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

1424

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

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

[1] School of Comp. Sci., Tel-Aviv Univ., Tel-Aviv 69978,
{yahave,sagiv}@math.tau.ac.il
[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 formulae 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 Buchi automata [40]. This system, and applications of it, will be the subject of a subsequent paper.

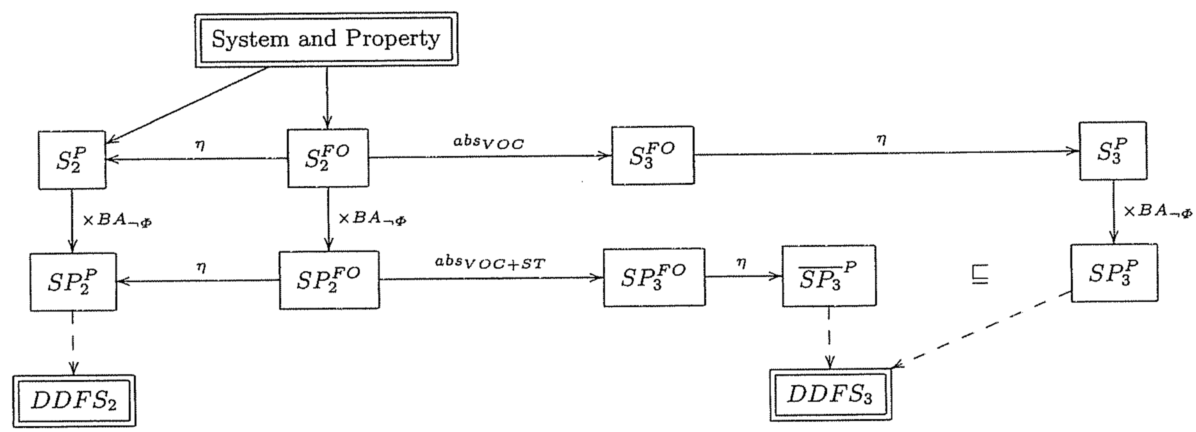

Fig. 1. Model checking and abstraction.

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 allocation of objects. A variant of SPIN, named dSPIN [19], does support dynamic allocation 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 discovers 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 predicates 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 $S_2^{FO} \rightarrow S_3^{FO}$, 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. The 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 $S_2^{FO} \rightarrow SP_2^{FO} \rightarrow SP_3^{FO} \rightarrow \overline{SP}_3^P$ 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 $\mathcal{A} = \langle \sum, S, \Delta_s, S_0, F \rangle$, where $\sum$ 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 $\mathcal{A}$ 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 ρ infinitely often. The language $\mathcal{L}(\mathcal{A})$ is the set of possible behaviors of the modeled system. Traditionally, verification of an LTL property Φ consists of the following stages:

- Building a Büchi automaton $A_\varphi = (\Sigma, S_\varphi, \Delta_\varphi, S_\varphi^0, F_\varphi)$ for $\varphi = \neg\Phi$ (the negation of the property Φ being checked). In this paper, we assume that the construction of a Büchi automaton for φ is performed by an existing algorithm such as [24, 17].
- Building a representation of the system as a Büchi automaton $A_s = (\Sigma, S_s, \Delta_s, S_s^0, S_s)$, where Σ 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_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 $M = (\Sigma, S_s \times S_\varphi, \Delta', S_s^0 \times S_\varphi^0, S_s \times F_\varphi)$, where $(\langle s_i, p_j \rangle, l, \langle s_x, p_y \rangle) \in \Delta'$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 counterexample 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 log-file), 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].

<pre>public class RequestHandler
implements Runnable { ...
  public void run() {
    while (true) {
      synchronized(l) {
        // do some critical stuff
      }
      ...
} } }</pre>
	<pre>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();
	} } } }</pre>

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 $C^\natural = \langle U^\natural, \iota^\natural \rangle$, where

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

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_1.

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

[2] In this paper, we use the *natural* symbol ($\natural$) to denote entities of the concrete domain.

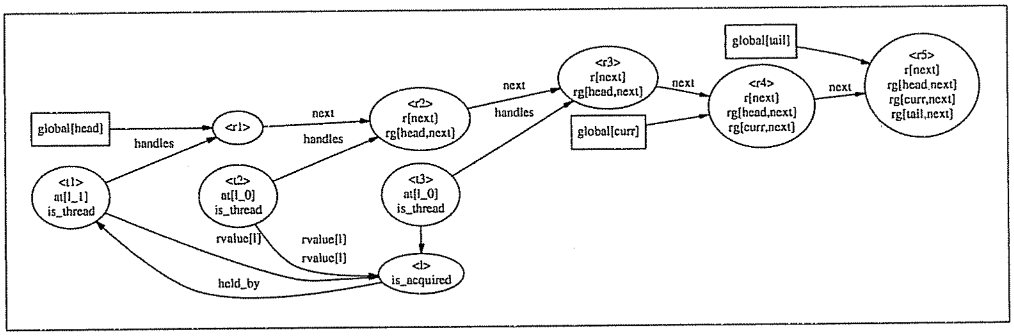

Fig. 3. A possible configuration $C_3{}^{\natural}$ of the example program.

Predicates	Intended Meaning
$is\_thread(t)$	t is a thread
$\{at[lab](t) \colon lab \in Labels\}$	thread t is at label lab
$\{rvalue[fld](o_1, o_2) \colon fld \in Fields\}$	field fld of the object o_1 points to the object o_2
$\{global[fld](o) \colon fld \in Fields\}$	field fld of the global environment points to the object o
$held\_by(l, t)$	lock l is held by the thread t
$blocked(t, l)$	thread t is blocked on the lock l

Table 1. Predicates for the semantics of a Java fragment.

3.2 System Transitions

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

- A *source label* l_{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_{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 $at[l_{src}](t_s)$ 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 `new` creates a new individual u_{new} and results in a structure $C^{\natural'} = \langle U \cup \{u_{new}\}, \iota' \rangle$. The special action `free` removes an individual from the universe.

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

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

l_0 `lock(l)` l_1
l_0 `blockLock(l)` l_0
l_1 `critical()` l_2
l_2 `unlock(l)` l_3
l_3 `skip()` l_0

Fig. 4. Transition system for a Handler thread.

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

Action	Precondition	Predicate-Update Formulae
$lock(v)$	$\neg\exists t \neq t_s : rvalue[v](t_s, l)$ $\wedge held\_by(l, t)$	$held\_by'(l_1, t_1) = held\_by(l_1, t_1) \vee (t_1 = t_s \wedge l_1 = l)$ $blocked'(t_1, l_1) = blocked(t_1, l_1) \wedge ((t_1 \neq t_s) \vee (l_1 \neq l))$
$unlock(v)$	$rvalue[v](t_s, l)$	$held\_by'(l_1, t_1) = held\_by(l_1, t_1) \wedge (t_1 \neq t_s \vee l_1 \neq l)$
$blockLock(v)$	$\exists t \neq t_s : rvalue[v](t_s, l)$ $\wedge held\_by(l, t)$	$blocked'(t_1, l_1) = blocked(t_1, l_1) \vee (t_1 = t_s \wedge l_1 = l)$

Table 2. Operational semantics for lock actions.

We divide our predicates into two (disjoint) sets: *core predicates* and *instrumentation predicates*. *Core predicates* serve as the building blocks used in formulae 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.

Predicate	Intended Meaning	Defining Formula
$r[fld](l)$	l is referenced by the field fld of some object	$\exists o$: $rvalue[fld](o, l)$
$is\_acquired(l)$	l is acquired by some thread	$\exists t$: $held\_by(l, t)$
$rg[glb, fld](o)$	object o is reachable from the object referenced by global[glb] using a path of *fld* edges	$\exists o_1: global[glb](o_1) \wedge rvalue^*[fld](o_1, o)$

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_2^{FO} structure is a four-tuple $k_2^{FO} = \langle S, R, S_0, I \rangle$, 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-STRUCT[P] where 2-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 fiat, 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_{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, $\eta(T,Q) = \{p | p \in Q, \iota_T(p) = 1\}$ is the set of all nullary predicates in Q that hold in T. Given a K_2^{FO} structure $k_2^{FO} = \langle S, R, S_0, I_2 \rangle$, the *extraction* of a Kripke structure from k_2^{FO} is $k = \langle S, R, S_0, I_1 \rangle$, where for all s in S, $I_1(s) = \eta(I_2(s), I_{voc})$.

4.2 Representing the Product Automaton

In this section, we use a single K_2^{FO} structure to represent the product of the K_2^{FO} 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^{FO} 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^{FO} 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 $log_2|states|$ predicates. To simplify the presentation, we present only the first approach).

Let $A_\varphi = (\Sigma, S_\varphi, \Delta_\varphi, S_\varphi^0, F_\varphi)$ be a property automaton where Σ consists of subsets of instrumentation predicates from the set I_{voc} (i.e., $\Sigma = 2^{I_{voc}}$). We define

- A set of nullary predicates $\{n | n \in S_\varphi\}$ corresponding to states of the automaton. In addition, a designated nullary predicate *accepting* is defined to denote accepting states. We denote the set of nullary predicates as $N = \{n | n \in S_\varphi\} \cup \{accepting\}$.
- A set of actions representing automaton transitions. A transition $(s_i, l, s_j) \in \Delta_\varphi$ 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 *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 $i = \neg\exists t, u \colon is\_thread(t) \wedge rvalue[l](t, u) \wedge held\_by(u, t)$. The LTL formula is therefore $\mathrm{G}i$.

We first compute the negation of the property, and generate instrumentation predicates to encapsulate first-order formulae. The negation of the property is $\mathrm{F}\neg i$ — 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.

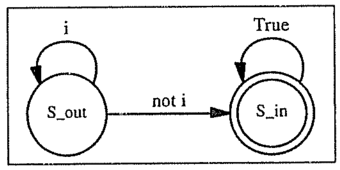

Fig. 5. An automaton for a simple liveness property.

The automaton has two states: S_{out} and S_{in}. The initial state is $S^0_\varphi = S_{out}$. The accepting state is $F_\varphi = S_{in}$.

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

Action	Precondition	Predicate-Update Formulae
$S_{out} \to S_{in}$	$S_{out} \wedge \neg i$	$S_{out} = 0,\ S_{in} = 1,\ accepting = 1$
$S_{out} \to S_{out}$	$S_{out} \wedge \neg i$	
$S_{in} \to S_{in}$	S_{in}	

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 $C^\natural = \langle U^\natural, \iota^\natural \rangle$ over the predicates $P \cup N$, where each individual in the universe $U^\natural$ represents a heap-allocated object, for every predicate $p \in P$ of arity k, $\iota^\natural(p) \colon {U^\natural}^k \to \{0, 1\}$, and for every predicate $n \in N$, $\iota^\natural \colon n \to \{0, 1\}$.

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, i' denotes the value of the instrumentation predicate i after the system-action has been applied.

Product Action	Product Precondition	Product Predicate-Update Formulae
$lock(v) \times$ $S_{out} \rightarrow S_{in}$	$\neg \exists t \neq t_s : rvalue[v](t_s, l)$ $\wedge held\_by(l, t)$ $\wedge S_{out} \wedge \neg i'$	$S_{out} = 0$, $S_{in} = 1$, $accepting = 1$ $i' = \neg rvalue[lock](t_s, l)$ $held\_by'(l_1, t_1) = held\_by(l_1, t_1) \vee (t_1 = t_s \wedge l_1 = l)$ $blocked'(t_1, l_1) = blocked(t_1, l_1) \wedge ((t_1 \neq t_s) \vee (l_1 \neq l))$
$lock(v) \times$ $S_{out} \rightarrow S_{out}$	$\neg \exists t \neq t_s : rvalue[v](t_s, l)$ $\wedge held\_by(l, t)$ $\wedge S_{out} \wedge \neg i'$	$i' = \neg rvalue[lock](t_s, l)$ $held\_by'(l_1, t_1) = held\_by(l_1, t_1) \vee (t_1 = t_s \wedge l_1 = l)$ $blocked'(t_1, l_1) = blocked(t_1, l_1) \wedge ((t_1 \neq t_s) \vee (l_1 \neq l))$
$lock(v) \times$ $S_{in} \rightarrow S_{in}$	S_{in}	$i' = \neg rvalue[lock](t_s, l)$ $held\_by'(l_1, t_1) = held\_by(l_1, t_1) \vee (t_1 = t_s \wedge l_1 = l)$ $blocked'(t_1, l_1) = blocked(t_1, l_1) \wedge ((t_1 \neq t_s) \vee (l_1 \neq l))$

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 $\sqcup$ is defined as follows: $x \sqcup x = x$, $1/2 \sqcup x = 1/2$, $0 \sqcup 1 = 1/2$.

Formally, an *abstract configuration* is a 3-valued logical structure $C = \langle U, \iota \rangle$, where each individual in the universe U represents possibly many allocated heap objects, and for every predicate $p \in P$ of arity k, $\iota(p)\colon U^k \rightarrow \{0, 1, 1/2\}$. 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 $C_3{}^{\natural}$ of Figure 3. The summary node labeled $s_{t.1}$ represents the threads t_2 and t_3, both of which are at the same program label l_0. The summary node labeled $s_{r.1}$ 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).

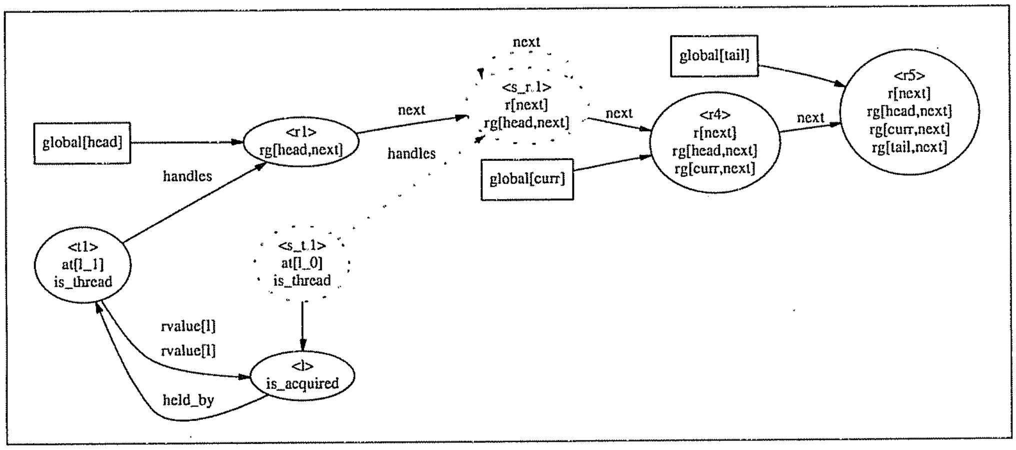

Fig. 6. An abstract configuration C_6 that represents the concrete configuration $C_3{}^{\natural}$.

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 $f \colon U \to U'$ 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:

$$\iota(p(u_1, \ldots, u_k)) = \iota'(p(f(u_1), \ldots, f(u_k))) \quad \textit{or} \quad \iota'(p(f(u_1), \ldots, f(u_k))) = 1/2$$

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 $C^{\natural}$, we denote the abstraction of $C^{\natural}$ over the set of abstraction predicates A by $abs_A(C^{\natural})$.

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^T 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-STRUCT[P] where 3-STRUCT[P] is the set of 3-valued logical structures over the set of predicates $\mathcal{P}$.

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_2^{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_2^{FO} structure k_2^{FO}. In particular, the structure k_3^{FO} is a finite structure whenever the FO structures labeling k_2^{FO} have been abstracted via canonical abstraction.

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

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

$$\begin{aligned} S &= \{abs_A(s) | s \in S^\natural\} \\ R &= \lambda s_1.\lambda s_2. \bigsqcup\nolimits_{\{abs_A({s_1}^\natural)=s_1, abs_A({s_2}^\natural)=s_2\}} R^\natural(s_1, s_2) \\ S_0 &= \lambda s_0. \bigsqcup\nolimits_{\{abs_A({s_0}^\natural)=s_0\}} {S_0}^\natural(s_0) \end{aligned} \tag{1}$$

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_3^{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_3^{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 language 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 $M_3^{FO} = \langle S_m, R_m, S_m^0, I_m \rangle$ and $Q_3^{FO} = \langle S_q, R_q, S_q^0, I_q \rangle$, 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:

- $I_m(s_m) \sqsubseteq I_q(s_q)$, 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 $s_m^0 \in S_m^0$, there is a corresponding initial state $s_q^0 \in S_q^0$ such that $(s_m^0, s_q^0) \in H$.

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 φ, $[M_3^{FO} \models \varphi] \sqsubseteq [Q_3^{FO} \models \varphi]$.*

We now show that the (finite) abstraction of the product-automaton *simulates* the (possibly infinite) concrete product-automaton. This corresponds to SP_3^{FO} 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 $BA_{\neg\Phi}$ 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 in 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_3^{FO} 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:

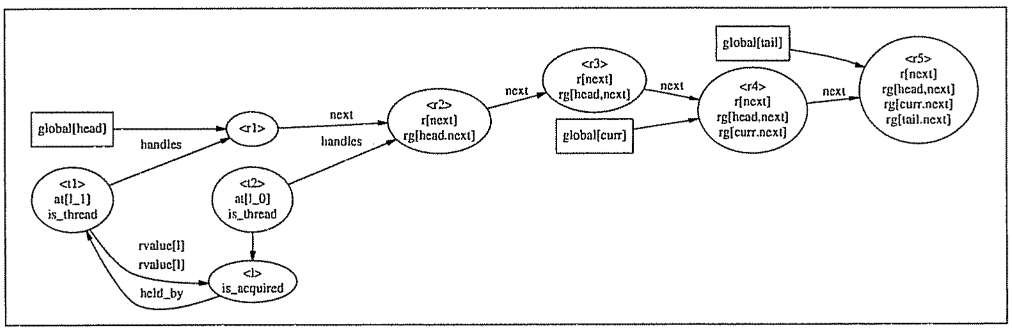

Fig. 7. A concrete configuration C_7 with partial request coverage.

Example 52 The concrete configuration C_7 shows a configuration, in which only part of the requests preceding curr in the request queue have been handled.

The concrete configuration $C_3{}^{\natural}$ 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: $\forall r\colon rg[head, next](r) \wedge \neg rg[curr, next] \rightarrow \exists t\colon is\_thread(t) \wedge handles(t, r)$. We would define a vocabulary that includes the instrumentation predicate $rc = \forall r\colon rg[head, next](r) \wedge \neg rg[curr, next] \rightarrow \exists t\colon is\_thread(t) \wedge handles(t, r)$, and F$rc$ would be the property of interest.

When instrumentation predicates from the vocabulary are ignored, the configurations $C_3{}^{\natural}$ and $C_7{}^{\natural}$ 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 $C_3{}^{\natural}$ and $C_7{}^{\natural}$; that is, with rc as an instrumentation predicate, $C_3{}^{\natural}$ and $C_7{}^{\natural}$ 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 $S_2^{FO} \rightarrow SP_2^{FO} \rightarrow SP_3^{FO} \rightarrow \overline{SP}_3^P$ 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 $\overline{SP}_3^P$ obtained by following this path may be more precise than the model SP_3^P obtained by following the path $S_2^{FO} \rightarrow S_3^{FO} \rightarrow S_3^P \rightarrow SP_3^P$ 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_3^P model from the K_3^{FO} model. In our prototype implementation, DDFS performs exploration of the K_3^{FO} model, building the K_3^{FO} 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 abstraction 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

1. P. Abdulla and A. Nyl'en. Better is better than well: On efficient verification of infinite-state systems, 2000.
2. G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. *Lecture Notes in Computer Science*, 1633:274–287, 1999.
3. G. Bruns and P. Godefroid. Generalized model checking: Reasoning about partial state spaces. In *CONCUR: 11th Int. Conf. on Concurrency Theory*. LNCS, 2000.
4. O. Burkart and B. Steffen. Model checking for context-free processes. In *CONCUR'92*, volume 630 of *LNCS*, pages 123–137, 1992.
5. D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *Conf. on Prog. Lang. Design and Impl.*, pages 296–310. ACM Press, 1990.
6. M. Chechik, S. Easterbrook, and V. Petrovykh. Model-checking over multi-valued logics. In *Proceedings of FME'01, March 2001.*, 2001.
7. E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterized networks. *Trans. on Prog. Lang. and Syst.*, 19(5):726–750, September 1997.
8. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, July 2000.
9. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *Trans. on Prog. Lang. and Syst.*, 16(5):1512–1542, September 1994.
10. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
11. T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of 27th POPL*, pages 54–66, January 19–21, 2000.
12. J. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. October 1998.
13. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, R. Shawn, and L. Hongjun. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd ICSE*, June 2000.
14. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proc. of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 233–242, Berlin, Germany, June 1991. Springer.
15. P. Cousot and R. Cousot. Temporal abstract interpretation. In *Proc. of 27th POPL*, pages 12–25, January 2000.
16. D. Dams, R. Gerth, and O. Grumberg. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19(2):253–291, March 1997.
17. M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. *Lecture Notes in Computer Science*, 1633:249–260, 1999.
18. S. Das, D.L. Dill, and S. Park. Experience with predicate abstraction. In *11th Int. Conf. on Computer-Aided Verification*. Springer-Verlag, July 1999. Trento, Italy.

19. C. Demartini, R. Iosif, and R. Sisto. dSPIN : A dynamic extension of SPIN, September 1999.
20. E. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th Workshop on Computer-Aided Verificaton*, June/July 1993.
21. M. Fitting. Many-valued modal logics. *Fundamenta Informaticae*, 15(3–4):335–3, 1991.
22. M. Fitting. Many-valued modal logics II. *Fundamenta Informaticae*, XVII:55–74, 1992.
23. M. Fitting and R.L. Mendelsohn. *First-Order Modal Logic*, volume 277 of *Synthese Library*. Kluwer Academic Publishers, Dordrecht, 1998.
24. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
25. G.J. Holzmann, D. Peled and M. Yannakakis. On nested depth first search. In *The Spin Verification System*, pages 23–32, 1996. Proc. of 2nd Spin Workshop.
26. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. *LNCS*, 1254:72–83, 1997.
27. K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Int. J.on Soft. Tools for Technology Transfer*, 2(4), April 2000.
28. G. J. Holzmann. Proving properties of concurrent systems with SPIN. In *Proc. of the 6th Int. Conf. on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 453–455, Berlin, GER, August 1995. Springer.
29. M. Huth, R. Jagadeesan, and D.A. Schmidt. Modal transition systems: a foundation for three-valued program analysis. To appear in Proc. of ESOP '01.
30. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *Program Flow Analysis: Theory and Applications*, chapter 4, pages 102–131. Prentice-Hall, Englewood Cliffs, NJ, 1981.
31. J. Knoop. Demand-driven model checking for context-free processes. In *Proc. 5th Annual Asian Comp. Sci. Conf.*, volume 1742 of *LNCS*, pages 201–213, 1999.
32. D. Lea. *Concurrent Programming in Java.* Addison-Wesley, Reading, Massachusetts, 1997.
33. G. Naumovich, G.S. Avrunin, and L.A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proc. of ICSE*, pages 399–410, 1999.
34. N. Rinetskey. Interprocedural shape analysis for recursive programs. 2000. To appear in CC'01. Available at http://www.cs.technion.ac.il/~maon/.
35. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999.
36. S.D. Stoller. Model-checking multi-threaded distributed Java programs. In *Proc. 7th Int. SPIN Workshop on Model Checking of Software*, volume 1885 of *LNCS*, pages 224–244. Springer-Verlag, August 2000.
37. M.Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.
38. P. Wolper and B. Biogelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Computer Aided Verification Conference*, pages 88–97, 1998.
39. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. Proc. of POPL '01, January 2001. Available at http://www.cs.tau.ac.il/~yahave/popl01.ps.
40. E. Yahav. 3VMC user's manual, 2000. Available at http://www.cs.tau.ac.il/~yahave.