Mining Library Specifications using Inductive Logic Programming.

Sriram Sankaranarayanan
NEC Laboratories America.
srirams@nec-labs.com

Franjo Ivančić
NEC Laboratories America.
ivancic@nec-labs.com

Aarti Gupta
NEC Laboratories America.
agupta@nec-labs.com

ABSTRACT

Software libraries organize useful functionalities in order to promote modularity and code reuse. A typical library is used by client programs through an application programming interface (API) that hides its internals from the client. Typically, the rules governing the correct usage of the API are documented informally. In many cases, libraries may have complex API usage rules and unclear documentation. As a result, the behaviour of the library under some corner cases may not be well understood by the programmer. Formal specifications provide a precise understanding of the API behaviour.

We propose a methodology for learning interface specifications using Inductive Logic Programming (ILP). Our technique runs several unit tests on the library in order to generate relations describing the operation of the library. The data collected from these tests are used by an inductive learner to obtain rich Datalog/Prolog specifications. Such specifications capture essential properties of interest to the user. They may be used for applications such as reverse engineering the library internals or constructing checks on the application code to enforce proper API usage along with other properties of interest.

Categories and Subject Descriptors: D.2.4 [Verification]: Statistical Methods, I.2.6 [Learning]: Induction.

General Terms: Verification, Theory.

Keywords: Software Specification, Verification, Datalog, Inductive Logic Programming, Machine Learning.

1. INTRODUCTION

Software libraries promote modularity and code re-use. Common examples include operating system functionalities such as *threads, files and sockets*; data structures such as *stacks, queues, lists and hashtables*; utilities for *multimedia codecs, data compression, encryption* and *transmission*; common functionalities such as *parsing* and so on. The functionalities provided by a library are exported through an *Application Programming Interface* (API).

An API consists of object types and function signatures. The interface usage rules are a combination of two basic rule categories: (a) pre-conditions on the arguments of function calls, and (b) admissible sequence of function calls to the API. Any violation of the API usage norm can lead to faulty or unpredictable behaviour in programs.

In theory, the restrictions on the calls to an API can be formally specified using many types of logical formalisms including Prolog/Datalog [13, 4] and automata-based formalisms such as *interface automata* [9]. In practice, however, specifications of API behaviour and restrictions on its usage are stated informally in a natural language. Such specifications may be ambiguous or even inaccurate. Often, informal specifications may not specify all the corner cases.

Existing verification tools such as SLAM [1], JPF [2] and F-Soft [3] can automatically check application code for conformance to formal specifications. These tools mostly rely on the user to provide specifications for the library APIs used by the application. The process of manually writing specifications is cumbersome and prone to errors. Therefore, it is desirable to have automatic techniques to learn such specifications. There are numerous approaches to learning specifications. First of all, the nature of the *specification* itself varies widely. Some techniques learn data preconditions for function call parameters expressed in a suitable constraint language. Other techniques derive automata-based temporal characterizations of legal/illegal calling sequences.

Specification inference may be *static* or *dynamic*. Static approaches analyze the source code to extract specifications while dynamic approaches infer the required behaviour by analyzing the runtime behaviour. Similarly, approaches may be classified as *direct* or *indirect*. A *direct approach* infers specifications directly by analyzing the library. An *indirect approach* analyzes numerous client applications that use the library to derive common usage patterns for the API.

In this paper, we present a direct technique to automatically infer declarative specifications of the API behaviour.

[1]Cf. `http://research.microsoft.com/slam`
[2]Cf. `http://javapathfinder.sourceforge.net`
[3]Cf. `http://www.nec-labs.com/~fsoft`

ICSE'08, May 10–18, 2008, Leipzig, Germany.

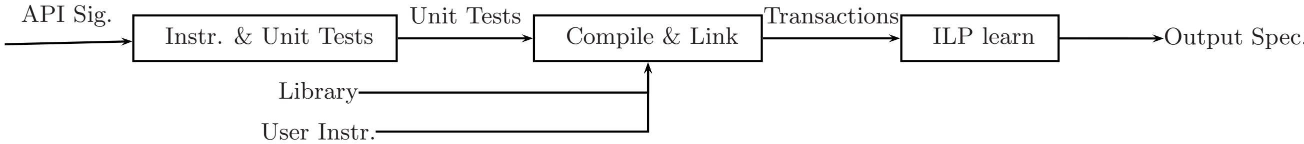

Figure 1: Block diagram for Inductive Learning of Specifications.

Our approach formulates numerous unit tests, while describing these tests and their outputs using some inbuilt relations. Our tool subsequently mines these relations to infer declarative specifications for *target concepts* such as the raising of an exception, the return of a special value by a function, the printing of some specific pattern onto the output, and other types of targets of interest to the user. The output of our tool consists of the declarative specifications that describe the target concept in terms of the operations performed upon the library and its responses.

One of the key requirements for our approach is the availability of many unit tests that need to be run on the library code. In most cases, the required unit tests are generated automatically using a randomized unit testing approach, based on an *interface description*. An interface description consists of the standard API header file with additional type qualifiers and a description of the target concept to be learned.

The primary advantage of our approach is that it can learn specifications for different types of target properties as desired by the user. Secondly, the source code for the library is not required. Hence, it is especially suited for reverse engineering applications. Furthermore, the required annotations for the header files often require very little user knowledge about the library's internals. Finally, by running tests and observing the output, all the behaviours considered are over actual program executions. This is in contrast to static approaches that depend on behavioural abstractions.

The key learning tool used by our approach is called *Inductive Logic Programming*(ILP) [14]. ILP tools learn Datalog (Prolog) specifications for a given target relation (predicate) p in terms of relations $p_1, \ldots, p_m$. The input data to an ILP problem consists of tuples that belong to each of the relations $p_1, \ldots, p_m$ as well as positive and negative examples of the target relation p. ILP tools can also use some user provided background information about the predicates along with restrictions on the desired form of the specifications to infer a set of Horn clause rules that explain the target in terms of the data. The specifications learned are quite expressive; they permit constraints on arguments of function calls as well as characterizations of calling sequences.

Figure 1 shows the basic components in our approach. Given an API signature describing the types and methods involved in the API along with some information about the method side effects, we automatically generate random unit tests that exercise some of the functions in the API. The tests are also instrumented to observe the system's response to the functions and print a log of the inputs/outputs. This log is called a *transaction*. The unit tests generated are compiled and linked with some additional instrumentation for the target concept. After execution, the resulting transactions are converted into relations and fed to an inductive logic programming (ILP) tool that learns the specifications. These specifications express the target concept in terms of the transactions observed in the unit test.

Figure 2: Function to empty and free stack contents.

```
void freeTheStack (stack_t * stack) {
    while( (x=pop(stack)) != NULL) freeElt(x);
}
```

The ILP engine is used in our approach as a black box. Nevertheless, it is possible to streamline the specification learning by providing background knowledge in the form of structural annotations that restrict the type of specifications considered, along with user defined background predicates that are not inbuilt into the ILP tool.

Motivating Example

Consider the stack data structure with an API as shown in Figure 3. The interface can be implemented in numerous ways with behavioural differences arising only for the corner cases. Consider the call to the "pop" method on an empty stack. Implementation A raises an exception for this case, requiring the user to check the stack length before calling pop. On the other hand, another implementation B returns a NULL pointer upon popping an empty stack. It is therefore conceivable that an user of library B writes the function shown in Figure 2 to empty out and free the contents of a given stack.

Figure 3 also shows a specification for each implementation written in a declarative formalism. The specification for library A predicts the raising of an exception on pop while that for B predicts the return of a NULL pointer. The specification for B indicates that the pop operation may also return NULL if a NULL pointer has been pushed onto the stack by an earlier operation. Therefore, the function shown in Figure 2 need not free the stack completely, and can lead to memory leaks. While the example may seem contrived, incomplete specification of libraries can lead to serious defects in application software.

2. RELATED WORK

Software Specification Mining. The automatic inference of API specifications for software libraries has been studied under a wide variety of settings, distinguished primarily by the type of specification learned and the techniques used to learn them.

Cohen uses ILP to learn Datalog specifications of different inbuilt views of a relational database system inside a telephony switch system by observing the results of its execution [7]. While the broad idea of discovering software specifications using ILP originates in that work, the work itself addresses the specific case-study consisting of the database system. However, the current work presents a more principled approach to learning specifications by describing unit

```
/* create a stack with some capacity */
stack_t * createStack (int capacity);
/* stack length */
int length(const stack_t * stack);
/* stack capacity */
int capacity(const stack_t * stack);
/*top of the stack*/
void* top (const stack_t * stack);
/*pop an element from the stack*/
void* pop (stack_t * stack);
/*push an element onto the stack*/
void push (stack_t * stack, void * elem);
/*free the stack*/
void free (stack_t * stack);
```

Implementation A:

```
/* pop throws an exception if the stack length is zero. */
throwsException("pop",s) :- length(s,0).
/* push throws an exception if the length is equal to the capacity. */
throwsException("push",s):- length(s,L), capacity(s,L).
```

Implementation B:

```
/* pop returns NULL if the stack length is zero. */
returnsNULL("pop",s) :- length(s,0).
/* pop returns NULL if a NULL is on the top of the stack. */
returnsNULL("pop",s) :- queryResult("top",s,NULL).
```

Figure 3: An API for stack implementations along with the behaviour of two different implementations for the push/pop operations.

tests relationally and using ILP to recover specifications for a target concept.

Ammons et al. mine specifications by running applications on the library and observing their interactions with the API of the library [3]. The specifications are mined using an automata learning algorithm that learns weighted finite state automata. Jiang et al. [12] propose an automata learning scheme based on NGRAMS to learn function call sequences by observing execution traces. The *Perracotta* tool due to Yang et al. [23] infers statistically common function call sequences by dynamically observing the application code. The static analysis tool ESP is used to check for violations of the usage patterns mined by the Perracotta tool [23].

Whaley et al. present a static analysis for inferring pairs of methods in a class that may raise an exception when called successively [22]. The statically obtained specification is compared against a dynamically obtained specification obtained by instrumenting and executing the code. The comparison yields useful information about bugs, code coverage and so on.

Ramanathan et al. perform an automatic static analysis of the application code to mine usage patterns for a given API [18]. Their technique computes the intersection of many abstract call patterns mined at different call sites. Acharya et al. infer partial orders among different functions in the API by static analysis of the application code [1]. They infer commonalities among the different partial orders and output them as possible API usage rules.

Our approach learns specifications for target concepts such as exceptions, special return values, the printing of some message to the output and so on. In contrast, indirect approaches can only claim to infer "common" usage patterns exercised by the client. It is not possible, in these approaches, to exercise usage patterns that are not exercised by the client in the first place. Therefore, specifications for functions used infrequently in the application code cannot be reliably inferred. In practice, functions which are underspecified tend to be used relatively infrequently.

Alur et al. [2] use predicate abstraction and model checking directly on the library itself to synthesize specifications that characterize correct calling sequences for a given API.

The recent work of Christodorescu et al. demonstrates the use of specification mining to automatically infer patterns describing malicious behaviour inside applications [6]. The Daikon tool of Ernst et al. discovers *likely invariants* directly from execution traces [10]. Learning invariants from runtime data is fundamentally different from specification learning for a target concept. Whereas, specifications are learned in the presence of negative as well as positive examples of the target concept, instances for likely invariants contain only positive examples of reachable states.

Randomized Unit Testing. Randomization is frequently used in lieu of exhaustive exploration. *Random testing* samples uniformly at random from the space of inputs, essentially treating the program as a black-box. The JCrasher tool due to Csallner et al. presents a randomized technique for generation of unit tests by exploring the *parameter graph* derived by analyzing the type signature of the API [8]. This technique is directly used in our approach to generate unit tests that allocate objects of various types and exercise a series of API methods on them.

Simplicity and ease of implementation are the essential advantages to random testing. However, certain inputs of interest may have a low probability of being discovered by chance alone. This can be alleviated using *concolic testing* to explore the program paths systematically using a constraint solver [11, 5, 19]. Concolic testing can enhance our testing process by exploring inputs that exercise different paths within a randomly chosen unit test.

Inductive Logic Programming. Inductive Logic Programming is a commonly used relational data-mining technique that seeks to infer Prolog/Datalog programs from relational data. An excellent description of the theory behind ILP and its applications is available from standard machine learning textbooks [14]. ILP learning algorithms generally perform a search on the space of permissible Horn clauses to discover rules that cover all the positive examples and none of the negative examples. However, in the presence of contradictions in the data due to noise, the learned clauses may cover some of the negative examples. Other approaches to the problem have used inverted resolution operators to induce clauses from the given data. ILP tools such as FOIL [17],FOCL [16], PROGOL [15], and ALEPH [20] are available online. We have used the ALEPH system extensively in our experiments and found it adequate for learning API specifications.

3. DECLARATIVE SPECIFICATIONS

We now present the basics of declarative specifications expressed in the *Datalog* formalism.

Datalog. Consider a signature Σ consisting of constants $c_0, \ldots, c_k$ and predicates $p_1, \ldots, p_j$. Let $x_1, \ldots, x_n$ be a set of typed variables over domains $D_1, \ldots, D_m$. Let $p(x_1, \ldots, x_n)$ be a distinguished goal (target) predicate that we wish to characterize in terms of the other predicates $p_1, \ldots, p_j$. A declarative specification consists of a set of *Horn clauses* $\{C_1, \ldots, C_n\}$, wherein each clause C_i is of the form

$$C_i : \; p_{i_1}(t_{11}, \ldots, t_{m1}) \; \wedge \; \cdots \; \wedge p_{i_m}(\cdots) \; \Rightarrow \; p(x_1, \ldots, x_k) \,,$$

where each t_{ij} denotes a variable $x_1, \ldots, x_n$, or a constant $c \in \Sigma$, while $\Rightarrow$ is used to denote implication. Following Prolog convention, such a clause is written as

$$p(x_1, \ldots, x_n) : - p_{i_1}(\cdots), p_{i_2}(\cdots), \ldots, p_{i_m}(\cdots) \,.$$

It is possible to extend the definition of a clause to consider recursion where p itself may be part of the left hand side of the implication. These clauses are well-defined provided the negation operator is not used in front of recursive predicates. The signature can be extended using function symbols, to yield the full power of Prolog specifications.

Example 3.1. *Consider the domain of human population along with the standard notion of biological father and mother. We assume predicates such as* $\mathsf{mother}(m, p)$ *and* $\mathsf{father}(f, p)$*, denoting that* m (f) *is the biological mother (father) of* p*. It is possible to characterize the familiar concept of "grandfather" using the following Datalog specification:*

$$\begin{array}{lcl} \mathsf{grandfather}(x, y) & :- & \mathsf{father}(x, z), \mathsf{mother}(z, y). \\ \mathsf{grandfather}(x, y) & :- & \mathsf{father}(x, z), \mathsf{father}(z, y). \end{array}$$

Similarly, the concept of an ancestor can be expressed:

$$\begin{array}{lcl} \mathsf{ancestor}(x, y) & :- & \mathsf{father}(x, y). \\ \mathsf{ancestor}(x, y) & :- & \mathsf{mother}(x, y). \\ \mathsf{ancestor}(x, y) & :- & \mathsf{ancestor}(x, z), \mathsf{ancestor}(z, y). \end{array}$$

Declarative specifications can capture useful behavioural properties of APIs such as permissible calling sequences, function call preconditions, special return values, outputs to a file and so on. In order to do so, we fix a first order language that describes functions, the objects that they manipulate and their return values, so that meaningful specifications can be expressed (later induced).

3.1 Interfaces

Let $T = \{t_1, \ldots, t_m\}$ be a set of types which include basic types such as int and char, along with *compound types* constructed by aggregating basic types into structures and arrays. We assume for the time being that the composition of compound types is unknown to us.

Def. 3.1 (Function Signature). *A function signature* f *is of the form* $(t'_1, \ldots, t'_n) \leftarrow f(t_1, \ldots, t_m)$*, wherein* $t_1, \ldots, t_m$ *denote the argument types and* $(t'_1, \ldots, t'_n)$*, the return types.*

With each function f, we also associate a set of argument indices that are destroyed as a result of the call. Destroyed arguments are denoted using a "$-$" superscript (see Ex. 3.2).

Example 3.2. *The function signatures for the functions in the stack API shown in Figure 3 are as follows:*

$$\begin{array}{rcl} \mathit{stack\_t*} & \leftarrow & \mathit{create\_stack}(\mathsf{int}) \\ \mathsf{int} & \leftarrow & \mathit{length}(\mathit{stack\_t*}) \\ \mathsf{int} & \leftarrow & \mathit{capacity}(\mathit{stack\_t*}) \\ \mathit{stack\_t*} & \leftarrow & \mathit{create\_stack}(\mathsf{int}) \\ \mathit{stack\_t*} & \leftarrow & \mathit{push\_stack}(\mathit{stack\_t*}^-, \mathit{elt\_t}) \\ (\mathit{stack\_t*}, \mathit{elt\_t}) & \leftarrow & \mathit{pop\_stack}(\mathit{stack\_t*}^-) \\ \mathit{void} & \leftarrow & \mathit{free}(\mathit{stack\_t*}^-) \end{array}$$

Note that the functions push_stack and pop_stack can modify their stack arguments. This is modelled by destroying the original stack argument and returning a modified stack.

In general, we model all the inputs (formal arguments, static variables and global variables) to a function as formal arguments, and model all the side effects by means of the result parameters. Furthermore, we assume that arguments passed as pointers to a function are always destroyed unless prevented from doing so by a "`const`" annotation. This allows us to automatically infer the signature from a programming language header file used commonly in languages such as C/C++ with very little user assistance.

A function signature f_i forms a *query* if (a) all its results are basic types, and (b) none of its arguments are destroyed. A query is assumed to model some attribute of its arguments. Likewise, functions that reclaim the memory allocated to compound objects are termed *destructors*. These functions destroy their compound arguments and return only basic types. Similarly, *allocators* for compound types accept basic types as inputs and allocate a return value of a required type. It is possible to relax the definition to permit allocators that take compound objects as inputs. In general, we require a *linear type hierarchy* among types so that the inputs for an allocator are always smaller in the type hierarchy than the output. In practice, the type hierarchy can be automatically inferred and functions classified as allocators, destructors or queries by means of a graphical analysis of the API structure.

Def. 3.2 (Interface Signature). *An* interface signature *is a tuple* $\langle T, \mathsf{Func} \rangle$*, where* T *is a set of types and* Func *is a set of function signatures.*

Example 3.3. *The interface signature for the stack in Figure 3, has a single compound type* stack_t*. The allocator for* stack_t *is* createStack*, the query functions include* length*,* capacity*. The functions* push *and* pop *modify their stack argument. The function free is a destructor for objects of type* stack_t*.*

3.2 Deriving Relations

We present a set of inbuilt relations based on the given interface signature that lets us accurately describe a sequence of operations on a given interface. These relations will later act as inputs to the learning problem.

Def. 3.3 (Transaction). *Given an API, a* transaction O *is a sequence of function calls to the interface functions along with their results:*

$$\begin{array}{lcl} \mathsf{op}_1 & : & \langle O_1 \rangle \leftarrow f_1(\langle I_1 \rangle) \\ \mathsf{op}_2 & : & \langle O_2 \rangle \leftarrow f_2(\langle I_2 \rangle) \\ & \vdots & \\ \mathsf{op}_n & : & \langle O_n \rangle \leftarrow f_n(\langle I_n \rangle) \end{array}$$

$$
\begin{array}{rcl}
\mathsf{op}_1 & : & o_1 \leftarrow \text{createStack}(\mathbf{10}) \\
\mathsf{op}_2 & : & \mathbf{0} \leftarrow \text{length}(o_1) \\
\mathsf{op}_3 & : & o_2 \leftarrow \text{createStack}(\mathbf{2}) \\
\mathsf{op}_4 & : & o_3 \leftarrow \text{pushStack}(o_2^-, \mathsf{ptr}0\text{x}32) \\
\mathsf{op}_5 & : & o_4 \leftarrow \text{pushStack}(o_3^-, \mathsf{ptr}0\text{x}45) \\
\mathsf{op}_6 & : & (o_5, \mathsf{ptr}0\text{x}0) \leftarrow \text{popStack}(o_1^-) \\
\mathsf{op}_7 & : & (o_6, \mathsf{ptr}0\text{x}45) \leftarrow \text{popStack}(o_4^-)
\end{array}
$$

Figure 4: A transaction on the Stack API.

Each operation is of the form $\mathsf{op}_j : \langle O_j \rangle \leftarrow f_j(\langle I_j \rangle)$ *where* op_j *is its operation ID,* $\langle O_j \rangle$ *denotes the tuple of objects output by the call to function* f_j*, while* $\langle I_j \rangle$ *denotes the tuple of arguments to the function call. We require that every compound argument* o *to an operation* op_i *should be the result of an earlier operation* op_j*. Furthermore,* o *should not be destroyed by any intermediate operation* op_k *for* $j < k < i$*.*

Transactions are similar to unit tests. In essence, a transaction describes a set of function calls to a library as well as their results. Note that transactions correspond to the popular *Single-Static Assignment* (SSA) form, commonly used in program analyses.

Example 3.4. *Figure 4 shows a transaction on the Stack API in Figure 3. Each object is given an unique ID. Objects destroyed by function calls are denoted by a* $-$ *superscript. Objects are referred to using their object IDs, whereas base types and pointers to them are denoted by their values and addresses respectively.*

Our framework learns Datalog descriptions of target concepts from relational data. Therefore, we need a standard technique for describing transactions in terms of inbuilt relations. We use relations that describe the sequence of operations, the functions called by each operation, the arguments and the result of each operation. In order to describe each operation of the transaction, we need to describe (a) the function called by the operation, (b) the object IDs and values for the basic-typed arguments, and (c) the object IDs and the values of the results. The predicates used in the relational description are summarized below:

- The relation $\mathsf{opSucc}(\mathsf{op}_i, \mathsf{op}_j)$ denotes that op_i is a successor (not necessarily immediate) of op_j.
- The predicate $\mathsf{fnCalled}(\mathsf{op}_i, f_i)$ denotes that the operation op_i calls function f_i.
- The predicate $\mathsf{fnArg_i}(\mathsf{op}_j, t_i, o_{ij})$ denotes that object ID o_{ij} is the i^{th} argument to function called by op_j and is of type t_i. This predicate is polymorphic, with different types for o_{ij}, depending on the function called.
- The predicate $\mathsf{fnRes_i}(\mathsf{op}_j, t_i, o'_{ji})$ denotes that object ID o'_{ji} is the i^{th} result parameter of the function called by operation op_j.

Example 3.5. *The relational view of the transaction in Example 3.4 is shown in Figure 5.*

Query Closure. We now focus on the nature of the query methods in an API. Since query functions are assumed not to destroy their arguments and always return basic types, we regard the results of a query q to be *attributes* of its argument(s).

$$
\begin{array}{rcl}
\mathsf{op}_1 & : & o_1 \leftarrow \text{createStack}(\mathbf{10}) \\
\mathsf{op}_2, q_0 & : & \mathbf{0} \leftarrow \text{length}(o_1) \\
q_1 & : & \mathbf{10} \leftarrow \text{capacity}(o_1) \\
\mathsf{op}_3 & : & o_2 \leftarrow \text{createStack}(\mathbf{2}) \\
q_2 & : & \mathbf{0} \leftarrow \text{length}(o_2) \\
q_3 & : & \mathbf{2} \leftarrow \text{capacity}(o_2) \\
\mathsf{op}_4 & : & o_3 \leftarrow \text{pushStack}(o_2^-, \mathsf{ptr}0\text{x}32) \\
q_4 & : & \mathbf{1} \leftarrow \text{length}(o_3) \\
q_5 & : & \mathbf{2} \leftarrow \text{capacity}(o_3) \\
\mathsf{op}_5 & : & o_4 \leftarrow \text{pushStack}(o_3^-, \mathsf{ptr}0\text{x}45) \\
 & \vdots &
\end{array}
$$

Figure 6: A query closed transaction.

Example 3.6. *For the API in Example 3.4 the query methods* length *and* capacity *can be regarded as attributes of the stack. They can be used to describe if a given stack is empty or full.*

Let $q(t_1, \ldots, t_k)$ be a query method with arguments of type $t_1, \ldots, t_k$. In order to have the maximal amount of information about the objects encountered in a transaction, we would like to run all the possible query methods on all the feasible object combinations that are produced during the transaction. Therefore, for each query q and each operation o_j, we seek to run q on all the new argument combinations that are made possible by the results of the operation.

The *query closure* of a transaction consists of adding *query calls* on all the newly created objects after every operation op_j. We distinguish the running of query functions on the new objects from the actual operations in a transaction.

Example 3.7. *The transaction shown in Example 3.4 is not query closed. However, the transaction may be modified to yield the query closed transaction shown in Figure 6.*

By convention, we let q_i denote a query while op_i denote a non query function call in a transaction. With each query function f_i, we denote a predicate

$$\mathsf{queryResult\_}f_i(o_1, \ldots, o_n, r_1, \ldots, r_m)\,,\ \text{wherein}$$

$o_1, \ldots, o_n$ denote the IDs of the parameters to the query while $r_1, \ldots, r_m$ denotes the values of the results returned by the query. Note that the operation ID corresponding to the query does not form any part of our query result predicate. The predicate is assumed to be a property of its argument objects $o_1, \ldots, o_n$ and not specific to the system state at the time of its execution. In some cases, queries are intended to capture the system state at the time of the execution. Examples include functions such as `fstat` for finding if a file exists on the disk or `time` for finding system time. In such situations, it is necessary to have the operation ID op_j be a part of the query result predicate.

4. LEARNING SPECIFICATIONS

So far, we have presented the notion of transactions on an API and shown how to represent the transaction in terms

opSucc	
op_1	op_2
op_1	op_3
$\vdots$	$\vdots$
op_1	op_7
op_2	op_3
$\vdots$	$\vdots$

fnCalled	
op_1	createStack
op_2	length
op_3	createStack
op_4	pushStack
op_5	pushStack
op_6	popStack
op_7	popStack

fnArg1		
op_1	int	10
op_2	stack_t*	o_1
op_3	int	2
op_4	stack_t*	o_2
op_5	stack_t*	o_3
op_6	stack_t*	o_1
op_7	stack_t*	o_4

fnArg2		
op_4	elt_t*	ptr0x32
op_5	elt_t*	ptr0x45

fnRes2		
op_6	elt_t*	ptr0x0
op_7	elt_t*	ptr0x45

fnRes1		
op_1	stack_t*	o_1
op_2	int	0
op_3	stack_t*	o_2
op_4	stack_t*	o_3
op_6	stack_t*	o_4
op_6	stack_t*	o_5
op_7	stack_t*	o_6

Figure 5: Relations generated by transaction on the Stack API.

father	
m_1	f_2
m_1	m_3
m_2	m_4
m_2	m_5
m_3	f_3

mother	
f_1	f_2
f_1	m_3
f_2	m_5
f_2	m_4
f_4	f_5
f_4	f_3

sibling	
m_3	f_2
f_2	m_3
f_3	f_5
f_5	f_3
m_4	m_5

¬ sibling	
f_2	m_5
m_4	f_2
f_1	m_3
f_3	f_1
m_1	m_5
m_3	f_5

Figure 7: ILP instance for the target concept sibling.

of relations. We now present a technique for mining these relations in order to learn interesting patterns about them. *Inductive Logic Programming* (ILP) is a basic technique that mines relational data for declarative specifications.

4.1 Inductive Logic Programming

Inductive Logic Programming (ILP) is a relational data mining technique that seeks to learn Prolog (Datalog) programs given some relational data, a target concept and background knowledge about the structure of the target program.

Let $p_1, \ldots, p_m$ be a set of relations over domains $D_1, \ldots, D_k$. Let p be a target relation which we would like to specify in terms of itself and the other predicates using a Datalog program. An ILP problem instance requires the following inputs:

- The relations $p_1, \ldots, p_m$,
- Positive tuples that belong to the target relation p and *negative* examples of tuples that lie outside the target relation p,
- Optionally, background knowledge that restricts the syntactic structure of the clause.

The output of an ILP instance is a set of clauses $C_1, \ldots, C_m$. Each clause C_i is of the form

$$C_i : \; p(\cdots) : -p_{i_1}(\cdots), \ldots, p_{i_k}(\cdots)\,.$$

The clauses conform to the syntactic and semantic restrictions specified as part of the background knowledge. Together, the disjunction of all the clauses cover all the positive examples and exclude all the negative examples provided.

Example 4.1. *Figure 7 shows an example ILP problem using the relations* father, mother *and the target predicate* sibling. *This instance has both positive and negative examples for the target predicate* sibling. *No background knowledge is assumed for this example.*

An ILP learner seeks to cover all the positive examples. This is achieved by repeatedly (a) choosing an uncovered positive example, (b) constructing a *saturation clause* that explains the positive example, and (c) generalizing this clause to cover as many positive examples as possible and no negative example. The generalized clause is added to the pool of learned clauses and the positive examples covered by it are removed from the uncovered set. This process is best illustrated by an example.

Example 4.2. *Consider the ILP instance in Example 4.1. We choose the positive example* $\langle m_3, f_2\rangle \in \mathsf{sibling}$. *The relevant tuples concerning* m_3, f_2 *are*

$$\mathsf{father}(m_1, m_3),\ \mathsf{mother}(f_1, m_3),\ \mathsf{father}(m_1, f_2),\ \mathsf{mother}(f_1, f_2)$$

These tuples enable us to construct the following saturation clause *by generalizing domain values by means of variables:*

$$\begin{array}{lll} \mathsf{sibling}(X,Y) & :- & \mathsf{father}(M,X), \mathsf{father}(M,Y) \\ & & \mathsf{mother}(F,X), \mathsf{mother}(F,Y)\,. \end{array}$$

This clause covers all but one of the positive examples and none of the negative examples. It is nearly an ideal clause. We now consider various generalizations of this clause. One of the important generalizations is to drop antecedents from the clause to cover more positive examples. However, this is not the only generalization possible. Many solvers may consider other means of generalizations such as making two instances of the same variable distinct, adding new predicates, abducing *predicates and so on.*

Let us consider the following generalizations of the saturated clause along with the number of positive and negative examples that satisfy the clause:

$$\begin{array}{lll|l} \mathsf{sibling}(X,Y) & :- & \mathsf{father}(M,X), \mathsf{mother}(F,Y) & (5,6) \\ \mathsf{sibling}(X,Y) & :- & \mathsf{father}(M,X), \mathsf{father}(M,Y) & (5,0) \\ \mathsf{sibling}(X,Y) & :- & \mathsf{mother}(F,X), \mathsf{mother}(F,Y) & (6,0) \end{array}$$

Among all the clauses the last one is ideal since it is the smallest clause that explains the largest number of positive examples and none of the negative examples. It is therefore added to the set of learned rules. Since there are no uncovered positive examples left, the final learned program consists of a single clause

$$C : \; \mathsf{sibling}(X,Y) : -\mathsf{mother}(F,X), \mathsf{mother}(F,Y)\,.$$

While this result conforms to the familiar concept of a "sibling", the following clause is not learned:

$$C' : \; \mathsf{sibling}(X,Y) : -\mathsf{father}(M,X), \mathsf{father}(M,Y)\,.$$

Due to inadequate data, clause C *is enough to explain all the data that can be explained by* C'.

4.2 Learning Target Specifications

ILP can be used to learn specifications for some target behaviour given the output of many transactions carried out on a library. In general, the target is chosen by the user depending on the intended application and specified as a part of the learning problem. Common examples of targets include an operation throwing an exception (an assertion violation), returning a special value of interest, printing a specified message on the output and so on. We assume that the target predicate can be evaluated by examining the transaction.

For the remainder of this section, we seek to predict the conditions under which a call to the function f_i throws an exception. The target predicate, denoted $\mathsf{throwsException}(f, \mathsf{op})$, indicates that a function call to f during an operation op throws an exception. We may instrument transactions to print positive examples of the predicate whenever an operation throws an exception and negative examples otherwise.

Let $O_1, \ldots, O_N$ be a set of transactions run on the library whose behaviour we seek to infer. Let us assume that the set of operation IDs and object IDs involved in these transactions are pairwise disjoint. We also assume that each transaction is query closed. The transactions are converted into inbuilt relations, as discussed previously. Furthermore, the outcome of the target predicate (eg., exception being thrown) is tested at the end of each operation and classified into positive and negative instances.

Finally, the user may provide some background knowledge (hints) that constrain the search for rules. The nature of this knowledge and its effect on the learning algorithm is discussed subsequently. The data thus collected yields an ILP instance and is fed to an ILP learner.

Example 4.3. *Consider the transaction shown in Example 3.7. We use the target* $\mathsf{popReturnsNullPtr}(\mathsf{op\_id})$. *The only positive example for the target is* $\mathsf{popReturnsNullPtr}(\mathsf{op}_6)$, *while all the other operations form negative examples. In general, the amount of data involved in this instance is quite small. Learning algorithms typically perform well only when data is available from a variety of transactions, each exercising a variety of situations. For instance, if the data volume is insufficient, the result may vary widely depending on the search heuristic employed. In this instance, the clause*

$$\begin{aligned}\mathsf{popReturnsNullPtr}(X) :- \quad & \mathsf{fnCalled}(X, \text{``}pop\text{''}),\\ & \mathsf{fnArg}_1(X, \mathsf{stack\_t}, S),\\ & \mathsf{queryResult}(\text{``}capacity\text{''}, S, \mathbf{10}).\end{aligned}$$

states that a pop operation returns NULL whenever the capacity of its first argument is 10. *This clause covers the single positive instance and excludes all the negative instances in this case. Nevertheless, it is possible in principle (by choosing the right search heuristics) to this example to yield:*

$$\begin{aligned}\mathsf{popReturnsNullPtr}(X) :- \quad & \mathsf{fnCalled}(X, \text{``}pop''),\\ & \mathsf{fnArg}_1(X, \mathsf{stack\_t}, S),\\ & \mathsf{queryResult}(\text{``}length'', S, \mathbf{0}).\end{aligned}$$

The clause above states that pop operation may return null if the length of its argument is 0. *The former clause can be avoided in the presence of enough data either because of negative examples of stacks with capacity* 10 *(but with* 1 *or more elements) that* do not *return a NULL on pop, or more positive examples with varying capacities but length* 0 *that return a NULL. Finally, commonly used search heuristics are governed by the* "Occam's Razor" *and prefer constants such as* 0 *over* 10.

We have observed that given higher data volumes, it is possible to reliably learn the right clause using many heuristics.

4.3 Background Knowledge

We distinguish between two different types of background knowledge (a) *Structural* information about the clauses, and (b) User defined predicates that may be used by the learner.

Structural Knowledge. It is possible to annotate clauses using structural information. The exact form of this information varies, depending upon the search used by the ILP learner. In general, it is possible to restrict the possible combination of predicates that may form a part of the body of a clause by means of *mode annotations* on predicates. Let $p(x_1, \ldots, x_m)$ be a predicate in an ILP problem instance. A *mode annotation* for p classifies each input argument x_i into one of three categories (a) an input variable x_i^I, (b) an output variable x_i^O or (c) a constant $x_i^\#$.

The ILP learner uses mode annotations to learn clauses according to the following rules:

1. Any term X that instantiates an argument x_i^I in the body of a predicate p_i *must* appear in the head p as an input argument, or as an output argument of an earlier predicate p_j in the clause body.
2. Any output argument x^O in the head must also appear as an output in the body of the clause.
3. Any argument $x_i^\#$ is instantiated by a constant term.

Annotations speed up the search by restricting the clause structure. Our experiments with ILP learners use the following annotations for the inbuilt relations:

- $\mathsf{throwsException}(\mathsf{op}^I, \mathsf{fn}^\#)$, $\mathsf{fnCalled}(\mathsf{op}^I, \mathsf{fn}^\#)$
- $\mathsf{fnArg}_i(\mathsf{op}^I, \mathsf{typ}^\#, \mathsf{obj\_id}^O)$, $\mathsf{fnRes}_i(\mathsf{op}^I, \mathsf{typ}^\#, \mathsf{obj\_id}^O)$,
- $\mathsf{queryResult\_fn}(\mathsf{obj}_1^I, \ldots, \mathsf{obj}_n^I, r_1^O, \ldots, r_m^O)$,
- $\mathsf{opSucc}(\mathsf{op}^I, \mathsf{op}^O)$.

Background knowledge may also consist of user-defined predicates. We require common predicates such as equality, comparison operators such as inequalities, disequalities, and constants such as $0, 1, -1$ and so on. Depending on the tool used, such predicates may or may not be inbuilt. We simply establish a library of common background predicates that the learner may use in the body of the learned clauses.

Finally background knowledge may also be provided in the form of dependency information between the target predicate and the different predicates used in the data. In general, if a predicate p_i is known not to affect the target predicate p, it need not be considered by the learner while inferring the specification for p.

5. TRANSACTION GENERATION

We now turn to the automatic generation of transactions given a library with an API signature $\langle T, \mathsf{Func}\rangle$. Our technique relies on the generation of *random unit tests*. These unit tests are instrumented to yield the relations that form the inputs to the learning problem. The unit test generator

Algorithm 1: Randomized unit test generation scheme.

```
Input: ⟨T, Func⟩: API Signature, N: initial pool size
Result: Unit Test output
begin
    for each obj. type t ∈ T do
        /* Initialize the pool for the type t    */
        pool[t] ← initPoolForType(t, N)
    while ··· do                          /* Run unit test */
        Sel. (t'_1, ..., t'_n) ← f(t_1, ..., t_m) ∈ Func − Queries
        Sel. arguments i_1, ..., i_m
        Args. are chosen randomly from their corr. pools
        Execute : ⟨o'_1, ..., o'_n⟩ ← f(i_1, ..., i_m)
        Remove destroyed arguments from their pool
        Run queries on the return vals. and objects in the pool
        Add returned objects to the pool
end
```

is called repeatedly in our framework and the resulting relations are merged with those collected from previous unit tests.

Random Unit Tests. Algorithm 1 outlines our random unit testing approach. Our approach first constructs a fixed size *pool* of object references to each type in the signature. The size of this pool is a user input.

The pool for basic types such as int, char are constructed by using random number generators. Arrays of basic types such as strings and integer arrays are constructed by allocating memory with randomly chosen lengths, and filling the allocated arrays with randomly generated content.

Pools of compound objects are then constructed by using allocator methods for these objects. In principle, any method that returns a reference to an object of type t can be thought of as an allocator for the type. However, during the construction of the object pool, not all such methods are suitable. For instance, the pushStack method for the stack API in Figure 3 is unsuitable for constructing a stack object pool, since it requires a stack object as an argument that is destroyed by the call. The allocators and the order of object allocations are selected by analyzing the *parameter graph* of the API signature following the approach of Csallner and Smaragdakis [8].

Once all the objects in the pool are allocated, the unit test generator repeatedly chooses non query methods at random and executes them with inputs chosen randomly from the pool. The outputs obtained are added back to the object pool, while the arguments destroyed by the method are removed. By removing objects that are destroyed by a function call, we ensure that their object IDs are not used in subsequent operations. In order to obtain query closed transactions, each query method in the API is run on all possible combinations of inputs from the pool, after each operation.

The unit tests generated by our technique are then instrumented in order to generate the transaction along with the relations required for the ILP learner. This is achieved by numbering different operations issued by the unit test and the objects produced/consumed by these operations.

The test generator is also instrumented based on the target predicates chosen by the user. For instance, in order to learn the exception behaviour, the unit test sets up exception handlers for various types of exceptions. Upon the receipt of an exception, a positive instance of the target predicate is generated. If an operation ends successfully without an exception being thrown, a negative instance of the corresponding target predicate is printed.

Each unit test is run for a fixed number of steps. Tests are stopped upon the throwing an exception or the exhaustion of a resource such as time/memory. Typically, the unit test generator is also reseeded repeatedly to enhance randomization. The relations obtained from different unit tests are all merged (after renumbering object and operation IDs to make them unique). They may be fed directly into an ILP learner to learn specifications.

6. EXPERIMENTS

We have implemented a tool flow that automatically generates random unit tests given the interface description. These tests are instrumented to generate relational transactions which are then fed as inputs to the ILP learning tool Aleph [20]. In order to facilitate the learning, we provide predicates that model relational operators such as $\{=, \geq, \leq, <, >, \neq\}$, as well as comparison with the constants zero and one. The output of the learner is post-processed to consider the *support* for each rule, i.e., number of positive examples explained by the rule. Rules with high support are more reliable than those with lower supports. The latter rules are more likely to be products of accidental biases in the tests than actual properties of the API. Consequently, we reject rules with inadequate support (<5% of the total positive instances). The exceptions explained by these rules are treated as *uncovered*.

We evaluate our tool on a variety of benchmark examples. These examples include data structure implementations for stacks and lists, the Parma Polyhedron Library, the Unix file library, the "Stdlib" Math library and the Bignum library used for cryptographic applications in OpenSSL. For each library, we run numerous unit tests, depending on the number of API functions and their complexity [4]. Table 1 shows the number of tests (each test generates one transaction) generated along with the number of exception instances. The output data for the ILP learner consists of the number of rules learned, the time taken and the number of exceptions that were left uncovered by the rules learned. The data show that our learned rules were able to explain a vast majority of all the exceptions raised by the unit tests with adequate support. Finally, the rules learned themselves were compared against the documentation for the library for accuracy. In general, almost all the rules were borne out by the library documentation. However, the output for the Unix file and GLPK libraries contained factually incorrect rules that had significant support.

Insufficient Instances. One of the common problems with random unit testing is that the number of positive and/or negative examples for a particular type of exception can be too few even though many tests are run.

Example 6.1 (Complex API requirements). *The documented requirements for the function*

$$\mathsf{addRowToLP}(\mathsf{lp\_t} *\ lp,\ \mathsf{int}\ len,\ \mathsf{int} *\ ind, \mathsf{float} *\ ptr)$$

in the GLPK library requires (a) a properly initialized lp, (b) the parameter "len" be less than the array lengths of the

[4]The data and results for these experiments are available by requesting sriram@nec-labs.com

Table 1: Summary of results on data structure implementations: #Fn: number of API functions, #Tr: number of transactions, #Pos: number of exception instances, #R: number of rules learned, T: ILP learner time taken in seconds, #U: number of uncovered exceptions, #In: Number of rules that are inaccurate.

Name	Descr.	#Fn	#Tr.	#Pos	#R	T	#U	#In.
Stack	Stack data struct.	6	200	160	4	.1	0	0
List	List data struct.	9	500	251	4	38	20	0
GLPK	GNU Linear Programming Kit (LP setup)	15	8000	7909	9	36	0	2
PPL	Parma Polyhedra Library	18	2000	1373	14	11	1	0
Varset	Bitvector sets	19	500	408	19	18	0	0
File	C stdio FILE interface	26	2000	1110	21	51	28	1
Math	Stdlib math	46	2000	31	1	1	0	0
Bignum	Openssl bignum package	63	5000	545	8	153	12	0

pointers "ind" and "ptr", (c) each element of the array "ind" be positive and less than the number of columns in "lp" and (d) each element of the array "ptr" be non zero.

The probability of automatically obtaining a random test that exercises this function while satisfying all the requirements listed above is vanishingly small. Therefore, no negative examples are encountered in this case. In turn, our tool concludes that such a function always raises an exception with support from a large number of positive examples (since no negative examples are present to refute such a claim).

In general, a large number of random unit tests may be required to provide a few examples that demonstrate the proper working of such a function. The resulting data volume can slow down the ILP engine. Alternatively, the incorrect clause can guide the design of specific unit tests that may demonstrate the correct working of the function, thus leading our tool towards the right concept.

Closed World Assumption. The ILP learner can fail to learn the right concept, or infer an incorrect concept when unmodelled factors depending on the run time environment can affect the behaviour of the API.

Example 6.2 (File Operations). *Consider the common library function*

$$\mathsf{FILE}* \leftarrow \mathsf{fopen}(\mathsf{char}*\ fname, \mathsf{rw\_mode\_t}\ rwmode)\,.$$

The behaviour of fopen *on some file* `foo.txt` *may vary depending on whether the file already exists in the disk, or not. This leads to non determinism in the function behaviour that cannot be explained by the learner.*

The problem is addressed by running a query function such as fstat *that finds if a file exists on the disk before each operation. Furthermore, since a file may be created on the disk by means of a previous operation, the query function in this case needs to be predicated on the subsequent operation ID in the transaction. Consequent to these changes, the learner correctly infers rules explaining the return value of the* fopen *API.*

Expressiveness. In general, limitations to the expressiveness stem from the lack of background knowledge or inbuilt predicates required to express certain properties.

Example 6.3. *The function* addRowToLP *described in Example 6.1 requires that the contents of arrays* ind *and* ptr *follow some specific requirements (requirements (c) & (d)). Our current model does not track array contents. Therefore, it can capture requirements (a), (b) for an exception. However, requirements (c), (d) cannot be learned.*

Table 2: Some rules inferred for `Bignum` library.

R1,R2	Check result of `BN_generate_prime` and `BN_mod_inverse` for NULL before use.
R3,R4	The second argument to `BN_generate_prime` and `BN_rand` should be positive.

Table 3: Checking the rules R1-R4 on applications using the `bignum` library. #C: Number of calls, #OK: Check passes, #NG: violation possible.

Rule	#C	#OK	#NG
R1	10	8	2
R2	12	11	1
R3	10	5	5
R4	98	90	8

Other pitfalls include noise in the data due to unreliable exception handling for faults such as segmentation violation, and also due to interference caused by aliasing and sharing, that can confuse the learner. In practice, ILP learners employ heuristics that make them insensitive to small amounts of noise in the data.

Checking Learned Rules

In principle, it is possible to convert the Datalog rules automatically into static checks on application code. Following the approach in this paper, we may statically gather all the possible sequences of function calls for a given object o by means of static analysis to form static transactions. The resulting static transactions can be converted into relations and checked against the learned rules to infer the possibility of an exception on some inputs. Efficient static analysis tools for evaluating Datalog queries such as BDDBDDB are ideal for this purpose [21].

We provide a proof of concept by manually interpreting the clauses into static checks, based on four of the rules inferred for the `bignum` library as described in Table 2. Table 3 shows the results of a manual inspection of the source code of the rest of the OpenSSL library and other applications available online for instances that violate the checks [5]. A majority of the calls to the functions involved do conform to the checks described in Table 2. Some of the violations found can be exploited to produce an exception while others simply translate into requirements on the OpenSSL library functions. At least one application considered could be crashed, specifically due to a missing check for R3 [6].

7. CONCLUSION

In conclusion, we have provided a framework for inductively inferring useful specifications for libraries targeted towards a specific concept. Our technique uses random test generation to compile the behaviour of the library into relations that can be mined for useful Datalog properties. We have also provided a proof of concept for potentially converting the mined properties into feasible static checks.

8. REFERENCES

[1] Acharya, M., Xie, T., Pei, J., and Xu, J. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07* (2007), ACM Press, pp. 25–34.

[2] Alur, R., Černý, P., Madhusudan, P., and Nam, W. Synthesis of interface specifications for java classes. In *Proc. ACM SIGPLAN symp. on Principles of prog. lang. (POPL)* (2005), ACM Press, pp. 98–109.

[3] Ammons, G., Bodík, R., and Larus, J. R. Mining specifications. In *POPL* (2002), pp. 4–16.

[4] Bergadano, F., and Gunetti, D. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, 1996.

[5] Cadar, C., and Engler, D. R. Execution Generated Test Cases: How to make systems code crash itself. In *SPIN* (2005), vol. 3639 of *LNCS*, Springer–Verlag, pp. 2–23.

[6] Christodorescu, M., Kruegel, C., and Jha, S. Mining specifications of malicious behavior. In *ESEC-FSE '07* (2007), ACM Press, pp. 5–14.

[7] Cohen, W. W. Recovering software specifications with inductive logic programming. In *AAAI* (1994), pp. 142–148.

[8] Csallner, C., and Smaragdakis, Y. Jcrasher: an automatic robustness tester for Java. *Softw., Pract. Exper. 34*, 11 (2004), 1025–1050.

[9] de Alfaro, L., and Henzinger, T. A. Interface automata. In *ESEC / SIGSOFT FSE* (2001), ACM Press, pp. 109–120.

[10] Ernst, M. D. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug. 2000.

[11] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05)* (2005), pp. 213–223.

[12] Jiang, G., Chen, H., Ungureanu, C., and Yoshihira, K. Multi-resolution abnormal trace detection using varied-length N-grams and automata. In *ICAC* (2005), IEEE Computer Society, pp. 111–122.

[13] Jones, C. B. *Software Development: A Rigorous Approach*. 1980.

[14] Mitchell, T. M. *Machine Learning*. McGraw-Hill, 1997.

[15] Muggleton, S. Inverse resolution and Progol. *New Generation Computing 13* (1995), 245–286.

[16] Pazzani, M., Brunk, C., and Silverstein, G. A knowledge-intensive approach to learning relational concepts. In *Proc. Workshop on Machine Learning* (1991), Morgan Kauffmann, pp. 432–436.

[17] Quinlan, J. R. Learning logical definitions from relations. *Machine Learning 5* (1990), 239–266.

[18] Ramanathan, M. K., Grama, A., and Jagannathan, S. Static specification inference using predicate mining. In *Proc. ACM SIGPLAN Prog. Lang. Design & Implementation (PLDI)* (2007), ACM press, pp. 123–134.

[19] Sen, K., Marinov, D., and Agha, G. Cute: A concolic unit testing engine for c. In *ESEC/FSE'05* (2005), ACM Press.

[20] Srinivasan, A. The Aleph manual. Available at http://www.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

[21] Whaley, J., and Lam, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04* (2004), ACM Press, pp. 131–144.

[22] Whaley, J., Martin, M. C., and Lam, M. S. Automatic extraction of object-oriented component interfaces. In *ISSTA* (2002), pp. 218–228.

[23] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M. Perracotta: Mining temporal API rules from imperfect traces. In *Proc. of ICSE* (2006).

[5] Cf. http://www.openssl.org/related/apps.html.

[6] Cf. Chameleon Hash Implementation: http://www.dsi.uniroma1.it/~patil/projects/cham/main.c.