Synthesizing Intensional Behavior Models by Graph Transformation

Carlo Ghezzi
Politecnico di Milano
DeepSE Group at DEI
Piazza L. da Vinci, 32
20133 Milano, Italy
ghezzi@elet.polimi.it

Andrea Mocci
Politecnico di Milano
DeepSE Group at DEI
Piazza L. da Vinci, 32
20133 Milano, Italy
mocci@elet.polimi.it

Mattia Monga
Università degli Studi di Milano
DICo
Via Comelico, 39
20135 Milano, Italy
mattia.monga@unimi.it

Abstract

This paper describes an approach (SPY) to recovering the specification of a software component from the observation of its run-time behavior. It focuses on components that behave as data abstractions. Components are assumed to be black boxes that do not allow any implementation inspection. The inferred description may help understand what the component does when no formal specification is available. SPY works in two main stages. First, it builds a deterministic finite-state machine that models the partial behavior of instances of the data abstraction. This is then generalized via graph transformation rules. The rules can generate a possibly infinite number of behavior models, which generalize the description of the data abstraction under an assumption of "regularity" with respect to the observed behavior. The rules can be viewed as a likely specification of the data abstraction. We illustrate how SPY works on relevant examples and we compare it with competing methods.

1. Introduction and Motivations

A *specification* of a component is a formal documentation of its behavior upon which clients can rely [14]. There is still no universally accepted way to formally specify a component. However, most approaches distinguish between a *syntactic part*, which describes the component's signature, and a *semantic part*, which describes the visible effects for the clients, achieved by using the component through its interface. The semantic part can be expressed in a formal notation, such as JML [18], which can be used to specify *contracts* [20] via pre-, post-conditions, and invariants. Other component descriptions may be provided in terms of different kinds of state machine models [15, 8].

Producing a specification can be as expensive as writing the component itself, therefore in practice an interface documentation is often given informally. Furthermore, often it is not kept fully consistent with the actual implementation during software evolution. Thus, the ability to recover the specification from the component may help improving its documentation.

The lack of a precise specification is especially troublesome in a scenario where components are exposed as black boxes for possible use by others, as in the case of Web services or, more generally, Service Oriented Architectures (SOAs) [17]. In a SOA, an explicit discovery phase supports the selection of possible candidate components to be composed. Discovery relies on a specification, which should tell what the service actually does. Furthermore, the code of an exposed service is not available for inspection. It may be invoked remotely from clients, but its internal details cannot be examined.

The goal of our research is to infer a formal specification of *black box components* by observing their run-time behavior. This paper investigates an important preliminary step. It focuses on *stateful components* that behave as *data abstractions*; more specifically, on components —implemented as Java classes— that behave as data containers. Being black-boxes, components can only be observed by dynamically invoking the operations exported through their interface and by inspecting the values returned from the calls.

The main input of our inference approach, called SPY (specification recovery) is a description of the interface of the data abstraction, that is, the signature of its constructors and methods. Every method is considered a *modifier* of the internal state. If a method has a non-void return type, we call it an *observer*. By definition, an observer is *pure* if it does not modify the internal state of the object. A secondary input consists of a set of *instance pools* which are used to get values for method parameters.

Because SPY is based on dynamic analysis, a component undergoes a sequence of state changes. It is thus necessary to state precisely and detect when the *same* state is entered. This is captured by the notion of *behavioral equivalence*, as given by [10]. Given two objects o_1 and o_2 instances of

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

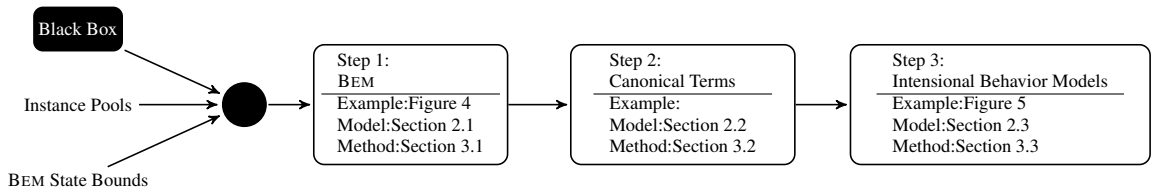


Figure 1: Outline of the proposed method and the paper structure

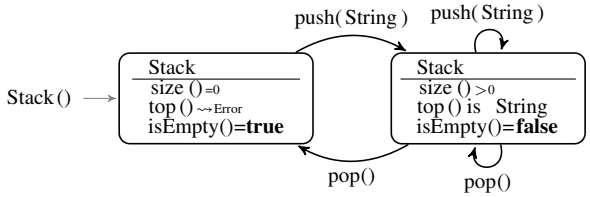


Figure 2: The ADABU model of the Stack data abstraction.

```
public class Stack {
  public Stack() { .. }
  public void push(String element) { .. }
  public void pop() throws Error { .. }
  public String top() throws Error { .. }
  public boolean isEmpty() { .. }
  public int size() { .. }
}
```

Figure 3: The public interface of a Stack of Strings

a class C, o_1 and o_2 are *behaviorally equivalent* if for any sequence of operations t of C ending with an observer, the objects $o_1.t$ and $o_2.t$ obtained by invoking t are themselves behaviorally equivalent. For observers returning primitive types, they are behaviorally equivalent if their values are the same. In other words, behavioral equivalence is an operational way to specify instances that are in the same "*abstract state*" [14], independent of their actual internal state.

Figure 1 outlines our approach together with the structure of this paper. Starting from a data abstraction signature and instance pools, our approach does the following:

1. We build (up to a configurable bound to the number of states k) a finite state automaton where each state denotes behaviorally equivalent classes of objects (*behavioral equivalence model* BEM) (Section 3.1).

2. We label each BEM state with a *canonical term*, that is, a string representing a sequence of method calls that can generate an object belonging to the equivalence class denoted by that state (Section 3.2).

3. We build a set of graph transformation rules that can generate the labeled BEM (Section 3.3.1), and then we apply invariant detection to generalize the rules and obtain an *intensional behavior model*, which can produce BEMs describing every possible behavior of the data abstraction (See Section 3.3).

The paper is organized as follows. Section 2 describes the background concepts and the rationale of our method through an example. Section 3 details the phases of the recovery of intensional behavior models through its phases. Section 4 illustrates the state of the art and positions our contribution. Section 5 evaluates our approach. Finally, Section 6 draws some conclusions and outlines future research directions.

2. An Introductory Example

This section provides a stepwise and informal introduction to the specifications recovered by SPY and shows how they apply to the venerable Stack example, whose interface is shown on Figure 3. The main focus here is on the models we can recover with SPY. Section 3 will then focus on our main contribution, namely the inference process supported by SPY, through which specifications can be recovered.

Although Stack is a simple data abstraction, it has been used traditionally as a reference example for formal specification, including approaches to specification recovery: from pioneers (like DAIKON [12]) to recent ones (like DYSY [7]). The application of our approach to more complex data abstractions is discussed later in Section 4.

2.1. Behavioral equivalence models

In the literature, *behavior models* are used to describe software components as finite state automata. They do so at different levels of abstraction by identifying states through an abstraction criterion, stated by a predicate, on the return values of observers. A state represents all instance objects such that the return values of observers satisfy the criterion associated with the state. Different approaches differ by the choosen abstraction criteria, which are a direct consequence of the purpose for which the model is used or inferred.

For example, ADABU [8] can recover behavior models that, in the case of Stack, is illustrated in Figure 2. The

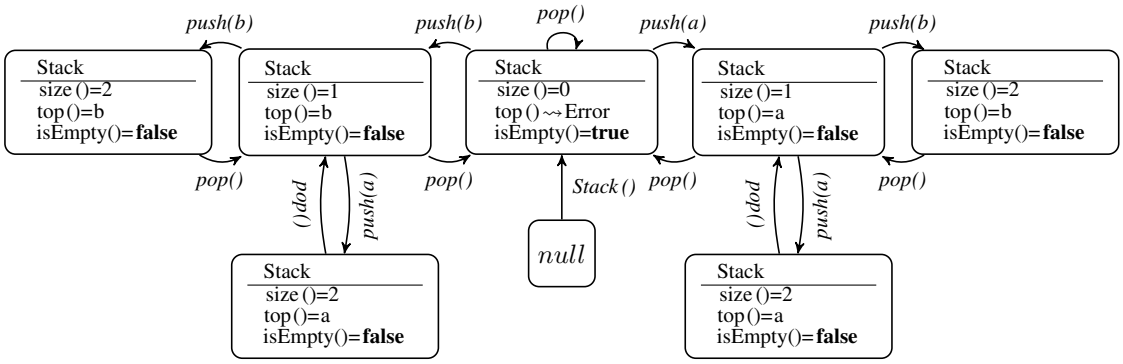


Figure 4: A BEM of the Stack data abstraction

model can show that after a *push(...)* operation the stack is not empty. Also, distinctive properties as the fact that the *top()* method returns the last element that was inserted, cannot be derived by the model. For example, the ADABU model shown in Figure 2 admits nondeterministic behaviors, although the component is deterministic. Moreover, it describes behaviors that do not occur in reality. For example, it describes a sequence of *push(...)* followed by *pop()* which yields a non-empty stack if it is applied to an empty stack. In conclusion, behavior models are too imprecise to be used as specifications.

In the sequel, we show how, under some loose regularity conditions (detailed in Section 5), one can provide precise behavior models for containers, which describe exactly all of the component's behaviors. To do so, we first introduce *behavioral equivalence models* (BEMs), which leverage the notion of behavioral equivalence. Figure 4 shows a BEM for Stack. The model describes the behavior of Stack when two strings, *a* and *b*, are used as possible storable objects, and up to size *2*. Each state of the BEM represents an abstract state, that is, a class of behaviorally equivalent objects. For instance, the state reached after the application of the constructor represents all empty stack instances. The state is labeled with the return values of observers that characterize the behavior of the empty stack.

Observer return values do not uniquely identify behavioral equivalence classes (i.e., BEM states). For example, starting from an empty stack, the stack obtained by first pushing *a* and then *b* and the stack obtained by first pushing *b* and then *b* yield the same return values of observers. However, the two objects are not behaviorally equivalent. Thus, two different states (the leftmost and the rightmost state in Figure 4) characterize the two different behavioral equivalence classes.

The BEM in Figure 4 provides a very precise, yet incomplete, behavioral description of Stack. Unlike the model of Figure 2, it shows that a *push(...)* operation followed by a *pop()* applied to an empty stack yields an empty stack. The model, however, is incomplete because (1) it defines the behavior of Stack for only two possible storable values (*a* and *b*), and (2) it handles only stacks up to size 2. Because the set of string values and the size of the stack are both finite, Stack can be modeled by a finite-state machine. However, for any nontrivial data abstraction (including Stack), these constraints do not hold. First, methods can have parameter types whose value set is theoretically infinite. Second, even for finite sets of parameter types, the number of behavioral equivalence classes is in theory infinite. For example, consider the Stack represented in Figure 4, which only models stacks of maximum length 2. It would be necessary to add new states to model stacks of maximum length 3. Similar considerations hold for any nontrivial data abstractions that represent data containers. For all of them, we may conclude that no finite BEM can be used to provide a precise and complete specification.

To achieve this goal, we use a formalism that combines BEMs with a *generative* mechanism. Because it is generally impossible to enumerate all BEMs extensionally, the formalism provides generative rules that achieve the same goal *intensionally*. Since BEMs are represented by graphs, a *graph transformation system* (GTS) may provide such formalism. Before we show in detail how a GTS can solve the problem, however, we need to address a preliminary issue. The rules that define a GTS require graph nodes to be uniquely identifiable by a label. As we observed earlier, however, the return values of the observers associated with each state do not uniquely identify the state. In the next section we show how we can do so by labeling each state with a string that represents a canonical sequence of operations that generates a representative object of the behavioral equivalence class modeled by that state.

2.2. Labeling states by canonical terms

Each BEM state can be uniquely labeled by a representative term. A representative term for a state S is a string

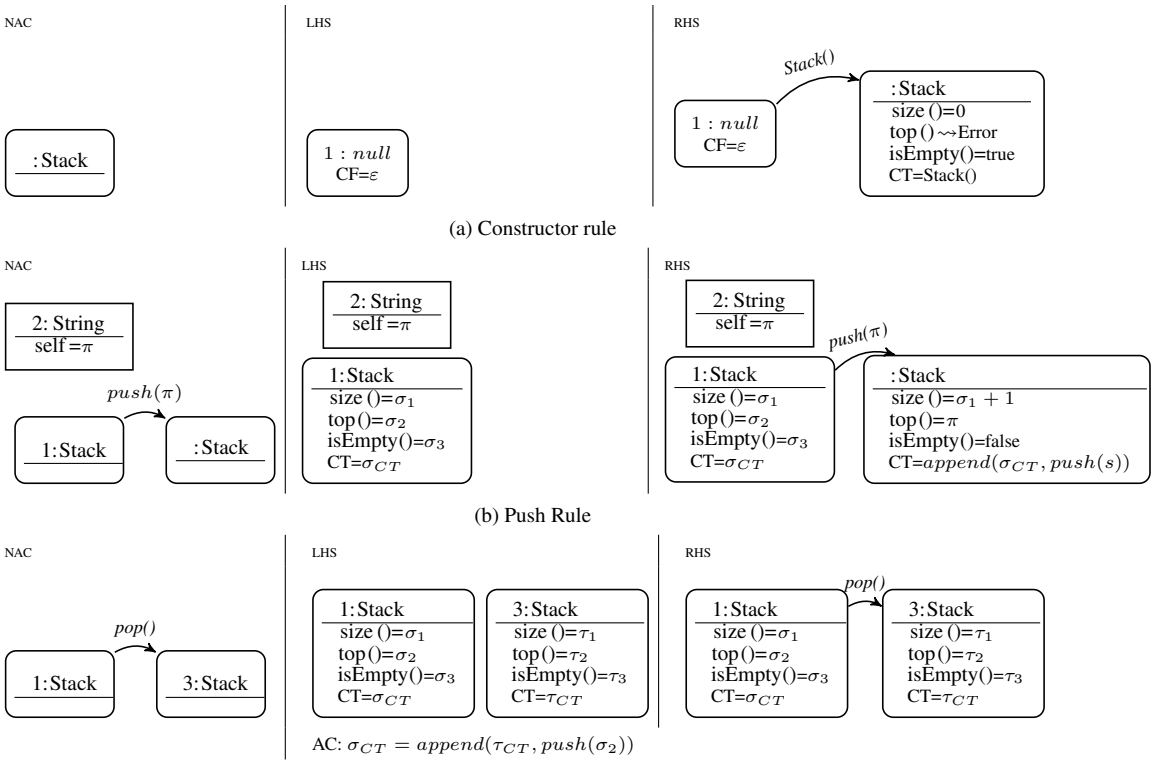


Figure 5: Stack Intensional Behavior Model

that defines a *canonical* sequence of invocations of operations that generates an object belonging to the behavioral equivalence class associated with S. We call it *canonical term (CT)*. Topologically, the CT that labels a state S is the concatenation of edge labels found by following a path on the graph describing a BEM, from the initial ($null$) state to S. Although in principle any sequence of operations that generates an object can be chosen to label the state denoting the object's behavioral equivalence class, in practice certain terms naturally stand as candidates for CTs. In the case of Stack, CTs can be defined as follows. For any state S of the BEM modeling Stack, the class of behaviorally equivalent objects associated with S can be represented by the object obtained by invoking the constructor followed by a sequence of *push(...)* operations. For example, the CT *Stack()push(b)push(b)* is used to label the leftmost state in Figure 4. We consider other terms, such as *Stack()push(b)pop()push(b)push (b)*, which also generate a behaviorally equivalent object for that state, not to be CTs.

The notion of a CT is reminiscent of the notion of normal form term in algebraic specifications [22]. This issue is not discussed here for space reasons. Formally, we can express the set of CTs by a formal language; in the Stack example, we can describe it by the following grammar:

$$\begin{aligned} S &\Rightarrow \varepsilon \mid Stack()\ T \\ T &\Rightarrow \varepsilon \mid push(x)\ T \end{aligned}$$

where x is in the domain of strings. In general, CTs are defined by a *canonical term language* $\mathcal{L}$ satisfying the following properties (ℓ_1) and (ℓ_2):

- (ℓ_1): each term is composed of an initial constructor and a —possibly empty— sequence of modifiers;
- (ℓ_2): for every $t = u \cdot m(p) \in \mathcal{L}$, $u \in \mathcal{L}$; that is, every term in the language is obtained by appending an operation $m(p)$ to a term u belonging to the language. By definition, $m(p)$ is called a *canonical operation* in the context of u.

In the case of Stack, the canonical term language is a simple regular language. This is rather an exception than the normal case: in general, the canonical language can belong to any class of Chomsky's hierarchy.

2.3. Defining BEMs by a GTS

Behavioral equivalence models are essentially finite-state automata, and thus, they are graphs. A graph trans-

formation system (GTS) [11] is a formal method to express how a class of graphs can be generated. In a previous paper, we studied the use of GTS to specify the behavior of data abstractions [4]: here we briefly summarize the proposal.

In a GTS, rules describe how a host graph is modified by their application. Specifically, we use *attributed GTSs* as defined in [11], where nodes and arcs are labeled with typed attributes. As a support tool for the formal method, we use AGG [24]. Each rule is described by three graphs, NAC, LHS, RHS, and a set of attribute conditions AC.

LHS, NAC, and AC define when a rule can be applied to a source graph. LHS and NAC are graphs whose nodes and arcs are labeled with variables on the domain of the attributes. Roughly speaking, LHS defines a positive topological condition: it describes a subgraph that must occurr in a source graph for the rule to be applied. Instead, NAC expresses a negative application condition: the rule cannot be applied if an occurrence of the NAC exists in the source graph. Moreover, we can define mappings for nodes and arcs in LHS, RHS and NAC by identically numbering them in the rule. AC is a binary predicate on variables defined on the LHS attributes, which must be checked to hold for the rule to be applied. Finally, the RHS describes how the graph should be transformed by the rule, with respect to attributes and topology.

If the positive and negative applicability conditions are verified, the rule transforms the source graph in a new graph. Since in our setting the LHS is always a subgraph of the RHS, the resulting graph is built by adding all the nodes and arcs which are introduced by the RHS (i.e., all the graph elements in the RHS that are not destinations of mappings from the LHS). Attributes are modified according to functions labeling nodes in the RHS.

Let us describe intuitively how the BEM of Figure 4 can be generated by applying the rules in Figure 5. Consider a graph containing only the $null$ state, which conventionally represents the state of an object before the application of any constructor. The only applicable rule is the constructor rule of Figure 5a, which contains the null state on the LHS. Its NAC does not hold, because the null state does not have any outgoing transition. The mapping is defined by identically numbered nodes: the node on the LHS labeled with 1, representing the null state, corresponds to the same node labeled with 1 in the RHS. The application of RHS simply adds to it a transition labeled with the constructor which leads to a newly generated state representing the empty stack; this is an example of a *canonical rule*. Canonical rules —that is, rules describing the application of a canonical operation— always introduce a new node, when applied to a graph. The node rerepresents a new state of the BEM, corresponding to a newly introduced behavioral equivalence class. It is important to observe that the canonical language is implicitly defined by the assignments to CT in the newly generated states, as defined by the RHS of the canonical rules.

The constructor rule can only be applied once; after being applied, it is disabled by its NAC.The canonical rule in Figure 5(b) is instead enabled, which corresponds to the application of the *push(String)* operation. The rule can be applied because the host graph contains a node matched by its LHS. The application generates a node representing a stack with a single element. It is then possible to apply the rule in Figure 5(c), corresponding to a *pop()* operation, by matching states 1 and 3 of the rule with the empty stack and the stack containing a single element. This is not a canonical operation: the rule simply adds a transition to the BEM.

The intensional model can generate every possible behavioral equivalence class: it models the fact that any parameter of type String can be inserted in the Stack and describes the behavior of any possible sequence of operations.

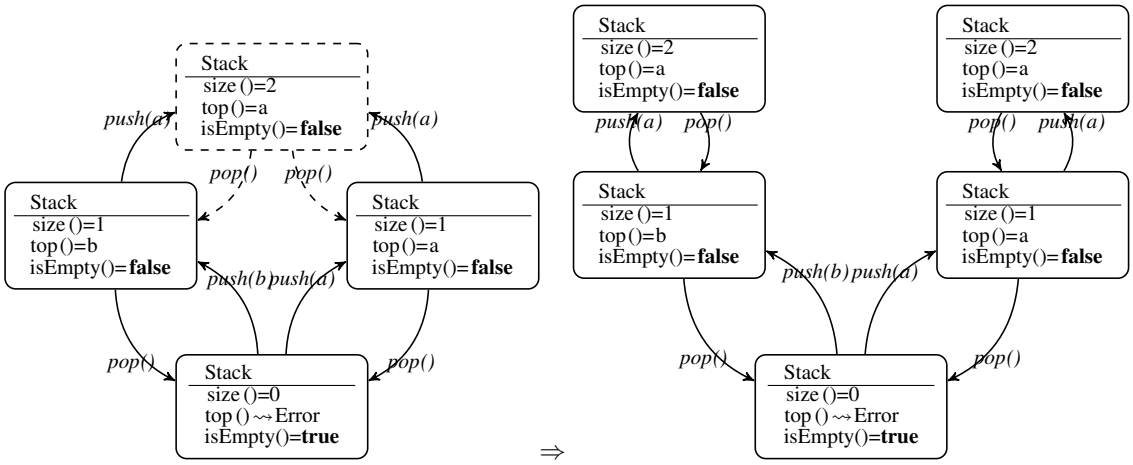
3. Recovery of Intensional Behavior Models

After a brief introduction of the formal notation we use in the specification, from now on we focus on the recovery approach. First, in Section 3.1, we discuss the recovery of BEMs; then, in Section 3.2, we illustrate how canonical terms can be found by means of searching a shortest-path tree on the BEM; finally, in Section 3.3, we show how to infer intensional behavior models by generalizing the recovered BEM.

3.1. Recovering BEMs

To generate a BEM of a given data abstraction we first need an instance pool for every formal parameter of any method. Second, we require the user to set an upper bound for the number of states of the behavior model: when the upper bound is reached, the algorithm stops producing new states and tries to complete just the remaining missing transitions. For example, the BEM of the stack shown in Figure 4 is built by specifying an upper bound of 7 states.

The BEM is built by invoking the methods of the class, starting from the constructors. For each constructor, we extract the return values of observers and produce states reachable from constructor transitions. Each next step extends the BEM by observing the behavior of objects represented by a previously introduced new state. Modifer methods are applied (with parameters taken from the appropriate instance pool) to all objects of the equivalence class denoted by that state; then, observers are applied to retrieve the observable part of the state and thus possibly build new states, until the upper bound is reached. Note that if the observable part of two objects' state is the same, the objects do not necessarily belong to the same behavioral equivalence class. This issue is taken up next on the example. Also note that



(a) Nondeterministic transitions on an ambiguous state (b) Nondeterminism resolution using a discriminating operation

Figure 6: Example of nondeterminism resolution

this discussion assumes observers to be pure. Since we do not know if an observer is pure, whenever we invoke it we must rebuild the original instance obtained by the method invocation, which might have been modified by the previous observer invocation. If two instances produced by different invocations exhibit the same observer return values, they are initially assigned to the same equivalence class.

Consider the Stack example in Figure 4. To build the BEM, we first specify the instance pool for the String parameter used by the $push(String)$ method: $I_{String} = \{a, b\}$. Then we begin to observe the behavior of the component by invoking the constructor. On the resulting object we invoke $size()$, $top()$ and $isEmpty()$ to get the observers' return values. At this point we can build a new state labeled $\{size() = 0, top() \rightsquigarrow Error, isEmpty = true\}$, reachable from an initial transition labeled with the constructor. If on that object we now invoke the two possible $push(String)$ operations, $push(a)$ and $push(b)$, we reach two different states w.r.t. the observer return values, identified as $\{size() = 1, top() = a, isEmpty = false\}$ and $\{size() = 1, top() = b, isEmpty = false\}$. In both cases, by invoking the $pop()$ operation we reach again the empty stack state. Instead, if we further invoke the $push(a)$ operation on the objects which expose the states denoted by $size() = 1$, we obtain two objects which exhibit the same observable part $\{size() = 2, top() = a, isEmpty = false\}$. We assign them temporarily to the same equivalence class, because if no other method is invoked, the two objects are indistinguishable.

However, if we invoke the $pop()$ operation on the two objects, we would face nondeterminism, since both the state denoted by $\{size() = 1, top() = a, isEmpty = false\}$, and the state denoted by $\{size() = 1, top() = b, isEmpty = false\}$ are reachable with the same method invocation (see Figure 6a). The manifestation of nondeterminism indicates that the objects whose observable state is $\{size() = 2, top() = a, isEmpty = false\}$ do not belong to the same behavior equivalence class. In fact, by invoking observers after a $pop()$ operation we would discover a part of the hidden state of the component, which is different for the two objects.

In other words, $pop()$ is a *discriminating operation* for the state denoted by $\{size() = 2, top() = a, isEmpty() = false\}$; that is, after the application of $pop()$ it is possible to verify if two objects that previously exposed the same observer return values are indeed behaviorally equivalent. In this case, the states reached after the $pop()$ operation correspond to distinguishable objects. Figure 6b shows how nondeterminism can be resolved, by deriving a behavior model where each state represents behavior equivalence classes.

Since SPY assumes that an upper bound on the number of states is initially given, in general it is not possible to discover all the different classes of behaviorally equivalent objects. In fact, the limitation implies that some states are incomplete, and the missing transitions could be used to discriminate between behaviorally different instances. This problem uncovers an important feature of BEMs: in general, transitions originating from incomplete states are *untrustable*, since the state may represent different classes of behaviorally different instances of the data abstraction.

3.2. Identifying Canonical Terms

As we discussed in Section 2.2, canonical terms are used to label BEM states. CTs represent behaviorally equivalent classes of instances of a data abstraction, and they can be defined by a suitable language $\mathcal{L}$. In principle, $\mathcal{L}$ should be inferred separately and then used to label BEM states

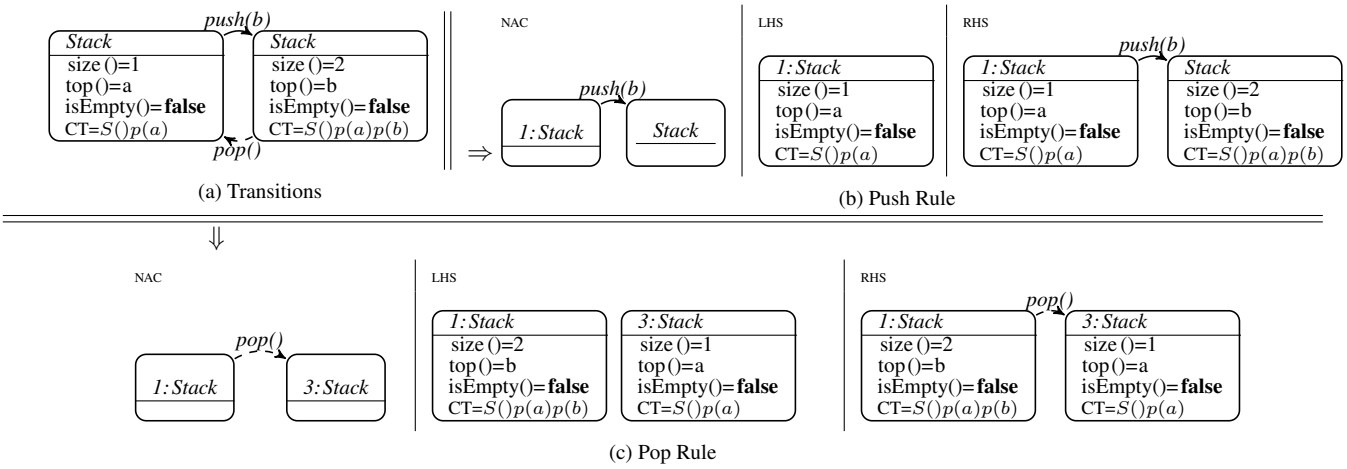


Figure 7: Examples of Inference Basis Construction ($\Rightarrow$ is the construction step; in CTs, $Stack()$ is abbreviated to $S()$ and $push(..)$ to $p(..)$)

with canonical terms. However, as we saw, the canonical language is implicitly defined by the assignments to CT attributes in the RHS of canonical rules. Thus, we can treat CTs as the other observers and we can generalize them in the inference step discussed in Section 3.3. In this step SPY must simply identify a set of CTs to label the states of the BEM recovered in the previous step. These terms must satisfy the two properties of canonical languages that we illustrated previously.

Topologically, a path on the BEM starting from the $null$ state corresponds to a term obtained by concatenating the corresponding sequence of arc labels. Any path would generate a term that satisfies property (ℓ_1) of the canonical language (i.e., it corresponds to a sequence of method invocations starting from a constructor). Property (ℓ_2) can be satisfied by searching a *shortest-path tree (*SPT*)* rooted in the $null$ state.

In fact, suppose that a positive cost is assigned to every transition of the BEM. An SPT is a tree such that, given the null state and any other state S, the path from the null state to S in the tree is the shortest path (minimum cost) on the BEM. A topological property of SPT is that the shortest path from null to a state x is always built from the shortest path to a state y reachable from x. Thus, if we label the BEM states by using the paths from a SPT, the corresponding terms will satisfy the second property of the canonical languages.

Since BEMs are directed graphs, given a cost assignment to every transition, we can use the well-known Dijkstra's algorithm [9] to build a SPT. In principle, any cost function for BEM transitions could be used since any SPT produces CTs which satisfy (ℓ_1) and (ℓ_2). Practically, the ease of generalization to a significant canonical language, which is performed by SPY in the next step, might depend on which transitions are selected.

To address this issue, we searched for a reasonable heuristic cost function for BEM transitions. A natural appraoch to assign a cost value to a transition is to define a *distance* between the abstract objects represented by the source and target state. Precisely, we define a distance by extending the notion of *object distance function (ODF)* [6]. ODF is defined on the internal representation of the object and captures how much the objects differ. Intuitively, in our extension, since we use a black-box approach, we apply the distance to the observable part of the state. An extension to Dijkstra's algorithm, which introduces further heuristics, is then applied to the BEM decorated with cost values. A full account of the computations of costs and of the extended SPT algorithm cannot be given here for space reasons[1].

3.3. Synthesizing Intensional Rules

The process of synthesizing graph transformation rules as intensional behavior models is composed of three different steps. First SPY builds a set of simple rules from each transition of the source BEM. Then it computes invariants for each rule in the inference basis. Last, rules are generalized if they express the same behavior.

3.3.1. The Inference Basis. In this step SPY produces the inference basis, that is, a set of non-generalized rules from each transition of the recovered BEM. Each rule in this set describes only the behavior of a single transition.

Transitions on the source BEM are classified according to the SPT. If the transition is part of the SPT, we build a corresponding canonical rule (i.e., a rule which introduces a new state in a BEM). Consider the BEM fragment shown

[1]Details can be found at [3].

in Figure 7(a). The solid transition is a *push(String)* transition, and it is part of the BEM SPT. Thus, SPY builds a new canonical rule from this transition, according to a simple construction pattern. In such a case, given a transition labeled with a canonical operation p from a state S to a state T, we build a rule which includes S in the LHS, the transition from S to T in the RHS, and a NAC which forbids that the same rule is applied twice to the same matching elements. Figure 7(b) shows the generated rule for the inference basis. Similar rules are built for every transition belonging to the SPT. At this point, the rules of the inference basis are able to produce the spanning tree, if applied to a graph containing the null state.

To proceed further, SPY considers the transitions which correspond to non-canonical operations in the source BEM. Since new states are only generated by canonical rules, the rules specifying the behavior of the methods that label these transitions simply add new edges to an existing graph. For example, consider the dashed transition in Figure 7(a), corresponding to a *pop()* method invocation. LHS contains a pair of states (the source state of the transition and the target state), and the RHS simply adds the *pop()* transition.

3.3.2. Computing invariants. In this step SPY elaborates each rule and transforms it into a form that is suitable for the next step, which performs actual inference through generalization. Each rule is transformed by (1) making it symbolic, that is, assigning symbolic values to observer return values and (2) identifying a set of invariant properties that constrain symbolic values. Invariant properties are identified by following a similar approach as DAIKON [12], with some changes that will be described later.

For both canonical and non-canonical rules, we can distinguish a source state S and a target state T. For canonical rules, S is the state in the LHS and T is the newly introduced state in the RHS. For non-canonical operations, S is the state from which the generated transition exits and T is the state in which it enters. Let $m(\bar{\pi}_1, \bar{\pi}_2, \ldots, \bar{\pi}_k)$ be the operation that labels the transition from S to T, and let $o_1(), o_2(), \ldots, o_n()$ be the observer methods[2]. The symbolic transformation applied to the rule replaces the concrete values returned by the observers with symbolic ones: let σ_i, τ_i be the symbolic observer return values for $o_i()$ for states S and T, respectively. Let σ_{CT} and τ_{CT} be the symbolic variables for the CTs of S and T. State S is now labeled with $\{o_1() = \sigma_1, o_2() = \sigma_2, \ldots, o_n() = \sigma_n, CT = \sigma_{CT}\}$ and T is labeled with $\{o_1() = \tau_1, o_2() = \tau_2, \ldots, o_n() = \tau_n, CT = \tau_{CT}\}$. Similarly, concrete values of parameters of method $m(\ldots)$ are replaced by symbolic variables $\pi_1, \pi_2, \ldots, \pi_k$. We keep track of concrete values by associating a function $range_R(\chi)$ to each symbolic variable χ, which maps it to the set of observed values.

[2]For simplicity and space reasons we assume here that observers do not have parameters. SPY, however, can deal with them.

Table 1: Symbolic Patterns for Canonical Terms

Type	Pattern	Description
Boolean	$contains(t, o)$	if term t contains operation o
param	$param(t, i, j)$	returns j^{th} param of i^{th} operation in t
term	$removeAll(t, o)$	removes all instances of o
term	$append(t, o)$	appends o in t
term	$appendAt(t, i, o)$	appends o in t at position i
int	$length(t)$	length of t
int	$count(t, o)$	number of times o is contained on t

Symbolic variables are then used to extract a set of invariants which describe the behavior of the rule. Such invariants are equalities in the form: $\tau_i = f(\sigma_1, \ldots, \sigma_n, \sigma_{CT}, \pi_1, \ldots, \pi_k), \forall 1 \leq i \leq n$. SPY is equipped with a set of symbolic patterns for functions. It instantiates symbolic patterns with variables $\sigma_1, \ldots, \sigma_n, \sigma_{CT}, \pi_1, \ldots, \pi_k$ and generates all equality invariants that are satisfied by replacing variables with the correspoding observer return values. At the end of this step, each rule R is in symbolic form and has an associated set of invariants I_R. Table 1 shows some of the symbolic patterns which involve canonical terms.

3.3.3. Rule Synthesis. This step generalizes the inference basis by merging symbolic rules which express the same behavior. Let us consider two rules R_1, R_2 with inviariant sets I_{R_1} and I_{R_2}, which express the behavior of the same operation m. If $I_{R_1} \cap I_{R_2}$ contains at least one invariant for every $o_i()$ and for CT, then R_1, R_2 are considered *compatible*, and they can be merged. The effect of merging consist of (i) eliminating one rule (say, R_2), (ii) associating the new invariant set $I_{R1} \cap I_{R_2}$ to R_1, and (iii) updating the range functions of R_1 symbolic variables, that is, for every variable χ, $range_{R_1}(\chi) = range_{R_1}(\chi) \cup range_{R_2}(\chi)$.

For example, let R_1 be the rule in Figure 7(b), and let R_2 be a similar rule, with identical LHS, describing the behavior of a *push(a)* operation, thus introducing a new state labeled with $\{top() = a, \ldots, CT = Stack()push(a)push(a)\}$. The two rules are first transformed into the symbolic form. The invariant set I_{R_1} contains $\tau_1 = \pi$, where τ_1 is the symbolic value of the $top()$ observer in the target state of the transition, and π is the symbolic value of the parameter of *push(...)*. I_{R_2} includes the previous invariant, but also an accidental one, $\tau_1 = \sigma_1$, where σ_1 is the symbolic value of the $top()$ observer in the source state. Such invariant states that the $top()$ observer does not change its return value after a *push(...)* operation, which is accidental (it is due to the specific use of a). $I_{R_1} \cap I_{R_2}$ contains at least $\tau_1 = \pi$; thus, the rules are com-

patible and the merge discards $\tau_1 = \sigma_1$.

This approach is repeated for all pairs of rules describing the behavior of a given operation until no mergeable pairs exists. The approach is identical for non-canonical rules; we start by merging compatible rules until no compatible rule pair is present in the set of rules. At the end, we have a reduced set of rules with equality invariants describing their behavior. Many of the accidental invariants computed for each rule in the inference basis step are removed by the rule merging. A simple rewriting substep relabels T-states in canonical rules in the form $\{o_1() = f_1(\ldots), \ldots, CT = f_{CT}(\ldots)\}$, where f_i corresponds to the syntactically shortest equality invariant for observer $o_i()$ in the invariant set. For non-canonical rules, the equality invariants are simply considered as application conditions of the rule, since S and T are both included in the LHS.

Finally, we can extract the likely application conditions of each rule. They are a set of predicates involving only $\sigma_1, \ldots, \sigma_n, \sigma_{CT}, \pi_1, \ldots, \pi_k$, and they are discovered with the same methodology described before, that is, by considering only functional patterns returning booleans, and range functions of symbolic variables.

4. Related Work

The techniques for recovering specifications from an existing application can be roughly classified in two categories. Specifications may be extracted through *static analysis* of code or through *dynamic analysis*. In the former case, classical program analysis techniques are used to extract a higher level view of the component behavior [21, 23]. Static analysis is a *white box* technique, since it assumes the source code to be available.

Dynamic analysis techniques are based on observing the execution traces and abstracting them by producing some higher level description. Although the field is relatively new, a number of approaches have already been explored, such as [12, 8, 16, 5]. Dynamic analysis techniques can be *black box* if the run-time observations are only made by observing the values that flow in and out when public methods are called. SPY focuses exactly on this approach.

A few approaches have been developed for specification recovery of data abstractions implemented as JAVA classes. HEUREKA [16] can retrieve *algebraic specifications*, by finding equations among sequences of operations, and then generalizing them by substituting concrete variables by universally quantified free variables, thus producing axioms. HEUREKA uses a black-box technique based on dynamic analysis: the tool can only observe the external behavior of the component whose specification must be inferred. In order to produce equations, the tool checks for behavioral equivalence.

A different approach to specification recovery of stateful components is based on *behavior models*. We already mentioned ADABU [8], which can infer nondeterministic finite state machines describing sequences of the operations. Each state is identified by a predicate on the return values of pure observers, and each transition corresponds to the application of a modifier. The identification of each state with a limited part of the observable state of the component leads to an overapproximation of the behavior of the component.

Other approaches, such as [25], can provide more precise behavior models. In [13] we show how to infer more precise behavior models, and we use them to improve the performance of HEUREKA. In both these approaches, the abstraction function on the observers' return values is not fixed but can be customized by the user. Despite this improvement, all of these approaches still identify each state with nothing more than the complete observable part, thus leading to approximations of the same kind of ADABU. Objects which expose the same return values of the observers might still be behaviorally different: for this reason, even these more precise models may include nondeterministic transitions.

In the end, HEUREKA (and the optimization we proposed in [13]) stands as the best possible competitor of our approach. The next section provides a quantitative assessment of our method with respect to HEUREKA.

5. Empirical Assessment

In a traditional setting, the problem of comparing a specification against an implementation assumes that the specification provides prescriptive description of the correct behavior that an implementation must satisfy. The goal of *verification* is to check the implementation against possible violations of the specification. In the case of specification recovery, the issue is reversed: we must verify that the recovered specification is correct with respect to an existing implementation. In the case of black box recovery, like SPY and HEUREKA, we can invoke the implementation and use it as an oracle to check if inferred behavior is exposed by the implementation. Ideally, a recovered specification is a correct description of an implementation if it is able to give a correct answer for all the possible behaviors. Practically, for each possible behavior given in terms of a predicted return value for a sequence of method invocations ending with an observer, the recovered specification might (i) correctly describe the behavior; (ii) predict a wrong behavior, or (iii) be unable to give an answer.

In general, any inference method makes assumptions under which it can generalize its findings from a number of observations. It assumes that the inference basis is *significant* with respect to the global behavior of the component; that is, the observed behaviors expose sufficient information to capture all behaviors of the component. For example,

Table 2: Empirical Results

	Class	Inference Basis	# axioms	# rules	Testing Basis	$!T_{Heureka}$	$!T_{SPY}$	$?T_{Heureka}$	$?T_{SPY}$
Relevant ADTs	Stack	67789	6	5	$0.6 \cdot 10^6$	0%	0%	0 %	0%
	Queue	58452	7	5	$0.6 \cdot 10^6$	0%	0%	0 %	0%
	Min-MaxSet	188462	12	6	$1.9 \cdot 10^6$	0%	0%	35.38 %	0%
	MTS	$2.2 \cdot 10^6$	40	11	$1.3 \cdot 10^7$	0 %	0%	45.80 %	0%
	SymbolTable	70347	18	8	$7.2 \cdot 10^6$	0%	0%	59.03 %	0%
java.util Classes	Stack	118516	12	5	$1.3 \cdot 10^6$	0%	0%	0 %	0%
	ArrayList	866663	107	14	$1.3 \cdot 10^6$	0%	0%	31.20 %	0%
	ArrayDeque	$0.9 \cdot 10^6$	142	23	$1.3 \cdot 10^7$	0%	0%	32.15 %	0%
	TreeSet	499316	89	14	$9.2 \cdot 10^6$	0%	0%	64.32 %	0%
	TreeMap	397178	128	13	$8.4 \cdot 10^6$	0%	0%	36.41 %	0%
	LinkedList	$1.1 \cdot 10^6$	139	27	$1.2 \cdot 10^7$	0%	0%	43.24 %	0%
	PriorityQueue	334071	52	13	$3.4 \cdot 10^6$	0%	0%	44.81 %	0%
	Amazon SQS	20760	7	5	41530	0%	0%	0 %	0%

SPY assumes that the behavior of the component exposed by the use of parameters from the instance pool can be generalized to any possible value for the corresponding actual parameters of methods. This means that the behavior of the component does not change significantly for unobserved behaviors given by parameters outside the instance pool. Second, it assumes a continuity property; that is, the behavior observed during the generation of the BEM, in terms of state transitions and modification of observer return values, is generalizable also for unobserved behavior. These assumptions are not SPY specific, but they are critical for every specification recovery approach: if they do not hold, generalizations predict wrong behaviors. The inference basis also influences the complexity of our approach, which is function of the number of BEM transitions and states.

Let us consider the problem of comparing two specifications S_1 and S_2, coming from two different specification recovery methods M_1 and M_2, against the same implementation I. First, for the comparison to be fair, M_1 and M_2 should use the same significant inference basis. Second, since it is impossible to check S_1 and S_2 against every possible behavior of I, we need to select a common set of test cases, called the test basis, to compare the recovered specifications; for the check to be fair, test case selection should not depend on the method. In the end, S_1 is "better" than S_2 w.r.t. I if the number of test cases giving a correct answer is greater for S_1 than for S_2. To resolve the first issue (fair inference basis), we use the same instance pools to recover both algebraic specifications with HEUREKA and intensional behavior models with our approach; moreover, we set the number of states in the recovered BEM in a way such that the total number of test cases in the inference basis with our approach is strictly contained in the one used by HEUREKA. Second, to obtain a fair test basis, we randomly generate a set of instance pools to test the recovered specifications and then generate exhaustively a set of terms, ending with an observer, up to a certain length of method invocations.

We applied both HEUREKA and SPY to three different sets of classes implementing data abstractions. The first set includes simple data abstractions, such as Queue, Stack, and more complex and critical ones, such as MinSet, MaxSet, SymbolTable and Majster's Traversable Stack (MTS) [19]. MinSet and MaxSet are simple ordered sets exposing only one observer, returning the minimum and the maximum element, respectively: these abstractions are important to check the benefits of our approach in cases where the exposed part of the internal state is very limited (i.e., just an observer). Another tested data abstraction is the MTS, which is essentially a stack with an operation to iterate on the contained objects. The second set of data abstraction is taken from the java.util package from the OPENJDK 7 project [2], and includes almost all collections from the package. Third, we applied our method to a web service, the Amazon Simple Queue Service (SQS) [1]. Table 2 summarized the empirical results of the comparison between HEUREKA and SPY. Details on the inference basis, test basis, results, and comparisons can be found at [3]; the complete recovered specifications are also omitted for space reasons, but they are included in the website. Column 1 lists the class name; column 2 contains the number of test cases used by both methods as inference basis. Column 3 and 4 give a hint on the conciseness of the recovered specifications in terms of the number of inferred axioms or rules. Column 5 contains the number of test cases used to check the adherence of the recovered specification to the class implementation. The last four columns share the following format: for each method $x \in \{\text{SPY}, \text{HEUREKA}\}$, $!T_x$ contains the percentage of test cases on which the specification recovered by x predicted a wrong behavior, while $?T_x$ contains the percentage of test cases on which the specification was unable to give an answer. The lower those values are,

the better the specification adheres to the class implementation. By careful inspection of Table 2, the reader can see that, under the aforementioned assumptions, SPY performs consistently better than HEUREKA.

6. Conclusions and Future Work

This paper describes SPY, a new approach to recovering specifications for data abstractions implemented as JAVA classes. SPY infers models by following a predefined black-box strategy that generates method invocations through which the component is exercised and observes input/output relationships. We described how SPY infers an intensional generalization of behavior models via graph transformation that specify the behavior of the data abstractions. The SPY method is based on the empirical observation that in many practical cases components follow a kind of "uniform behavior" law. This of course cannot be proven in general, but it held in most practical cases for which we made experiments. SPY is supported by a prototype tool that can be downloaded to experiment with black-box specification recovery. The prototype has been used for an assessment of the method for non trivial data abstractions, collections from the JAVA library, and the Amazon Queue Service. In all these cases, we observed that the method infers high quality specifications, far more precise than those that could be inferred by previously published methods. Future work will continue to extend the SPY approach to other kinds of software components, especially in the context of open an dynamic environments, such as SOAs.

Acknowledgments

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

[1] Amazon Simple Queue Service. http://aws.amazon.com/sqs/.

[2] OpenJDK project. http://jdk7.dev.java.net/.

[3] Spy website. http://home.dei.polimi.it/mocci/spy/.

[4] L. Baresi, C. Ghezzi, A. Mocci, and M. Monga. Using graph transformation systems to specify and verify data abstractions. In *Proc. of GT-VMT'08*, Electronic Communications of the EASST, Volume X (2008), 2008.

[5] L. C. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. In *Working Conf. on Reverse Engineering*, pages 57–66, 2003.

[6] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *RT'06: Int. workshop on Random testing*, pages 55–63, New York, NY, USA, 2006. ACM.

[7] C. Csallner, N. Tillmann, and Y. Smaragdakis. DySy: dynamic symbolic execution for invariant inference. In *Int. Conf. on Software engineering*, pages 281–290, New York, NY, USA, 2008. ACM.

[8] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with Adabu. In *Int. Wks. on Dynamic Analysis*, May 2006.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

[10] R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[11] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in TCS. Springer, 2005.

[12] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D. thesis, University of Washington, Seattle, Washington, Aug. 2000.

[13] C. Ghezzi, A. Mocci, and M. Monga. Efficient recovery of algebraic specifications for stateful components. In *Int. Wks. on Principles of Software Evolution*, Dubrovnik, Croatia, Sept. 2007.

[14] J. V. Guttag and B. Liskov. *Program Development in Java: Abstraction, Specification and Object-Oriented Design*. Addison-Wesley, 2001.

[15] D. Harel and E. Gery. Executable object modeling with statecharts. In *Int. Conf. on Software Engineering*. IEEE, 1996.

[16] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Trans. Software Eng.*, 33(8):526–543, 2007.

[17] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.

[19] M. E. Majster. Limits of the "algebraic" specification of abstract data types. *SIGPLAN Notices*, 12(10):37–42, 1977.

[20] B. Meyer. Design by Contract: The Eiffel Method. In *Int. Conf. on Technology of Object-Oriented Languages and Systems*, 1998.

[21] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Int. Conf. on Software Engineering*, pages 338–348, 1996.

[22] M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer-Verlag, 1977.

[23] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *International Conference on Software Engineering*, pages 254–263, 2005.

[24] G. Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Application of Graph Transformations with Industrial Relevance*, volume 3062 of *LNCS*, pages 446–456. Springer, 2004.

[25] T. Xie, E. Martin, and H. Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *Int. Conf. on Software Engineering, Research Demos*, pages 835–838, May 2006.