $\{0, 1, 1/2\}$.

Note that transition edges in a condensed $K_3^{FO}$ structure are typically "may-
transition" edges. This contrasts with previous work that has made use of surjective
embeddings for abstracting system models, where 2-valued Kripke structures are mapped to 2-valued Kripke structures (cf. [8]).
Simulation Preorder of $K_3^{FO}$ Structures In this section, we show that a $K_3^{FO}$ structure may simulate a more precise $K_4^{FO}$ structure. We define the simulation preorder between $K_3^{FO}$ structures, and show that the results obtained by evaluating LTL formulae over an abstracted $K_4^{FO}$ structure are conservative. The semantics of 3-valued propositional modal logics is described in [3]. Combining 3-valued modal logic with a first-order logic yields a 3-valued first-order modal logic as the one used in this paper. The reader is referred to [23] for more details on first-order modal logic.

Given $M_3^{FO} = (S_m, R_m, S_m^0, I_m)$ and $Q_3^{FO} = (S_q, R_q, S_q^0, I_q)$, two $K_3^{FO}$ structures, a relation $H \subseteq S_m \times S_q$ is a simulation relation iff for every $s_m \in S_m$ and $s_q \in S_q$ such that $(s_m, s_q) \in H$ the following holds:

- $I_m(s_m) \subseteq I_q(s_q)$, i.e., the configuration labeling $s_m$ is embedded in the configuration labeling $s_q$.
- for every state $s'_m$ such that $(s_m, s'_m) \in R_m$, there exists $s'_q$ such that $(s_q, s'_q) \in R_q$ and $(s'_m, s'_q) \in H$.

We say that $M_3^{FO}$ is simulated by $Q_3^{FO}$, denoted by $M_3^{FO} \preceq Q_3^{FO}$, if there exists a simulation relation $H$ such that, for every initial state $s_0^0 \in S_m^0$, there is a corresponding initial state $s_0^0 \in S_q^0$ such that $(s_0^m, s_0^q) \in H$.

Lemma 51 $\preceq$ is a preorder on the set of $K_3^{FO}$ structures.

Theorem 52 Given two $K_3^{FO}$ structures, $M_3^{FO} \preceq Q_3^{FO}$. For every LTL formula $\varphi$, $[M_3^{FO} \models \varphi] \subseteq [Q_3^{FO} \models \varphi]$.

We now show that the (finite) abstraction of the product-automaton simulates the (possibly infinite) concrete product-automaton. This corresponds to $SP_3^{2FO}$ of Figure 1 simulating $SP_2^{FO}$ of the same figure.

Theorem 53 Let $S_2^{FO} \in K_2^{FO}$ be a 2-valued Kripke structure modelling system behavior, and $BA_{\varphi}$ be the Buchi automaton for the property to be verified. The product automaton constructed using our framework, $SP_3^{FO}$, simulates the (possibly infinite) concrete product automaton $SP_2^{FO}$, i.e., $SP_3^{FO} \preceq SP_2^{FO}$.

5.3 Instrumentation, Abstraction, and the Property-Guidedness Principle

An instrumentation predicate is defined using a logical formula over core predicates. However, rather than evaluating the defining formula for each configuration that arises, instrumentation predicates are explicitly updated by the predicate-update formulae of the actions. The reason for doing things this way is that, in 3-valued logic, the value generated for a configuration by an instrumentation predicate's predicate-update formula—evaluated in the previous configuration—may be more precise than the value obtained by evaluating the instrumentation predicate's defining formula in the (current) configuration. This is known as the Instrumentation Principle [35].
A critical aspect of instrumentation predicates is the fact that they affect the precision of the abstraction applied to configurations. In our framework, abstraction is guided by two things:

- **Unary abstraction predicates**: Individuals having identical values for unary abstraction predicates are mapped into a single abstract individual. Therefore, adding unary abstraction predicates may allow maintaining finer distinctions among individuals.

- **Nullary predicates**: The value of a nullary instrumentation predicate of a 2-valued configuration is unaffected by canonical abstraction (cf. Eqn. (1)). Therefore, when we introduce more nullary predicates, we refine the set of $K^\text{FO}_S$ structures that is induced by canonical abstraction.

Nullary instrumentation predicates are being used here in two ways: (i) to record the vocabulary of the temporal-logic property of interest, and (ii) to record the state of the property automaton (see Section 4.2). Both kinds of nullary predicates introduce distinctions between configurations that are otherwise identical.

The nullary automaton-state predicates are particularly important for improving the precision of the analysis. Because these instrumentation predicates capture different states that are relevant to the verification of the property, the abstraction that they induce is targeted to the structure of the automaton—and hence to the formula that states the property of interest. It is this aspect of our approach that permits us to claim that model extraction is “property-guided” in the sense of Obs. 2. It should also be noted that, because the nullary automaton-state predicates are generated automatically from the automaton for the property of interest, property guidedness is obtained fully automatically.

The vocabulary instrumentation predicates are also important for improving the precision of an analysis, as shown by the following example:

---

**Example 52** The concrete configuration $C_T$ shows a configuration, in which only part of the requests preceding curr in the request queue have been handled.

---

Fig. 7. A concrete configuration $C_T$ with partial request coverage.
The concrete configuration $C_3^b$ corresponds to a similar configuration in which all requests preceding curr have been handled. Both configurations are represented by $C_6$.

Now, suppose that we want to reason about whether all requests preceding curr in the request queue are handled. We formulate this requirement as: $\forall r: r g[\text{head}, \text{next}](r) \land \neg r g[curr, \text{next}] \rightarrow \exists t: is\_thread(t) \land handles(t, r)$.

We would define a vocabulary that includes the instrumentation predicate $rc = \forall r: r g[\text{head}, \text{next}](r) \land \neg r g[curr, \text{next}] \rightarrow \exists t: is\_thread(t) \land handles(t, r)$, and $Frc$ would be the property of interest.

When instrumentation predicates from the vocabulary are ignored, the configurations $C_3^b$ and $C_7^b$ are represented by the same abstract configuration $C_6$, which does not contain definite information about whether all requests preceding curr are handled. That is, $rc$ evaluates to $1/2$ in $C_6$.

Adding $rc$ as an instrumentation predicate refines the abstraction so that it is capable of distinguishing between $C_3^b$ and $C_7^b$; that is, with $rc$ as an instrumentation predicate, $C_3^b$ and $C_7^b$ map to different abstract configurations. Moreover, all abstract configurations of the abstract state-space now record whether “some” versus “all” requests that precede curr in the request queue have been handled.

6 Prototype Implementation

We have implemented a prototype of our framework called 3VMC [40].

Our framework follows the path $S^P_2 \rightarrow SP^P_2 \rightarrow SP^P_3 \rightarrow SP^P_3$ shown in Figure 1. The result of this path is a finite abstract 3-valued propositional model simulating the possibly infinite concrete model. Moreover, the model $SP^P_3$ obtained by following this path may be more precise than the model $SP^P_3$ obtained by following the path $S^P_2 \rightarrow S^P_3 \rightarrow S^P_3 \rightarrow SP^P_3$ since it allows the abstraction to be affected by the property being verified.

Technically, our prototype implementation does not perform the extraction of a $K^P$ model from the $K^P$ model. In our prototype implementation, DDFS performs exploration of the $K^P$ model, building the $K^P$ model on the fly.

The algorithm is conservative: it cannot miss a counter-example, but it might find an artificial counter-example that is caused by the abstraction used. That is, the algorithm may produce false alarms; it may detect a cycle (and return a counter-example) even when the language $L(M)$ is empty. This is a consequence of Theorem 53.

In general, model checking of programs with procedure calls, concurrency, and unbounded data structures is undecidable. Here we have side-stepped this problem by giving an algorithm that—in a finite amount of time—is guaranteed only to provide a safe answer.

The number of abstract configurations that can arise for a program when using a particular abstraction is $O(2^{3|A|+|P|})$ where $|A|$ is the number of abstrac-
tion predicates used for modelling a global state of a program, and \(|P|\) is the number of predicates used to encode the property automaton.

Our experience in [39] indicates that for safety properties, the actual number of configurations arising for a program is significantly smaller than the upper bound. We do not yet have an experience with liveness properties of large programs.

References