Synthesis of Interface Specifications for Java Classes

Rajeev Alur

Pavol Černý

P. Madhusudan

Wonhong Nam

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{alur, cernyp, madhusudan, wnam}@cis.upenn.edu

ABSTRACT

While a typical software component has a clearly specified (static) interface in terms of the methods and the input/output types they support, information about the correct sequencing of method calls the client must invoke is usually undocumented. In this paper, we propose a novel solution for automatically extracting such temporal specifications for Java classes. Given a Java class, and a safety property such as "the exception E should not be raised", the corresponding *(dynamic) interface* is the most general way of invoking the methods in the class so that the safety property is not violated. Our synthesis method first constructs a symbolic representation of the finite state-transition system obtained from the class using *predicate abstraction*. Constructing the interface then corresponds to solving a *partial-information two-player game* on this symbolic graph. We present a sound approach to solve this computationally-hard problem approximately using algorithms for learning finite automata and symbolic model checking for branching-time logics. We describe an implementation of the proposed techniques in the tool *JIST*— Java Interface Synthesis Tool—and demonstrate that the tool can construct interfaces accurately and efficiently for sample Java2SDK library classes.

Categories and Subject Descriptors: D.2.4 [Software Engineering] Software/Program Verification -*formal methods, model checking*; D.2.1 [Software Engineering] Requirements/Specification -*methodologies, tools*; D.2.2 [Software Engineering] Design Tools and Techniques -*modules and interfaces*

General Terms: Algorithms, Verification

Keywords: Behavioral interfaces, synthesis, software components, abstraction, model checking, games, learning regular languages.

POPL'05, January 12–14, 2005, Long Beach, California, USA.

1. INTRODUCTION

Contemporary software development emphasizes components with clearly specified APIs. In current practice, components such as Java library classes have a clearly specified static interface that consists of all the (public) methods, along with the types of input parameters and return values that these methods support. However, there are often implicit constraints on the sequencing of method calls that capture the intended use of the component. For example, for a file system, the method *open* should be invoked before the method *read*, without an intervening call to *close*. While such interfaces can be made precise, using for instance regular expressions as types, these kinds of precise specifications are typically not documented. Such dynamic interfaces for components can help application programmers writing client code for the class, and program analysis tools may even be able to check automatically whether the client code invokes the component correctly. In this paper, we propose a rigorous and automated approach for extraction of dynamic interfaces from existing code for Java classes.

Formally, a (behavioural) *interface* I for a Java class C maps a history of method calls and return values to the methods that can be invoked after this history. Given a set E of *unsafe* valuations for the class variables, we say that the interface I is *safe* with respect to the requirement E if invoking any of the methods allowed by I avoids the state of C to reach E. Typically, the safe set will correspond to requirements such as "the exception e is never raised," or "an error value is never returned." Different applications can employ the same class with different requirements, and different interfaces can be safe for different requirements. There is a natural notion of the *most permissive* interface for a given class C with respect to a given requirement E. Needless to say, typical decision problems concerning this most permissive interface are undecidable. In this paper, we propose a method to algorithmically construct a safe, but not necessarily most permissive, interface that can be represented as a finite-state automaton.

The first step of our solution employs *predicate abstraction*, a powerful and popular technique for extracting finite-state models from complex and potentially infinite-state models [14, 19]. Given a (concrete) Java class C and a finite set $\mathcal{P}$ of boolean predicates over the class variables, A is an abstraction of C with respect to $\mathcal{P}$ such that it has the same set of methods as C, but the input parameters, return values,

and abstract states are (the finitely many) combinations of truth values to the boolean predicates in $\mathcal{P}$. The abstract transition relation over-approximates the concrete one in the standard way. As a result, the abstract class A is nondeterministic: whenever one of its methods is called, there are multiple possible executions that can result in different abstract states and return values.

The interface computation for the abstract class then corresponds to a two-player partial information game. Player 0, the user of the class, chooses to invoke one of the methods. Player 1, the abstract class, chooses a corresponding possible execution through the abstract state-transition graph which results in an abstract return value. Player 0 does not know the current state of the abstract class precisely, and has to choose the next method based on the history of the invoked methods and the values they returned thus far. A strategy for player 0 is *winning* if the game always stays away from the abstract states satisfying the requirement E. A winning strategy for player 0 in this game is a safe interface for the original class C with respect to the requirement E.

The second step of our solution corresponds to computing a winning strategy in the two-player partial information game over the abstract class A with respect to the safety requirement. From classical results concerning partial information safety games [26], it follows that the most permissive winning strategy in this game can be represented by a deterministic finite-state automaton (DFA) I of size exponential in the number of states of A (which, in turn, is exponential in the number of predicates used for abstraction).

We compute the strategy automaton using the L^* algorithm for learning a regular language using membership and equivalence queries [4, 27]. The learning-based approach produces a *minimal* DFA, and the number of queries is only polynomial in the size of the output automaton. Furthermore, this approach allows us to encode the abstract class A symbolically, and we use an existing BDD-based symbolic model checker NuSMV [10] to answer the queries.

The membership query is to test whether all runs of the abstract class A corresponding to a given sequence σ of method calls and return values stay away from the states that satisfy E, and can be posed as an invariant verification problem for the composed model $A\|\sigma$. The equivalence test is to check whether the current strategy automaton J has the same language as the most permissive winning strategy I for A with respect to E. To test $L(J) = L(I)$, we first use the subset query $L(J) \subseteq L(I)$ (i.e., is J safe?). This reduces to checking whether E is an invariant of the model $A\|J$. The model checker NuSMV is used for this test, and if the test fails, the model checker returns a counter-example that can be used by the learning algorithm to update J. The superset query $L(J) \supseteq L(I)$ (i.e., is J most permissive?) cannot be naturally posed as a model checking query: a counter-example is a sequence $\sigma \notin L(J)$ such that *every* run of A on σ stays within the safe region. In fact, we show that the superset query is NP-hard. We hence use approximate and heuristic techniques that employ symbolic model checking to answer the superset query. In summary, our approach terminates with a deterministic finite-state automaton J such that (1) J is the minimal DFA accepting $L(J)$, (2) the number of model checking queries is polynomial in the size of J and in the length of the longest counter-examples obtained while synthesizing the automaton (3) J is safe for A with respect to E, and (4) either J is declared to be the most permissive interface for A, or J is declared to be approximate (and in this case, J is guaranteed to be maximal in the sense that if J' is any interface more permissive than J, we are assured that J' has more states than J does).

We report on an implementation of our solution in a prototype tool called *JIST, the Java Interface Synthesis Tool*. The JIST abstractor processes *Jimple*, an intermediate representation of Java byte code used in the Soot framework [30]. Given an input Jimple class, and a set of predicates, the abstractor transforms the input class line by line, producing a class with only boolean (or enumerated) variables. Currently, only a subset of Jimple is supported and only those abstraction predicates that compare a variable to a constant are handled. The transformed class then is rewritten to a symbolic representation compatible with the input format of the model checker NuSMV. The JIST synthesizer implements the L^* learning algorithm via CTL model checking queries on this symbolic representation using NuSMV.

We report on the performance of the tool on four classes: *AbstractList$ListItr.java*, *Signature.java*, *ServerTableEntry.java* and *PipedOutputStream.java*. In each case, the class file has a few (less than 10) methods with about a hundred lines of code. As a requirement, we choose a particular exception, and as abstraction predicates, we include all conditions of the form "a variable is equal to a constant" that are checked before raising the exception. After the transformations, the input to the symbolic model checker has 20–50 boolean variables. The interface is computed by the synthesizer within a few minutes. More importantly, the interfaces computed by the tool are guaranteed to be safe and maximal, and in practice, seem to capture useful information. For the class *AbstractList$ListItr.java* we also show how the choice of initial predicates impacts the synthesized interface.

Related Work

This work was inspired by the work of Whaley et al. on extracting interfaces from Java classes [32]. In their original work, the focus was on finding pairs of methods (m, m') such that calling m' after m will certainly raise an exception. The solution used static analysis techniques such as constant propagation, augmented with dynamic techniques analyzing execution runs, and extensive experimentation with large applications is reported. Our paper presents a more general and formal solution to interface synthesis. In particular, the interface generated by [32] is not guaranteed to be safe, and the safety requirement, the abstraction, and the states of the interface automaton are hard coded in the solution.

A relevant line of research is the recent work on software verification tools such as Bandera [13], SLAM [6], Feaver [22], and Blast [21]. Our abstraction strategy is closest to the one employed by SLAM for abstracting C code using boolean predicates [5]. In all these works, the focus is on *verifying* the code with respect to user-specified requirements, while our focus is on *synthesizing* a safe interface. Abstract interpretation has been used to automatically generate invariants such as linear constraints over program variables [15, 23].

Using automata as *types* has been explored in programming languages research, particularly for access control [17, 31]. The focus of these efforts is on providing the user with a formal way of specifying interfaces, and enforcing type consistent usage either statically or at runtime. The proposed synthesis approach can be viewed as *type inference* in this context, and is complementary.

There is a rich tradition of research in program analysis aimed at extracting specifications from sample execution traces and related dynamic techniques [3, 24]. These techniques can be effective in learning about the typical usage of the class. However, they cannot provide soundness guarantees that verification techniques such as ours do.

There is an extensive literature on games in the context of design and verification of systems [2, 28] as well as in programming languages semantics [1]. The idea of synthesizing interfaces using games does not seem to appear explicitly in this literature. A related project is Chic, where the authors use games to formalize and check compatibility of user specified interfaces [9, 16].

The work presented in [18] involves synthesis of environments that is similar to our setting, but synthesizes interfaces using *explicit* graph techniques that involve a subset construction of the state space, while our techniques are symbolic involving graphs constructed using predicate abstraction. In [29], the authors present techniques to construct environment models using user-provided environment requirements and by examining environment implementations.

Finally, our use of learning algorithms is related to the work of Cobleigh et al. [12] (see also [7]) who use the L^* algorithm to automatically construct assumptions for compositional verification. In order to verify that the composition of two components C_1 and C_2 satisfies a safety requirement E, the authors propose to learn the assumption I on inputs to C_1 such that $C_1 \| I$ satisfies E and C_2 satisfies I [12]. In this setting, C_2 is given and can be used to answer queries, while in our setting no concrete environment that can aid answering queries is provided.

2. INTERFACES

In this section, we formalize the notion of a symbolic class and its interfaces. A *symbolic class* is a tuple $C = (M, X, \{D_x\}_{x\in X}, X_r, \mathit{Init}, \{T_m\}_{m\in M}, \{R_m\}_{m\in M})$ that consists of the following components:

- M is a finite set of *method names*. For simplicity, we assume that methods do not have input parameters.
- X is a finite set of *variables* used in the class.
- For each $x \in X$, D_x is the domain of the variable x. A *state* $s \in \Pi_{x\in X} D_x$ is a valuation of the variables in X. Let S denote the set of all states. A *predicate* over X is a constraint over the possible valuations, and for a predicate p and a state s, we use $s \models p$ to denote the satisfaction relation.
- $X_r \subseteq X$ is a set of *return variables* used for return values from method calls. We use S_r to denote the set of all valuations for X_r.
- $\mathit{Init}(X)$ is an *initial state predicate* over X. For any state s, if $s \models \mathit{Init}$, then s is an *initial state.*
- Let $X' = \{x' |\ x \in X\}$ be the set of *primed* variables corresponding to the variables in X. For each $m \in M$, $T_m(X, X')$ is a *transition predicate* over $X \cup X'$ for the method m.
- For each $m \in M$, $R_m(X)$ is a *return predicate* over X for the method m.

When a method m is invoked in a state s, the transition relation T_m is applied repeatedly until a *return state* s' satisfying R_m is reached. For any such return state s', the values of the return variables X_r in the state s', denoted $s'[X_r]$, are the corresponding return values. Formally, for states s and s' and a method m, $s \xrightarrow{m} s'$ holds iff there exist states $s = s_0, s_1, \ldots, s_n = s'$ such that $s' \models R_m$, and for all $0 \le i < n$, $s_i \not\models R_m$ and $T_m(s_i, s_{i+1})$.

When a program interacts with a class, it invokes a sequence of methods, say $m_1, \cdots, m_l$, and gets return values $v_1, \cdots, v_l$, where each v_i is in S_r. An *interface* for a class is a function that takes such a history of interaction and prescribes a set of methods that can be called after this interaction. Formally, an interface for a symbolic class C is a function $I : (M \times S_r)^* \to 2^M$. Given a class C and an interface I, $\mathit{runs}(C,\ I)$ is the set of behaviors the class exhibits when used in accordance with the interface, and is the set of all (finite or infinite) sequences $\rho = s_0, m_0, s_1, m_1, s_2, \cdots$ over $S \cup M$ that satisfy the following conditions:

- $s_0 \models \mathit{Init}$ and $m_0 \in I(\varepsilon)$.
- For every $i \ge 0$, $s_i \xrightarrow{m_i} s_{i+1}$.
- For every $i \ge 0$,
$$m_{i+1} \in I((m_0, s_1[X_r]), \cdots, (m_i, s_{i+1}[X_r])).$$

We specify safety requirements using predicates that capture the "bad" states. A run $\rho = s_0, m_0, s_1, m_1, s_2, \cdots$ is said to be *safe* with respect to a predicate E over the class variables in X if $s_i \not\models E$ for every $i \ge 0$. An interface I for C is said to be a *safe interface* with respect to the predicate E if every $\rho \in \mathit{runs}(C, I)$ is safe with respect to E. We refer to E as an *exception predicate* in what follows.

There is a natural ordering among interfaces: if I and I' are two interfaces for a class C, then we say that I is *more permissive* than I' if for all $\sigma \in (M \times S_r)^*$, $I'(\sigma) \subseteq I(\sigma)$. That is, if I is more permissive than I', then after any possible history of calls and returns, I allows invoking any method that is allowed by I'. The *most permissive safe interface* for C, if one exists, is a safe interface I such that for every safe interface I', I is more permissive than I'. For a given exception E, it is easy to see that the most permissive safe interface exists.

Example: Signature Class

To illustrate the definition of an interface, we consider the class `Signature` from the package `java.security` from Java2SDK. The `Signature` class provides the functionality of a digital signature algorithm. We pick the five most interesting methods for interface synthesis: `initSign(), initVerify(), sign(), verify()` and `update()`. There are eight other methods in the class. Users sign by invoking `sign()` and check the input signature using `verify()`. Both operations need initialization via `initSign()` and `initVerify()`, respectively. Once such initialization method is invoked, the user can also update the data to be signed or verified by `update()`.

A finite automaton (DFA) over the alphabet $(M \times S_r)$ can be interpreted to represent an interface. For such a DFA J, the corresponding interface I is defined as: $I(\sigma) = \{m \in M \mid \exists s_r \in S_r, \sigma \cdot (m, s_r) \in L(J)\}$. For example, Figure 1 illustrates an interface automaton. In this example, $X_r = \emptyset$ and

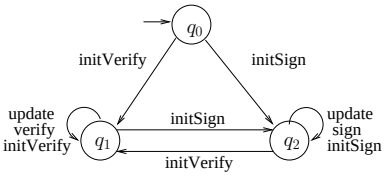

Figure 1: Signature

we use M instead of $M \times S_r$; hence the labels on edges are over M. All states are final, and missing transitions indicate disallowed calls. For instance, $I(\texttt{initSign initVerify})$ is the set of all methods on the outgoing transitions from q_1, that is, {`initSign, initVerify, update, verify`}. The formal notion of how automata describe interfaces is explained in Section 5.1. It is easy to see that interface automata capturing the most permissive interface do exist if the symbolic class is finite (i.e. each domain D_x is finite), but may not exist otherwise.

We say a safe interface automaton J for a class C is *maximal* if it is true that for every safe interface automaton J' with $L(J) \subseteq L(J')$ (i.e. where J' is more permissive than J), J' has more states than J does. Note that the minimal automaton describing the most permissive safe interface automaton is, by definition, maximal.

3. OVERVIEW OF JIST

Given a symbolic class C, presented as a class written in Java, our aim is to find a safe (and if possible the most permissive) interface for C. Our solution is in two steps. The first step, described in Section 4, involves abstraction of a Java class into a boolean symbolic class, that is, a class whose variables are booleans (or of enumerated type), and all predicates are expressed as propositional formulas. This step obviously introduces a loss of information, but leads to a representation that can be manipulated by symbolic computational techniques such as BDD-based model checkers. The second step requires solving a partial-information 2-player safety game over the abstract boolean class. We wish to output a finite interface automaton over the alphabet $M \times S_r$ that captures the language of method sequences allowed by the interface (as in Figure 1). However, since this problem is hard and infeasible to compute accurately, our method strives to construct the most permissive interface failing which it will produce an approximation with the guarantee that the interface output is maximal. The synthesis step is presented in Section 5.

The complete tool chain is given in Figure 2. On the implementation level, the following transformations are performed:

1. *Java to Jimple*: The Java source is compiled into Java byte code and then the Soot tool is used to transform it to the Jimple format. Jimple is a 3-address representation that has been designed to simplify analysis and transformation of Java byte code and is thus suitable for our purpose. In the current implementation of the prototype tool, a pre-processing step is done manually on the Java source code to replace the 'throwing' of exceptions (other than the ones we are interested in for the interface) to statements that return 0.

2. *Jimple to Boolean Jimple*: The Jimple to Boolean Jimple conversion is done by the *Predicate abstractor*. The input to the abstractor is a Jimple file along with a set of predicates. The algorithm and the implementation are described in Section 4.

3. *Boolean Jimple to NuSMV*: The boolean The Jimple code is converted to a boolean model in the NuSMV language.

4. *Synthesizing an interface*: The *Interface Synthesizer* takes in the NuSMV code for the boolean abstract class and an exception predicate E, and computes an interface automaton that is either the most permissive interface or a maximal safe interface. We use a regular-language learning algorithm and CTL symbolic model checking for the synthesis. The synthesis algorithm and its implementation are described in Section 5.

Using JIST to synthesize interfaces

A user of JIST can go about synthesizing interfaces in the following manner. First, the user identifies the exception predicate for the class with respect to which the interface is required, and also an initial set of predicates that may be useful in extracting an appropriate interface. In our experiments, the exception predicate corresponds to a particular exception being thrown, and the initial predicates are chosen as the predicates in the conditionals that guard the throwing of this exception.

The user then runs the tool to get a maximal safe interface J with an indication of whether or not the interface is guaranteed to be the most permissive for the chosen abstraction. The user can then examine this interface to see whether it adequately describes all behaviors one would expect from the class. For example, in one of the experiments (`ListItr`), it turned out that the initial interface was too weak as it completely ruled out calling certain methods. Checking adequacy of the interface is something that is currently left entirely to the user; indeed, there can be several ways to check this (for example, the user could check conformance of typical clients of the class to the interface).

If the interface is not adequate and the tool has not guaranteed that J is the most permissive interface, then the user can find a method-call sequence that she believes must be allowed by the interface. The tool can then take this string and examine whether it is safe (using a membership query as explained in Section 5), and if it is safe, use it as a counter-example to the superset query (see below for details) to continue learning a better interface.

If the user-supplied string is not safe or if the tool has assured that the current interface is the most permissive, then then the user must provide new predicates that refine the abstraction and rerun the tool.

Discovering abstraction predicates automatically is an active area of research in software verification (see [6, 20]), and some of these techniques are potentially useful for the purpose of synthesizing interfaces, but this is left for future work.

4. PREDICATE ABSTRACTION

4.1 Sound abstractions for symbolic classes

Predicate or *boolean abstraction* uses boolean variables corresponding to assertions (predicates) over the states of a

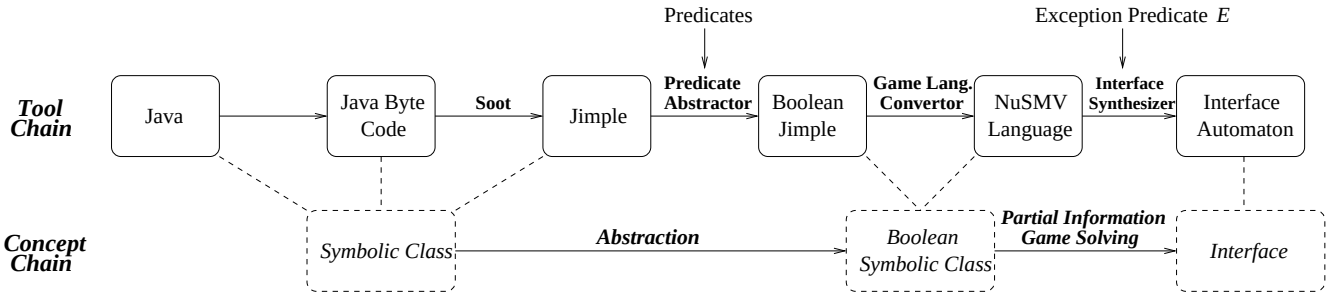

Figure 2: The complete tool chain

program to form a coarse model of the program. In predicate abstraction, we take a set of predicates $\mathcal{P}$ over the concrete set of variables X and treat these predicates as abstract variables. We then transform the class so that it keeps track of these abstract variables rather than the concrete variables. Note that if there is a finite number of predicates, abstraction reduces the (possibly infinite) concrete state space into a finite state space.

When abstracting a class, we must ensure that all behaviors of the program correspond to some behavior in the abstracted class. It will then follow that if we design a safe interface for the abstract class, then the interface will indeed be safe for the concrete class as well. We formalize below a standard set of rules under which the abstraction captures all concrete behaviors. We restrict ourselves to abstraction predicates such that if a predicate refers to a variable in X_r, then it refers only to the variables in X_r, that is, to predicates over X_r or over $X \setminus X_r$.

Let $C = (M, X, \{D_x\}_{x\in X}, X_r, \mathit{Init}, \{T_m\}_{m\in M}, \{R_m\}_{m\in M})$ be a symbolic class and let $\mathcal{P}$ be a set of abstraction predicates. A symbolic class $A = (M, X^a, \{D_A\}_{x\in X^a}, X^a_r, \mathit{Init}^a, \{T^a_m\}_{m\in M}, \{R^a_m\}_{m\in M})$ is said to be an abstraction of C with respect to $\mathcal{P}$ if the following conditions hold. The set of methods stays the same. The abstract domain D_A is the set $\{0, 1, *\}$, where, intuitively, 0 stands for an abstract predicate being false, 1 stands for it being true, and $*$ stands for it being unknown. $X^a = \{x_P \mid P \in \mathcal{P}\}$ is a set of variables over the domain D_A, one for each predicate in $\mathcal{P}$. A state s^a of the abstract symbolic class is hence an element of $S^a = \Pi_{x\in X^a} D_A$. Let $\gamma : S^a \to S$ be the concretization function defined by:

$$\gamma(s^a) = \{s \mid \forall P \in \mathcal{P}, s \vdash P \Rightarrow s^a[x_P] \neq 0, \quad s \nvdash P \Rightarrow s^a[x_P] \neq 1\}$$

We say that s^a abstracts s if $s \in \gamma(s^a)$.

The abstract class must further satisfy:

- X^a_r is the set of abstract variables x_P such that P is a predicate that involves some variable in X_r.
- For every abstract state s^a, if there exists a state $s \in \gamma(s^a)$ such that $s \models \mathit{Init}$, then $\mathit{Init}^a(s^a)$ holds.
- For all $m \in M$, if for some $s \in \gamma(s^a)$ and $s' \in \gamma(s'^a)$, $T_m(s, s')$ holds, then $T^a_m(s^a, s'^a)$ holds.
- For all $m \in M$, if for some $s \in \gamma(s^a)$, $s \models R_m$, then $s^a \models R^a_m$.

Also, we require that for all $m \in M$, $R_m \in \mathcal{P}$. Note that the abstraction of C with respect to $\mathcal{P}$ is not unique, because of the flexibility in the abstraction of the transition predicate.

Let C be a symbolic class and let A be its abstraction with respect to $\mathcal{P}$. Let $\gamma^* : (M \times S^a)^* \to (M \times S)^*$ be the natural extension of γ.

Let J be an interface of the class A. An interface I_J for the class C is a concretization of J if $I_J(\sigma) = J(\sigma^a)$ whenever $\sigma \in \widehat{\gamma}(\sigma^a)$ where $\widehat{\gamma}$ is the natural extension of γ to $(M \times D_{X_r})^*$ and D_{X_r} is the product of the domains of return variables of the class A.

LEMMA 1. $\mathit{runs}(C, I_J) \subseteq \gamma^*(\mathit{runs}(A, J))$

Given the exception predicate E, we define the abstract predicate E^a as follows: for all $m \in M$, if for some $s \in \gamma(s^a)$, $s \models E$, then $s^a \models E^a$. The next theorem captures correctness of the abstraction: whenever an abstract interface is safe for the abstracted class, then the concretization of the interface is safe for the concrete class.

THEOREM 1. *Let C be a symbolic class. Let A be its abstraction with respect to $\mathcal{P}$. Let J be a safe interface for the class A with respect to the exception predicate E^a. Then I_J is a safe interface for the class C with respect to E.*

4.2 Abstracting Jimple programs

A Jimple program can be seen as a symbolic class. Our tool abstracts the Jimple class into a boolean Jimple program. A boolean Jimple program is simply a Jimple program that has only boolean or enumerated data types. The tool is implemented on top of Soot, a framework for optimizing Java bytecode. This framework is implemented in Java, and supports a number of representations for Java bytecode. Our tool uses the API of the Soot framework to perform transformations on Jimple code.

Our abstractor works on a subset of Jimple (the grammar of the subset that we handle is available at `http://www.cis.upenn.edu/jist`). A large part of core classes of Java 2 falls into this subset and thus can be analyzed by our tool. The main features Jimple that are not treated here are floating-point types, arrays, recursive function calls, and exceptions (apart from the exception with respect to which we are constructing the interface).

The abstraction algorithm proceeds line-by-line on the original program, and is inspired by the SLAM toolkit [5]. In the SLAM tool, an automatic theorem prover is used to compute abstractions. In our prototype implementation, the abstractions of Jimple statements can be precomputed, since we use predicates of a simple form and the tool abstracts simple expressions (and safely overapproximates the

rest). The predicates are of the form $x = k$ where x is a variable and k is a constant. We denote such a predicate by the abstract variable `b(x,k)` over the three-valued domain $\{0, 1, *\}$. Since Jimple does not have a nondeterministic conditional, we create a class `TriBool` and make each predicate that we add an instance of this class. In abstracting Jimple code, the values of the `TriBool` variables are set and read using unimplemented methods. As a consequence, our boolean abstractions are valid Jimple programs that can be executed with any implementation of `TriBool` (for example, it can randomly resolve the nondeterminism).

Let P be a predicate and Pr be a set of predicates used for abstraction. We need the following notions in order to define the abstraction algorithm on Jimple statements:

- **WP(st,P)** (weakest precondition) is the weakest predicate over the concrete variables X whose truth before a statement **st** entails the truth of **P** afterwards.
- The command **assume(P)** silently terminates if **P** evaluates to *false*. By 'silently terminates' we mean that the method halts without returning and such runs are ignored in the analysis.
- **Implies(Pr)(P)** is the best boolean function on **Pr** that implies **P**, i.e. **Implies(Pr)(P)** $\Rightarrow$ **P**, and if **F** is a boolean function on **Pr** such that **F** $\Rightarrow$ **P**, then **F** $\Rightarrow$ **Implies(Pr)(P)**.
- **ImpliedBy(Pr)(P)** is the best boolean function on **Pr** that is implied by **P**, i.e. **P** $\Rightarrow$ **Implies(Pr)(P)**, and if **F** is a boolean function on **Pr** such that **P** $\Rightarrow$ **F**, then **ImpliedBy(Pr)(P)** $\Rightarrow$ **F**.

In general, a statement `st` is abstracted into the following sequence of commands: (one command for each predicate p in Pr):

```
b(p) = Implies(Pr)(WP(st,p)) ? true :
       Implies(Pr)(WP(st,not p)) ? false : *
```

It is well-known that if we abstract a program statement by statement in this way, we will get a sound abstraction. However, it should be noted that in order to prove formal correctness of the abstractor, we need to formally associate a symbolic class $C(J)$ with a Jimple class, and prove that if the abstractor transforms a class J to J', then $C(J')$ is an abstraction of $C(J)$ as defined in Section 4.1.

We present here the abstraction of assignments, conditional statements and method calls. The abstraction of other statements is quite straightforward in our setting.

Abstraction of assignments

Consider a sample Jimple statement

```
x=y+1
```

Let $(y, l1) \cdots (y, ln)$ be the set of predicates involving y. Note that if one of these predicates is true, the others are false. For all predicates of the form (x, k), we get the following series of abstracted statements if $(y, k-1)$ is in the predicate list:

```
b(x,k)=b(y,k-1)
```

If $(y, k-1)$ is not in the predicate list, we get:

```
if (b(y,l1)==TRUE or ... or b(y,ln)==TRUE) then
  b(x,k)=FALSE
else
  b(x,k)=*
```

Abstraction of conditionals

Consider a sample statement

```
if (x==k) then C1 else C2
```

If (x, k) is in the predicate list, the abstracted class contains:

```
if b(x,k)==*
  resolve-nondeterminism{b(x,k1),...,b(x,kn)}
if b(x,k)==TRUE
  Abs(C1) // abstracted code for C1
else // b(x,k)==false
  Abs(C2) // abstracted code for C2
```

If (x, k) is not in the predicate list:

```
if b(x,k1)==* or ... or b(x,kn)==*
  resolve-nondeterminism{b(x,k1),...,b(x,kn)}
if b(x,k1)==TRUE or ... or b(x,kn)==TRUE
  Abs(C2) // abstracted code for C2
else
  // Every b(x,ki)=FALSE
  Abs(C1) // abstracted code for C1
```

where $(x, k1) \cdots (x, kn)$ are the set of predicates involving x. In the above translation, `resolve-nondeterminism(b(x,k1), ...,b(x,kn))` sets all variables whose value is $*$ to `TRUE` or `FALSE` in such a way that if for any i, $b(x, ki)$ is set to `TRUE`, then $b(x, kj)$ is set to `FALSE`, for all j different than i.

Abstraction of method calls

Currently, calls to other methods are inlined in the code and abstracted. If the method is not implemented, then it is abstracted in a coarse way: an assignment that contains a method call to f

```
x=f()
```

is replaced by:

```
b(x,k1)=*
...
b(x,kn)=*
```

where $(x, k1) \cdots (x, kn)$ are the set of predicates involving x. However, if the specifications of what the method must do are known, we replace the call with this specification (manually) using appropriate Java code.

5. INTERFACE SYNTHESIS

In this section, we describe the interface synthesis algorithm for a given (abstracted) boolean symbolic class A. It turns out from standard results in games that the most permissive safe interface can be captured by a finite automaton, and hence corresponds to a regular language that we shall henceforth refer to as U.

The standard way to generate the interface would be to generate, using a subset construction, a new complete-information game. In this new game, after any interaction with the class, the player playing the role of the interface would keep track of the *set* of states the original game can be in. This new game can then be solved using a standard fix-point computation that computes the winning positions in the game. However, solving a partial information game in this way requires manipulating sets of states of A. Recall that a state

of A is a valuation of the boolean predicates, and the transitions of A are given symbolically. Working with sets of states of A symbolically seems hard, and explicitly enumerating the state space of A is unaffordable.

Furthermore, in our setting, we expect the interface to be a much smaller DFA than the abstract class, and solving the game using the above method makes us pay an exponential cost in computation time, regardless of the size of the strategy we want to build.

To avoid this problem, we have chosen to implement the interface synthesis using a learning algorithm; the learning algorithm tries to learn the most permissive safe interface. We use a standard algorithm to learn regular languages called the L^* algorithm [4, 27]. The L^* algorithm is an algorithm that learns an unknown regular language U (in our setting, U is the language of the most permissive safe interface) by asking two kinds of queries to a teacher who knows U: membership queries (i.e. asking whether a given string σ is in U) and equivalence queries (i.e. asking whether a given conjecture regular language L is precisely U). If an equivalence query is answered in the negative, the teacher also provides a counter-example, i.e. a string which is either in L and not in U, or which is in U but not in L.

The number of queries the L^* algorithm makes is dependent on the size of the interface automaton it constructs. We answer the queries of the L^* algorithm using calls to a standard symbolic model-checker, and thus we do not resort to a subset construction. Hence, the learning-based solution avoids both the problems stated above: the complexity of the algorithm depends on the size of the interface constructed, and the algorithm is implemented using symbolic model checking techniques.

In our setting, we split the role of the teacher answering equivalence queries into two: one that checks subset queries (whether a given conjecture language is a subset of U) and one that checks superset queries. As we show below, membership queries and subset queries can be efficiently handled using model-checking algorithms. However, we do not know how to handle the superset query efficiently. We can in fact show that while the computational complexity of membership queries and subset queries is in polynomial time (in fact in nondeterministic log-space), the superset query is NP-hard, and hence a simple algorithm to answer it is unlikely to exist.

We hence propose heuristic approximations to handle superset queries. We ensure that the equivalence query is first checked as a subset query, and only if the subset query passes, it is passed on as a superset query. The superset query teacher first checks for a certain stronger property, which if true, implies that the conjecture language L is indeed a superset of U, and the superset query can return *true*. But if this property fails, we gain no information as to whether L is a superset of U. At this point, if we declare that the language L is indeed U, the L^* algorithm will terminate with an interface that is a subset of U, and is not guaranteed to be exactly U.

However, in practice, this kind of termination results sometimes in interfaces that are too restrictive. Hence, we do an additional test that looks for certain simple counter-examples (strings in U that are not in L) that we do not want to miss. If this procedure fails to generate a counter-example, we terminate and output the interface.

In summary, our algorithm is guaranteed to terminate (since the L^* algorithm guarantees termination) and output an interface J, which is a minimal DFA that is guaranteed to be safe. Also, the algorithm either declares that J is the most-permissive interface (in which case it is guaranteed that J is the same as U) or the algorithm declares that J may not be the most-permissive interface. In the latter case, we are assured that J is maximal (this follows from the property of the L^* algorithm that it is the minimal automaton that satisfies the "observation table" it has maintained). Finally, the algorithm has built-in heuristics that search for a certain class of counter-examples that try to make the output interface closer to U.

We describe now formally the notion of automata representing interfaces, the learning algorithm and the handling of queries.

5.1 Interfaces

Let us start by defining formally how finite automata describe interfaces. We use deterministic finite automata over an alphabet of method-call–return-value pairs, whose language is prefix-closed, to capture interfaces. Given a boolean class $A = (M, X, \{D_x\}_{x\in X}, X_r, Init, \{T_m\}_{m\in M}, \{R_m\}_{m\in M})$, consider a deterministic finite automaton (DFA) $J = (Q, q_0, F, \delta)$ over the alphabet $(M \times S_r)$, where:

- M is the set of method names of A and S_r is the set of all valuations for return variables of A.
- Q is a finite set of states; $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of accepting states, and $\delta : Q \times (M \times S_r) \to Q$ is the transition function.

Intuitively, if $\delta(q, (m, r)) = q'$, it means that when at state q, the interface corresponding to the DFA can invoke the method m, and if it receives the return value r, it moves to state q'. Note that if an interface prohibits a sequence $\sigma \in (M \times S_r)^*$, then it prohibits all extensions of σ and the language of the interface is hence prefix-closed. We therefore require that there is at most one non-accepting state and it is a sink if it exists (i.e., all transitions from it go to itself).

For $q \in Q$, let the set of *legal methods* from q, denoted $LM(q)$ be the set of methods m such that for some return value r, $\delta(q, (m, r)) \in F$. The transition function δ generalizes to strings in $(M \times S_r)^*$ in a natural way: $\delta(q, \varepsilon) = q$, and $\delta(q, \sigma \cdot (m, r)) = \delta(\delta(q, \sigma), (m, r))$, for $\sigma \in (M \times S_r)^*$, $(m, r) \in M \times S_r$. Now, a DFA $J = (Q, q_0, F, \delta)$ represents the interface $I : (M \times S_r)^* \to 2^M$, given by $I(\sigma) = LM(\delta(q_0, \sigma))$, for every $\sigma \in (M \times S_r)^*$.

Given a boolean class A, an exception predicate E and an automaton J representing an interface, we say J is a *safe interface* for A with respect to E if J represents a safe interface I for A with respect to E. We henceforth refer to automata that represent interfaces also as interfaces and treat them synonymous with the interfaces they represent.

5.2 L* algorithm

The L^* algorithm learns an unknown regular language and generates a minimal DFA that accepts the regular language. This algorithm was introduced by Angluin [4], but we use an improved version by Rivest and Schapire [27].

The algorithm infers the structure of the DFA by asking a teacher, who knows the unknown language, two types of questions: *membership queries* and *equivalence queries*. On a membership query, the learner asks whether a string σ is

L^* **Algorithm**

```
1:  S := {ε}; E := {ε};
2:  foreach (s ∈ S), (a ∈ Σ) and (e ∈ E) {
3:      T[s, e] := Member(s·e);
4:      T[s·a, e] := Member(s·a·e);
5:  }
6:  repeat:
7:      while ((s_new := Closed(S, E, T)) ≠ null) {
8:          Add(S, s_new);
9:          foreach (a ∈ Σ) and (e ∈ E) {
10:             T[s_new·a, e] := Member(s_new·a·e);
11:         }
12:     }
13:     C := MakeConjectureMachine(S, E, T);
14:     if ((cex := Equivalent(C)) = null) then return C;
15:     else {
16:         e_new := FindSuffix(cex);
17:         Add(E, e_new);
18:         foreach (s ∈ S) and (a ∈ Σ) {
19:             T[s, e_new] := Member(s·e_new);
20:             T[s·a, e_new] := Member(s·a·e_new);
21:         }
22:     }
```

Figure 3: L^* algorithm

accepted by the unknown language, and the teacher answers *true* or *false*. On an equivalence query, the learner conjectures that the machine it has constructed is equivalent to the unknown language. The teacher replies that the conjecture is either correct or incorrect, and in the latter case gives a counter-example which is a string accepted by one but not the other.

Figure 3 illustrates the L^* algorithm. Let U be the unknown regular language and Σ be its alphabet. At any given time, the L^* algorithm has information about a finite collection of strings over Σ, classified either as members or non-members of U. This information is maintained in an *observation table* (S, E, T) where S and E are a set of strings over Σ, and T is a function from $(S \cup S \cdot \Sigma) \cdot E$ to $\{true, false\}$. Intuitively, S can be viewed as a set of representative strings that lead from the initial state (uniquely) to the various states of the DFA, and E as experiments that are performed at these states in order to distinguish states. T maps strings σ in $(S \cup S \cdot \Sigma) \cdot E$ to *true* if σ is in U, and to *false* otherwise. Initially, S and E are set to $\{\varepsilon\}$, and T, which is implemented as a two-dimensional array, is initialized using membership queries for every string in $(S \cup S \cdot \Sigma) \cdot E$ (line 2–5). In line 7, it checks whether the observation table is *closed*; that is, for every $s \in S$ and $a \in \Sigma$, there exists $s' \in S$ such that $T[s \cdot a, e] = T[s', e]$ for every $e \in E$. If not, each such $s \cdot a$ (e.g., s_{new} is $s \cdot a$ in line 8) is simply added to S. The algorithm again updates T with regard to $s \cdot a$ (line 9–11). Once the table is closed, it constructs a conjecture machine $C = (Q, q_0, F, \delta)$ as follows (line 13): $Q = S$, $q_0 = \varepsilon$, $F = \{s \in S \mid T[s, \varepsilon] = true\}$, and for every $s \in S$ and $a \in \Sigma$, $\delta(s, a) = s'$ such that $T[s \cdot a, e] = T[s', e]$ for every $e \in E$. Finally, if the answer of the equivalence query is yes, it returns the current machine C; otherwise, a counter-example $cex \in ((L(C) \setminus U) \cup (U \setminus L(C)))$ is provided by the teacher. The algorithm analyzes the counter-example *cex* in order to find the longest suffix e_{new} of *cex* that witnesses a difference between U and $L(C)$ (line 16). Adding e_{new} to E reflects the difference in the next conjecture by splitting a state in C. It then updates T with respect to e_{new}.

The L^* algorithm guarantees to construct a minimal DFA for the unknown regular language using only a polynomial number of membership and equivalence queries: more precisely with $O(|\Sigma|n^2 + n \log m)$ membership queries and at most $n - 1$ equivalence queries, where n is the number of states in the final DFA and m is the length of the longest counter-example provided by the teacher for equivalence queries.

5.3 Synthesis of interfaces using L*

In this section, we explain how we apply the L^* algorithm to synthesize a safe interface. Given a boolean class A and an exception predicate E, we can make the L^* algorithm construct the most permissive interface for A with respect to E by providing answers for membership and equivalence queries to the learner.

Figure 4 is a high-level pseudo-code for the teacher who answers queries in the L^* algorithm. We implement teachers for the membership query, the subset query and the superset query, using model-checking procedures for CTL (Computational Tree Logic) model checking [11].

5.3.1 Membership and subset queries

Given a string $\sigma = (m_0, r_0), \cdots, (m_n, r_n)$, the teacher for membership queries checks whether the string σ is in the language U (i.e., whether $\sigma \in U$). The membership query can be reformulated as a subset query, where we first construct a simple interface J_σ with $|\sigma| + 2$ states that accepts precisely the string σ and its prefixes. It is easy to see that the membership query for σ is equivalent to the subset query for J_σ.

Since we know that the language we are trying to learn is prefix-closed, we ensure that the L^* algorithm does not ask membership queries for σ if a prefix of σ has already been known to be not in the language. This reduces the number of membership queries [8].

Given an interface J, the subset query asks whether the language of J is a subset of the language U (i.e., whether $L(J) \subseteq U$). Since U consists precisely the set of all the safe method-call–return-value sequences, this question is equivalent to asking whether J is a safe interface for A.

The teacher for subset queries checks, using a standard CTL model checker, whether J is a safe interface for the class A with respect to E. The state space of the interaction between A and J, $A||J$, is defined as $S_{A||J} = \{t_A, t_J\} \times M \times S \times Q$. A state $s = (t_A, m, s_A, q) \in S_{A||J}$ means that at the state s, the class A has 'turn', the currently executing method is m, and the class A and the interface J are in states s_A and q, respectively. Transitions of $A||J$ are as follows.

- Initially, the interface has 'turn' and selects a method $m \in LM(q_0)$ to be executed. Then, it passes 'turn' to the class.
- When the class gets 'turn' (let $s = (t_A, m, s_A, q)$ be the current state of $A||J$), the method m is simulated. The class keeps the 'turn' until it reaches a state $s'_A \in S$ with $s'_A \models R_m$, and then passes the 'turn' to the interface with a return value $r = s'_A[X_r]$.
- If the interface receives 'turn' with a return value r from the class (let $s = (t_J, m, s_A, q)$ be the current

```
Boolean Member(String σ) {
  J_σ := ConstructInterface(σ);
  if (Subset(J_σ) = null) then return true;
  else return false;
}

String Equivalent(Interface J) {
  if ((cex := Subset(J)) = null) then cex := Superset(J);
  return cex;
}

String Subset(Interface J) {
  A := ReadAbstractClass();
  φ := AG (E = false);
  cex := CTLModelChecking(A||J, φ);
  return cex;
}

String Superset(Interface J) {
  A := ReadAbstractClass();
  ψ := AG((legal=0 ∧ turn=t_J) → EF E);
  if ((cex := CTLModelChecking(A|||J, ψ)) = null) then {
    print ("L(J) is a superset of U.");
  } else if (¬Member(cex)) then {
    if ((cex := OneStepFurther(J)) = null) then
      print ("L(J) may or may not be a superset.");
  }
  return cex;
}

String OneStepFurther(Interface J) {
  foreach (q ∈ Q) and (m, r), (m', r') ∈ (M × S_r) {
    if ¬Accept(J, σ_q(m, r)(m', r')) then
      if (Member(σ_q(m, r)(m', r'))) then
        return σ_q(m, r)(m', r');
  }
  return null;
}
```

Figure 4: Implementation of the L^* teacher

state of $A||J$), it changes its state to $q' = \delta(q, (m, r))$ and picks a new method $m' \in LM(q')$. It then passes 'turn' to the class and the interaction continues as before.

A model checker checks (exhaustively) whether all the method-call–return-value sequences allowed in J keep the class A away from states that satisfy E by checking whether $A||J$ satisfies the CTL specification

$$\texttt{AG}\ (\neg E).$$

If it does, then J is a safe interface and the answer to the subset query is *true*. Otherwise, the model checker gives a counter-example path of $A||J$ that reaches a state where $E = true$. The method-call–return-value sequence extracted from this counter-example is provided to the learner as a counter-example string $\sigma \in L(J) \setminus U$.

5.3.2 Superset query

Given a conjecture interface J, the superset query asks whether the language of J is a superset of the language U (i.e., whether $L(J) \supseteq U$).

The teacher for superset queries must check that $\forall \sigma \in (M \times S_r)^*$, $\sigma \notin L(J) \rightarrow \sigma \notin U$; that is, for every method-call–return-value sequence $\sigma \notin L(J)$, there is *some* run of the boolean class A corresponding to the sequence σ that does not stay within the safe set ($E(X) = false$). We do not know how to implement this exactly and efficiently.

Let us consider the computational complexity of the membership, subset and superset queries to see why we think superset queries are inherently hard. Given a finite class A, let the size of A be the sum of its methods, states, and transitions (i.e., the size of A is the the size of the class when it is represented *explicitly*). Assume the safety predicate E is given as a subset of the states of the class. Now, given a conjecture strategy automaton J, the problem of checking whether $A||J$ is safe can be done in polynomial time (in fact in nondeterministic log-space). Similarly, membership queries can be handled in polynomial time. However, it turns out that superset queries are NP-hard:

PROPOSITION 1. *Given an (explicit) class A, an unsafe set E and a strategy automaton J, checking whether there is some string $\sigma \notin L(J)$ such that $L(J) \cup \{\sigma\}$ is safe, is* NP*-hard.*

The proof of the above is by a reduction from 3-SAT, and crucially uses the fact that the game is a partial information game; we omit the proof.

We hence turn to ways that heuristically and approximately answer superset queries. Our first step is to pose a stronger property ψ that asks whether for every method-call–return-value sequence $\sigma \notin L(J)$, *all* runs of the boolean class A corresponding to the sequence σ do not stay within the safe set ($E(X) = false$). Note that if the property ψ is true, then $L(J) \supseteq U$; otherwise, we cannot conclude whether $L(J) \supseteq U$ or not.

To check the property ψ, we define a new interaction between A and J, $A|||J$, that simulates legal method sequences in J followed by at most one method not allowed by J. The domain of $A|||J$ is $S_{A|||J} = \{t_A, t_J\} \times M \times S \times Q \times \{0, 1\}$ which adds a *legal* bit to $S_{A||J}$. Transitions of $A|||J$ are as follows.

- Initially, the interface has 'turn' and selects a method $m \in M$. If $m \in LM(q_0)$, it sets *legal* to 1, otherwise to 0. Then, it passes 'turn' to the class.
- When the class receives 'turn', the method m is simulated in the same manner as in $A||J$; however, if $legal = 0$, then at the end of the method the model halts and does not return to the interface.
- If the interface gets back 'turn' with a return value r, let $s = (t_J, m, s_A, q)$ be the current state of $A|||J$ (*legal* must be 1). Then, it moves from q to $q' = \delta(q, (m, r))$, picks a new method $m' \in M$, sets *legal* to 0 iff $m' \notin LM(q')$, and passes 'turn' to the class.

A model checker checks whether the first method call not allowed by the interface J always leads to an unsafe state during its execution, by checking the following CTL specification

$$\texttt{AG}\ ((legal = 0 \wedge turn = t_J) \rightarrow \texttt{EF}\ E).$$

Note that the above formula captures an inherently branching-time property and cannot be captured using linear temporal logic nor can be checked using a simple invariant checker.

If $A|||J$ satisfies the above specification, the teacher answers true for the superset query and the L^* algorithm terminates with an interface and reports that it is the most permissive safe interface. Otherwise, the model checker provides a counter-example which is a method-call–return-value sequence σ. By definition, there is at least one run corresponding to σ that is safe, but we do not know whether *all* runs corresponding to σ are safe; hence we cannot return σ as a counter-example to the superset query, as σ may not be in U. We now check whether σ is indeed in U by a membership query on σ. If σ is in U, then σ is a witness for $L(J) \not\supseteq U$ and is returned to the L^* algorithm to update the conjecture J. Otherwise, σ is not in U and we discard it.

If the above method fails to produce a counter-example, we turn to some heuristic ways to check for natural counter-examples that the interface may have missed. By a property of the observation table, a conjecture interface always allows all the safe method calls from every state since the L^* algorithm checks membership for every string $\sigma \in S \cdot \Sigma$ (more precisely, $\sigma \in (S \cup S{\cdot}\Sigma) \cdot E$) and allows calling a method m from a state s if $s{\cdot}m$ is a member. However, in the examples we have experimented with, there are often scenarios where a safe method call sequence $m{\cdot}m'$ is disallowed from a particular state q even though calling m guarantees that calling m' is safe. The reason is this: if q' is the state reached from q on m, then there can be another path to q' that makes calling m' from q' unsafe.

To find these counter-examples for superset queries (in the above case, calling $m{\cdot}m'$ at q), we check whether from every state the conjecture interface allows all the safe method call sequences of length 2 (procedure *OneStepFurther*). Formally, we denote one representative of the strings that reach q as σ_q (this representative is the one in the set S, in the L^* algorithm). We check whether $\exists q \in Q, (m,r), (m',r') \in M \times S_r$. $\sigma_q(m,r)(m',r') \notin L(J) \wedge \sigma_q(m,r)(m',r') \in U$ by traversing J and asking membership queries. If there exists $\sigma_q(m,r)(m',r')$ satisfying the above, we use $\sigma_q(m,r)(m',r')$ as a counter-example for the superset query. Otherwise, our synthesis terminates with the maximal safe interface J (maximality of J is assured by the L^* algorithm).

In summary, the teacher for superset queries always provides a counter-example to the learner, or terminates and says *true*, and additionally may give the assurance that the superset query indeed was true. If it terminates with this assurance, then the L^* algorithm has learned the most permissive interface. If not, the L^* algorithm has learned a safe interface that is guaranteed to be maximal.

6. EXPERIMENTS

In this section, we report results obtained by using JIST on some sample Java classes. In all cases, the tool could generate a useful and safe interface automatically using little computational resource. The original Java classes, intermediate forms (such as the abstracted Jimple files) and results are available at `http://www.cis.upenn.edu/jist/` .

ListItr Class

We present the class `ListItr` as the first example. It is an inner class of `AbstractList` from the package `java.util` and supports random access for a list. It has ten methods. We focus on the following five: `next(), remove(), previous(), set() and add()`.

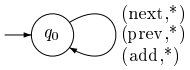

Figure 5: List Iterator (1 predicate)

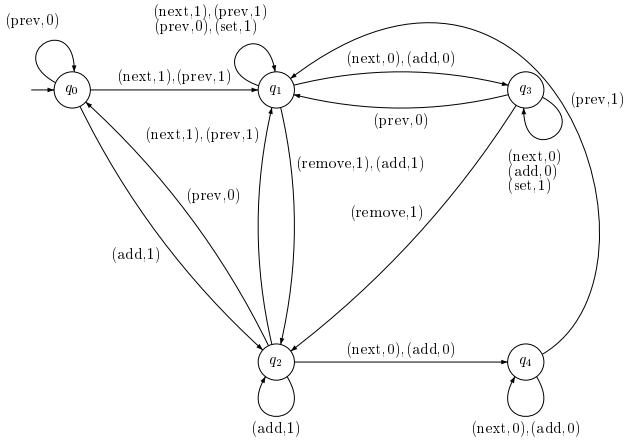

Figure 6: ListItr (2 predicates)

The class is used to navigate and modify a list. Methods `next` and `previous` are used to traverse the list, `add` to add a new element, `set` and `remove` to modify/remove the last accessed element. The last accessed element is tracked using an index into the list (called `lastRet`) and is updated by the `next` and `previous` methods. However, if `remove` or `add` are called, this variable is set to -1. If any of the methods `remove` or `set` is invoked when `lastRet` is -1, an exception is raised. The exception predicate we use corresponds to raising of this exception.

The interface shown in Figure 5 is a safe interface. However, it is overly restrictive. While it allows calls to `next(), previous()` and `add()` methods, it completely disallows calls to `remove()` and `set()`. The interface was produced using only one predicate, $lastRet = -1$, i.e. the predicate that triggers the raising of the *IllegalStateException* exception. The value of `lastRet` is the index of element returned by most recent call to next or previous. It is reset to -1 if this element is deleted by a call to remove.

In order to produce a more permissive interface, we add one more predicate, $cursor = -1$, the predicate that is used as a test in the `if` statements in several of the methods. The value of `cursor` is the index of element to be returned by subsequent call to next. We need to keep track of whether or not it is equal to -1, because the code contains the assignment $lastRet = cursor$.

The resulting interface is shown in Figure 6. The high level description of this interface is that the methods `next(), previous()` must be called successfully (return 1) before `remove` or `set` can be called. Furthermore, notice that:

- When a method is not allowed by the interface at a given state (i.e. there are no outgoing edges labeled with the method's name), then it is not safe to call this method (because an exception might be raised). For example, the method `set` is not allowed at q_0. As noted before, only the final states of the automata are shown.

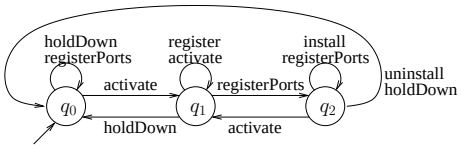

Figure 7: ServerTableEntry

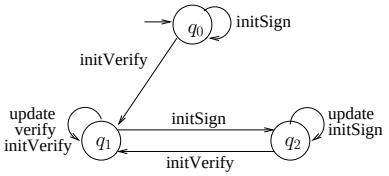

Figure 8: Signature (programming error)

- if there is an outgoing edge from a given state q for a method m but the interface does not mention all possible return values m from q, it means that the return values that are missing are not feasible. For example, the method `next` can not return 0 at q_0.

We also present a case when the ListItr class is abstracted using an additional irrelevant predicate, $expectedModCount = 0$ (we checked manually that it was irrelevant). In this case, the resulting interface is the same as the one we got without it. This suggests that the tool is resilient to addition of irrelevant predicates, and we believe that an automatic tool for refinement of predicates will not adversely change the quality of results of our tool.

As mentioned previously, an interface for this class was computed by the tool presented in [32]. Their interface however is not sound. For example, their interface declares that one can call `remove` after a call to `prev`; however, there are situations where `prev` may fail (say when one is at the beginning of the list) in which case `remove` is not allowed. On the other hand, our interface is guaranteed to be safe: for example, in Figure 6, if `prev` fails (i.e. after (`prev`, 0)), `remove` is disallowed. Our interface also gives more precise information about possible return values: for example, Figure 6 says that calling `prev` initially will always fail (i.e. return 0).

ServerTableEntry Class

`ServerTableEntry` from the package `com.sun.corba.se.internal.Activation` is a class related to the server table for Corba objects. It has 19 methods. We choose the following six: `activate()`, `register()`, `registerPorts()`, `install()`, `uninstall()`, `holdDown()` to construct the interface. Once users activate the system by `activate()`, they can enroll a server by `register()` and listeners' network ports by `registerPorts()`. After ports are registered, the user can install the server or uninstall it. Users can call `holdDown` whenever they want. We produced the interface for the class using five predicates, $state = 0$, $state = 1$, $state = 2$, $state = 3$ and $state = 4$.

The resulting interface is illustrated in Figure 7. Once `activate()` is called, `register` or `registerPorts` can be called at q_1. After the registration, `install` or `uninstall` are enabled at q_2. `holdDown()` can be called at any state, and results in returning to the initial state q_0.

Signature Class

The class `Signature` is taken form the package `java.security`. This class was described in Section 2. To produce the interface for this class, we used three predicates, $state = UNINITIALIZED$, $state = SIGN$ and $state = VERIFY$. The resulting interface is shown in Figure 1.

In order to demonstrate how such an interface can help a programmer to discover errors in programs, we consider the `Signature` class with an artificially introduced error. We modify the `initSign` method so that in some cases initialization may fail, thus not making it possible to guarantee that the call to `Sign` is possible after a call to `initSign`. The interface produced in this case is in the Figure 8 and the error is clearly visible: it is not possible to call the `Sign` method.

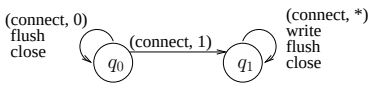

Figure 9: PipedOutputStream Class

PipedOutputStream Class

The class `PipedOutputStream` from the package `java.io` is an implementation of an abstract class `OutputStream`. It has four methods, `connect`, `write`, `flush`, `close`. One property that the user of the class should know is that it is necessary to call `connect` before calling `write`.

This property is captured by the interface produced by the JIST tool. The predicate $sink = null$ was used in constructing the interface, because it is the predicate guarding the `NullPointerException`. Note that while creating this interface, the heuristic check for the natural counter-examples for the superset query was used. (Note: The prototype implementation of our tool currently handles only predicates of the form "var=integer constant"; so, the abstraction step had to be done manually for this example. However, there is no difference conceptually between the predicates the tool handles and the predicate $sink = null$.)

Experiments Summary

All experiments were performed on a PC using a 1GHz Pentium III processor, 1.5GB memory and a Linux operating system. The synthesis results for three classes are shown Table 6. It gives the number of predicates used for abstraction in the Java program, the number of derived predicates in the corresponding Jimple code (Jimple introduces temporary variables corresponding to Java variables) and the number of variables in the NuSMV boolean model (this model has additional boolean variables for handling the control flow, like the program-counter). The table also shows the number of membership, subset and superset queries that the learner asked in the interface synthesis phase, whether the heuristic method (*OneStepFurther*) found a counter-example for the superset query and the total execution time in seconds. The last column indicates whether it was possible for the tool to conclude that the resulting interface is the most permissive one. Though the tool could not conclude this in two cases, it turns out that even in these cases the interfaces generated were indeed the most permissive ones.

Class name	Predicates in Java	Predicates in Jimple	Var	MQ	SubQ	SuperQ	OneStepFurther found a cex?	Time (sec)	Tool reports interface is most permissive?
Signature	3	7	24	83	3	3	N/A	10.3	yes
ServerTableEntry	5	9	25	53	3	3	N/A	9.2	yes
ListItr (1 pred.)	1	5	20	35	1	1	no	5.2	no
ListItr (2 pred.)	2	17	29	288	5	2	N/A	83.4	yes
ListItr (3 pred.)	3	19	31	288	5	2	N/A	101.8	yes
PipedOutputStream	1	5	19	64	2	2	yes	6.0	no

Table 1: Experimental results

7. CONCLUSIONS

We have proposed a technique for automatically synthesizing behavioral interface specifications for Java classes using abstraction and games. Our initial prototype and experimentation shows promising results. The proposed solution to computing the most permissive winning strategies in partial information games using learning algorithms and symbolic model checking can be useful in contexts other than interface synthesis. There are many directions for future research. First, the current tool handles a simple subset of Java. We would like to develop similar techniques for extracting interfaces for a set of classes that call one another, and for handling class and exception hierarchies. Second, the current tool allows only simple forms of predicates for abstraction. The abstractor can be made much more powerful without sacrificing automation. Coupled with slicing techniques, this will lead to a robust toolkit that will permit extensive experimentation. Finally, software verification tools can address scalability using our tool for automatically abstracting classes by their interfaces for compositional verification, and this application deserves further exploration.

Acknowledgments

We thank Gunjan Gupta and Anshuman Srivastava for their help in implementing components of the JIST prototype. This research was partially supported by NSF award CCR-0306382 and ARO URI award DAAD19-01-1-0473.

8. REFERENCES

[1] S. Abramsky. Semantics of interaction. Technical report, Oxford University, 2002.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):1–42, 2002.

[3] G. Ammons, R. Bodík, and J. Larus. Mining specifications. In *Proc. 29th ACM POPL*, pages 4–16, 2002.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[5] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. PLDI*, ACM SIGPLAN Notices 36(5), pages 203–213, 2001.

[6] T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th ACM POPL*, pages 1–3, 2002.

[7] H. Barringer, C.S. Pasareanu and D. Giannakopolou. Proof rules for automated compositional verification through learning. In *Proc. of the 2nd Int'l Workshop on Specification and Verification of Component Based Systems*, 2003.

[8] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin's learning. In *Proc. of International Workshop on Software Verification and Validation*, 2003.

[9] A. Chakrabarti, L. de Alfaro, T.A. Henzinger, M. Jurdzinski, and F. Mang. Interface compatibility checking for software modules. In *Proc. 14th CAV*, LNCS 2404, Springer, pages 428–441, 2002.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An Opensource tool for symbolic model checking. In *Proc. 14th CAV*, LNCS 2404, Springer, pages 359–364, 2002.

[11] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. of Workshop on Logic of Programs*, pages 52–71, 1981.

[12] J.M. Cobleigh, D. Giannakopoulou, and C.S. Pasareanu. Learning assumptions for compositional verification. In *Proc. 9th TACAS*, LNCS 2619, Springer, pages 331–346, 2003.

[13] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd ICSE*, pages 439–448, 2000.

[14] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM POPL*, pages 238–252, 1977.

[15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM POPL*, pages 84–96, 1978.

[16] L. de Alfaro and T.A. Henzinger. Interface automata. In *Proc. 9th ACM FSE*, pages 109–120, 2001.

[17] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM POPL*, pages 59–69, 2001.

[18] D. Giannakopoulou, C.S. Pasareanu and H. Barringer. Assumption generation for software component verification. In *Proc. 17th ASE*, pages 3–12, 2002.

[19] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th CAV*, LNCS 1254, pages 72–83, 1997.

[20] T.A. Henzinger, R. Jhala, R. Majumdar, K.L. McMillan. Abstractions from proofs. In *Proc. 31st ACM POPL*, pages 232–244, 2004.

[21] T.A. Henzinger, R. Jhala, R. Majumdar, G. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proc. of CAV*, LNCS 2404, pages 526–538. Springer, 2002.

[22] G. Holzmann and M. Smith. Software model checking - extracting verification models from source code. In *Formal Methods for Protocol Engineering and Distributed Systems*, pages 481–497, 1999.

[23] F. Logozzo. Automatic inference of class invariants. In *Proc. of VMCAI*, LNCS 2937, pages 211–222, 2004.

[24] J. Nimmer and M. Ernst. Automatic generation of program specification. In *Proc. of ISSTA*, pages 229–239, 2002.

[25] G. Ramalingam, A. Warshavsky, J. Field, D. Goyal and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *ACM PLDI*, pages 83–94, 2002.

[26] J.H. Reif. Universal games of incomplete information. In *Proc. of the 11th ACM symposium on Theory of computing*, pages 288–308. ACM Press, 1979.

[27] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[28] W. Thomas. Infinite games and verification. In *Proc. 14th CAV*, LNCS 2404, pages 58–64. Springer, 2002.

[29] O. Tkachuk, M.B. Dwyer, and C.S. Pasareanu. Automated environment generation for software model checking. In *Proc. 18th ASE*, pages 116–127, 2003.

[30] R. Vallée-Rai, L. Hendren, V. Sundaresan, E.G. Patrick Lam, and P. Co. Soot - a Java optimization framework. In *Proc. CASCON*, pages 125–135, 1999.

[31] D. Walker. A type system for expressive security policies. In *Proc. 27th ACM POPL*, pages 254–267, 2000.

[32] J. Whaley, M.C. Martin, and M.S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. ISSTA*, pages 218–228, 2002.