Efficient Recovery of Algebraic Specifications for Stateful Components

Carlo Ghezzi
DEI
Politecnico di Milano
Via Ponzio 32
Milano, Italy
carlo.ghezzi@polimi.it

Andrea Mocci
DEI
Politecnico di Milano
Via Ponzio 32
Milano, Italy
andrea.mocci@mail.polimi.it

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

ABSTRACT

Specification recovery is a necessary step of many reverse engineering and reuse efforts. This paper deals with recovering the semantic part of a component's interface. It focuses on stateful components that provide data abstractions. Recovery is achieved by following a black-box strategy, i.e. by observing the component's dynamic behavior. Among the published approaches, `Heureka` recovers algebraic specifications from Java classes. Another approach (`Adabu`) recovers behavioral models. The work we describe here adapts the latter, which provides an approximate semantic description for the class, to significantly optimize the former. The resulting approach, called `Adiheu`, is described in the paper with a preliminary assessment.

1. MOTIVATIONS

An interface specification for a software component describes what a component can do on the client's behalf. It is composed of two parts. The *syntactic part* describes the component's signature, i.e., the syntactic form of how the component may be used by the clients. The *semantic part* describes the visible effects for the clients, which may obtained as a result of using the component through its interface. In an ideal world, the semantic part is associated with the component in terms of formal documentation, for example described in some notation, like the *Java Modeling Language (*JML*)* [10] in the case of Java components. JML is a specification language that follows the design by contract approach of `Eiffel` [11] using Hoare-like preconditions, postconditions and invariants. In practice, the formal documentation is often given informally. Sometimes, it is totally absent, and even inconsistent with the actual implementation.

The problem of recovering a specification from an existing implementation arises in many practical contexts. A typical case occurs during software evolution, when we need to reconstruct a high-level view of a software component as we become engaged in a reverse engineering activity. Other possible cases may occur when we examine existing components as candidates for possible reuse within an application. Even though the component might have some associated semantic description, we often do not know how faithful it is and thus it would be useful to extract from the component some formal interface description that may support its reuse on more reliable and sounder grounds. Finally, there are cases in which we cannot even examine the internals of the component to extract information. All we can do is to observe the run-time behavior. This happens, for example, in the case of a service oriented architecture (SOA), such as web services, where services are exposed by service providers to be used by service users.

The techniques for recovering specifications from an existing application can be classified in two categories. Specifications may be extracted through *static analysis* of code or through *dynamic analysis*. In the former case, classical static analysis techniques are used to extract a high-level view of the component's behavior [14, 12, 13]. In the latter case, the component is examined as it behaves at run-time; i.e., by observing the values provided as input data and computed as results, to infer possible high-level views of the component [5, 3, 9, 1]. Of course, the latter technique may be the only one available in the case of web services.

The techniques can also be classified along another dimension, which distinguishes between stateless and stateful components. A *stateless component* behaves as a procedural abstraction [7] and its interface specification may be given in terms of pre- and post-conditions. The behavior of a *stateful component*, instead, depends on its state, which is often encapsulated and not visible directly through the component's interface. Understanding stateful components is often harder, because the effect of certain operations may depend on the history of previously executed operations.

In this paper we focus on a specific class of stateful components that define *data abstractions* [7]. A component is thus an implementation of an *abstract data type*, whose internal state is hidden to the clients and made available only through operations. A typical example is a Java collection, such as a queue, a set, etc. We also focus on recovering a specification by applying dynamic analysis.

Specification inference via dynamic analysis is a relatively new field of research. Interesting existing results are presented in [5, 3, 9]. [9] and [3] explicitly focus on specification inference for data abstractions. The method described in [9] (called `Heureka`) infers algebraic specifications of abstract data types, while [3] (called `Adabu`) infers nondeterministic finite state machines which describe legal sequences of the operations. Our work combines the two methods and shows the benefits that can be achieved through such combination. More precisely, we propose a method, called `Adiheu`—*ADabu Improves HEUreka*, in which we adapt [3] to be used in combination with [9] to provide an optimized method to infer algebraic specifications.

The paper is organized as follows: Section 2 provides background concepts and clarifies the terminology. Section 3 illustrates the `Heureka` approach, defined by [9] for discovering algebraic specifications of a component implementing a data abstraction. Section 4 shows our method which improves `Heureka` inspired by `Adabu`, defined by [3]. Section 5 provides an initial assessment of our proposal and shows the improvements we obtained with respect to `Heureka`. Finally, Section 6 contains some conclusions and outlines directions for future research.

2. BACKGROUND CONCEPTS AND TERMINOLOGY

This section provides a brief overview of the background concepts underlying our approach and the terminology used throughout the paper. A reader who is familiar with them may safely skip the section.

We focus on components that provide data abstractions by encapsulating and hiding the local data attributes and exporting only the methods through which objects may be accessed. More precisely, components are Java classes that implement an abstract data type. Class methods can be classified according to the following roles:

- *constructors*: they produce a new instance object.
- *observers*: these methods inspect the internal state of an object, by returning values through which they expose information about the internal state.
- *mutators*: these methods change the internal state of an object.

It is possible for a method to play more than one role: typically, it can behave both as a mutator and as an observer at the same time. For example, Fig. 1 shows the interface of a Java ObjectStack class. Method ObjectStack() is a constructor; methods isEmpty() and size () are observers, method push(Object) is a mutator and method pop() is both an observer and a mutator. Method pop() is also called an *impure* observer, because it has side effects, as opposed to size (), which is *pure*.

There are two different notions of equality among objects defined by an abstract data type: *behavioral equivalence* and *structural equivalence*. Behavioral equivalence defines two objects as equivalent if they cannot be distinguished by applying to them any sequence of operations. Structural equivalence defines two objects as equivalent if their internal representation is the same. If two objects are structurally equivalent they are also behavioral equivalent, but the converse is not necessarily true. In this paper we consistently use the concept of behavioral equivalence. For brevity, we will thus use "equivalence" as a synonym for behavioral equivalence. If needed, we explicitly use the term "structural equivalence" otherwise.

```
public class ObjectStack {
    ..

    public ObjectStack() { .. }

    public void push(Object element) { .. }
    public Object pop() { .. }

    public boolean isEmpty() { .. }
    public int size() { .. }

}
```

Figure 1: The public interface of ObjectStack

The formal specification of an abstract data type implemented by a Java class may be given by using different notations. Since Java classes are stateful components, the key issue is how to specify the behavior of public methods, which depends on the state, without disclosing the hidden encapsulated state. The approach adopted by JML generalizes pre- and post-conditions by using abstract models. Algebraic specifications, which were initially proposed and investigated by [8, 6] and are supported by languages like [2, 4], take a different approach. The hidden state is implicitly taken into account by specifying axioms on sequences of operations.

An algebraic specification has two parts: the *algebraic signature* and the *set of axioms*. The algebraic signature defines the types used in the algebra (*sorts*) and the signatures of operations. Each axiom is a universally quantified formula expressing an equality among terms in the algebra. For example, Fig. 2 shows the algebraic specification for the class ObjectStack described in Fig. 1 [1].

3. DISCOVERING ALGEBRAIC SPECIFICATIONS

Henkel and Diwan proposed a tool for discovering algebraic specifications of Java classes [9]. The tool, named `Heureka,` supports the task of writing formal documentation for reusable components, such as containers, in which the use of algebraic specifications is particularly powerful and effective.

In order to define a proper algebraic signature for a Java class, `Heureka` maps each Java class to a sort with the same name; each method belonging to the public interface of a

[1]For simplicity, we assume that a **null** object is returned by a pop() operation on an empty stack, and that **null** objects cannot be pushed onto a stack.

sorts:

ObjectStack, Object, Boolean

operations:

ObjectStack	$\rightarrow ObjectStack$
push	$ObjectStack \times Object \rightarrow ObjectStack$
pop	$ObjectStack \rightarrow ObjectStack \times Object$
isEmpty	$ObjectStack \rightarrow Boolean$

axioms:

$$isEmpty(ObjectStack()) = true$$
$$isEmpty(push(s, e)) = false$$
$$pop(push(s, e)) = < s, e >$$
$$pop(ObjectStack()) = < ObjectStack(), null >$$

Figure 2: An algebraic specification of a stack of elements

class is mapped to an operation between the related sorts. Each operation which is not a constructor has a right hand side (codomain of the operation) which is the Cartesian product of the sort whose specification is being constructed and the return type of the operation. This means that an implicit assumption is made that any operation can modify the current object through a side effect. Should an operation be a pure observer, additional equations will be discovered by the tool, which states equivalence between objects before and after the application of the operation. This mapping between the method signature and the algebraic signature is implemented by using queries to the Java reflection API. For example, the algebraic signature of method Boolean isEmpty() (see Fig. 1) is $isEmpty : ObjectStack \rightarrow ObjectStack \times Boolean$.

Given a class and its method signatures, `Heureka` generates a list of *terms* up to a given length. A term is a sequence of legal method applications with fixed actual parameters, starting from a constructor. The *length* of a term is the number of method applications that appear in it[2]. For example, consider the class ObjectStack shown in Fig. 1.

push(ObjectStack().state, Object@3).state

is a term of length two (expressed in the `Heureka` syntax) denoting a stack resulting from a call to push, by using as actual parameters the stack resulting from the invocation of the constructor and a constant object of the Object sort. Two terms are defined to be equivalent iff they generate equivalent objects.

The set of the generated terms is then analyzed by an *equation generator*, which tries to find equivalent terms.

The tool may discover three kinds of equations:

- *State equations*: A state equation represents an equivalence between two states generated by the execution of two different mutator terms. For example:

  pop(push(ObjectStack().state, Object@3)).state = ObjectStack().state

- *Observer equations*: These equations represent the equality between the return value from an observer term and a constant. For example:

  size (push(ObjectStack().state, Object@3)).retval = 1

- *Difference equations*: These equations represent a constant difference between the return values from two observer methods. For example:

  size (ObjectStack().state). retval + 1 = size (push(ObjectStack().state, Object@3)).retval

State equations are generated by verifying equivalence between terms according to an algorithm described in [9]. Intuitively, for each pair of terms to be compared, the algorithm generates a sequence of zero or more *stub* operations which are applied to both terms; then the algorithm evaluates equivalence by comparing the results of applying every observer.

Equations are then generalized by replacing subterms with universal quantifications among typed variables, thus producing the set of axioms. Finally, each axiom is checked with several automatically generated test cases to determine if it is likely to hold; the resulting set of axioms is then polished to eliminate redundancies.

The term generation part exploits two main optimization techniques. First, it uses static analysis to identify side-effect free operations (pure observers)[3]. Second, to reduce the term set, it generates only the minimal term for any generated object representation. If two terms generate structurally equivalent objects (evaluated by object serialization) only the shortest one is kept. The minimal term generation is also used to build up equations. When a term generates an object which is structurally equivalent to an object generated by a previous term, a corresponding equality equation between the two terms is generated.

4. OPTIMIZING DISCOVERY

In its search for equivalence, `Heureka`'s equation generator is completely blind: it only knows about the method signatures, but no behavioral information is used. In order to improve performance we should try to reduce the number of terms to be compared for behavioral equivalence. The next section describes an approach (inspired by [3]) which, given the maximum length of terms to generate, significantly reduces the number of terms to be compared in order to produce equations. A further optimization concerns the procedure described by [9] which checks for equivalence of terms. These optimizations were suggested by the approach followed by `Adabu` [3], a tool that infers a behavioral model for a Java class. The next section provides an introduction to `Adabu`. Our optimizations are described in sections 4.2 and 4.2.2.

4.1 Mining Object Behavior with Adabu

`Adabu` [3] is a tool designed to infer the *object behavior models* of Java classes. Each class is modeled by a non-deterministic

[2]Actually, Heureka uses a different definition of length that also implicitly depends on the parameters (see [9]). Our approach takes this notion into account, but we prefer to explain the way it works by using a simpler definition.

[3]This optimization can be performed only if the internals of the component are accessible.

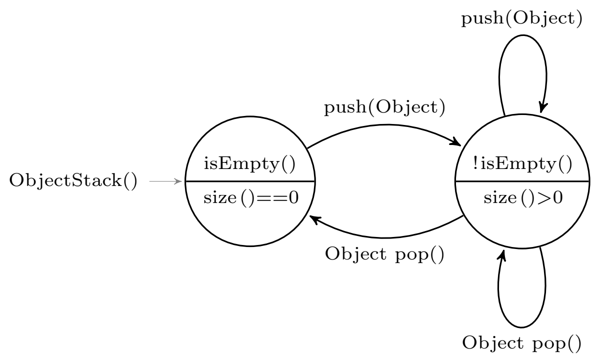

Figure 3: The Adabu model of the ObjectStack class.

finite state automaton which captures the relationship between mutator and pure observer methods. The strong assumption behind this tool is that *common behavior is often correct behavior*; thus, the models derived by observing common behaviors represent *universal likely invariants*.

The generated automaton consists of a set of semantically-labeled states and transitions. Each state corresponds to an abstracted set of possible return values of pure observer methods; each label on a transition corresponds to a mutator application. Initial states are reached from constructor applications. The automaton provides a semantic description of the component's interface. The behavior model of the ObjectStack class in Fig. 1 is shown in Fig. 3.

As the model shows, after a call of the push(Object) method, the observer isEmpty() always returns **false**. Note that models are built by observing the real, dynamic object behaviors, i.e. by executing observers after manipulating objects via mutators. For instance, the fact that in the isEmpty() state there is no exiting transition labeled pop() does not strictly mean that this operation is *illegal* or *exceptional* according to the specification and implementation of the class; it simply means that since the method invocation has not been observed during a concrete use of the class it surely is an uncommon operation, and according to the normal behavior assumption, we might just assume that it is *reasonably* illegal.

The model of the object behavior is built through the following steps:

1. *Pure Observer Detection*: This step recognizes methods that behave as pure observers. Pure observers must have a non-void return type; they must be side-effect free, and have no parameters. It is possible to use a static analysis tool to identify methods that do not change the state of the object. Any other method is considered to be a mutator.
2. *Instrumentation*: After detecting pure observers, the code is instrumented to trace the state of each instance of the class. First, a new generic state-extractor method is added, which calls every pure observer method. Each mutator of the class is enriched with a call of the state-extractor method before and after the actual code of the mutator; then a call for the addition of a new transition on the model is added.
3. *Execution*: The execution of the instrumented program provides the information needed to infer the behavior models.

Given a sequence of observer methods $(o_1, \ldots, o_n)$, a *concrete state* is a vector of the corresponding return values $(v_1, \ldots, v_n)$. A *trace* for each object instance of the class to be modeled consists of a sequence of triples (v_i, m_i, v'_i) where v_i and v'_i are the concrete states before and after the execution of a mutator m_i. Obviously, the possible return values of observers generate a very large number of possible concrete states. Adabu uses *abstraction* over the return values of observers to reduce the number of states. Specifically:

- For numerical values of types such as **int** or **float**, it uses three abstracted states: negative values, positive values, and 0. Thus, for methods like size(), only size()==0 and size()>0 are observed (and meaningful).
- Object references (for any kind of class) are mapped to the abstract state **null** and **instanceof** c, where c in one of the possible classes the object can be a reference of.
- Enumerations and boolean values are not abstracted, i.e. there is one state for each possible value of the object.

After executing the program, for each instance of the class to be modeled, Adabu mines an object-related state machine by abstracting the concrete state and adding transitions with regard to the observed traces. At the end, the global class model is generated by merging the single object models.

Fig. 3 shows the automaton inferred for ObjectStack. The automaton has one concrete and one abstract state, which describes all non-empty stacks. The inferred automaton is in general nondeterministic, for two main reasons: first, pure observers can only expose a limited view of the internal state; this means that two objects are considered to belong to the same state even if their internal state is different but this difference is not unconvered by the pure observers. Second, the built-in abstraction mechanism collapses a large number of concrete state into the same abstract state.

4.2 Driving term generation with a behavioral model

The behavioral models that can be inferred by Adabu provide a useful, high-level description that provides a concise semantic view of data abstractions. Algebraic descriptions, on the other hand, are more expressive[4], but harder to infer. The question we address is whether, and how, the two approaches can be combined to achieve better results than

[4]A finite state behavioral model cannot represent an unbounded stack, which can instead be described with an algebraic specification. Consequently the behavioral model like the one in Fig. 3 represents also unacceptable behaviors

each of them separately. Indeed, we will show that term generation for Heureka can be driven by a behavioral model à la Adabu. The underlying idea is that a behavioral model like the one in Fig. 3 can be used to generate terms by applying the methods corresponding to each transition on the model. The resulting approach, called Adiheu, is illustrated hereafter. We develop a new component to derive behavioral models. As opposed to Adabu, the inference algorithm implemented by our component does not require a step of static analysis otherwise needed to detect pure observers. Hence the approach does not require any knowledge of the source code of the container component.

4.2.1 First steps towards Adiheu

Our algorithm generates a set of terms, associated with each state of the automaton, up to a maximum length L. The terms associated with a state are those that that must be checked for equivalence. The algorithms works as follows: Each set is initialized to the empty set. Starting from the constructors, which are represented by the initial transitions on the automaton, we generate the terms of length 1. For each state, if the corresponding set of terms is non-empty, we generate a new term for each transition exiting the state, and we add this new term to the target state of that transition. This process is repeated until the required maximum length of the terms is reached.

If the model is deterministic, different transitions that bring to different abstract states always bring also to different concrete states. In fact, two abstract states are not equivalent when at least one observer yields a different value (since the concrete state is also different).

Using this property, the pairs of algebraic terms which are generated from transitions which bring to different abstract states are guaranteed not to generate equivalent objects. Thus, the terms belonging to the sets associated with states can be considered to be an *abstract likely equivalence class of terms (ALEC)* and we can restrict comparison of terms for equivalence within each ALEC.

If the model is non-deterministic, the same operation may bring to different abstract states. We might suspect that our method does not check for equivalence of some pairs of terms which are actually equivalent just because they belong to two different ALECs. For example, if we consider the transitions on the model on Fig. 3, the object generated by the term ObjectStack() belongs to the ALEC associated with the state isEmpty()==**true**, while object generated by the term pop(push(ObjectStack(), Object@3)) belongs to the ALEC associated with the state isEmpty()==**false**. The two terms are equivalent, but they belong to two different ALECs. This, however, is not a problem because there exists another non-deterministic transition which generates the term pop(push(ObjectStack(), Object@3)) as belonging to the ALEC associated with the state isEmpty()==**true**. This property, which has been informally explained on this example, can be proved to hold in general. In conclusion, we can safely restrict the check for equivalence to terms that belong to the same ALEC.

By reasoning just about the transitions, we ignore the semantics of states. Suppose that we generate terms by following transitions until we reach some state with exiting non-deterministic transitions. The term resulting by appending the operation labelling these transitions to a term associated with the state yields a concrete state, which can be retrieved by invoking observers on the object generated by the term. Thus, despite non-determinism, the term is inserted in the ALEC associated with only one of the states reachable by the non-deterministic transitions. Considering the example on Fig. 3, the term pop(push(ObjectStack(), Object@3)) is inserted only with the ALEC associated with the state labelled with isEmpty()==**true**.

The approach we illustrated so far has been implemented by a preliminary version of Adiheu, which fed Heureka's equation generator according to the previously described term generation algorithm. This straightforward method provided only minor improvements. In fact the really simple abstraction function used by Adabu does not provide a significant distribution of terms among states, which is instead crucial to speed up Heureka. For example, the highly abstracted state labelled with !isEmpty() and size()>0 accumulates a very high number of terms while the isEmpty() state remains with only two terms. If instead we unfolded highly abstracted states, the size of ALECs could become smaller and the equation generator much faster. This observation can lead to a significant optimization that is described in the next section.

4.2.2 Unfolding the models

In order to provide a better distribution of generated terms among abstract states, we provide a semantic unfolding of behavior models, i.e. a variable and configurable abstraction function for observers returning *numerical values.* In general, by unfolding the states of the automaton, terms are better distributed among the ALECs, thus resulting in a reduced number of comparisons to be performed during equation generation.

For instance, in the case of the ObjectStack class and the size() observer, we may choose to abstract the returned value as three possible values (zero, one, or greater than one) instead of zero or positive. This simple "unfolding" approach yields the automaton shown in Fig. 4. Similar abstractions might be provided for all numeric types.

Another useful change of the abstraction function applies to the observer methods which return non-primitive types. This can be more easily explained for a modified version of ObjectStack, called ObjectStack', obtained by adding a new pure observer method Object top() which returns the object on top of the stack, without removing it. We show later that the approach can be applied also to the original data abstraction shown in Fig.1 where pop() both returns an object and removes it from the stack, and in general for any non-pure observer.

Heureka uses an instance pool for primitive types and for Objects used as actual parameters. Considering generic containers, we can expect the object returned by the observers to be the same objects used as actual parameters of previous operations. For example, if we use an instance pool $I_{Object} = \{obj_1, obj_2, obj_3\}$ for Objects, then we can generate the following abstraction function for the Object top()

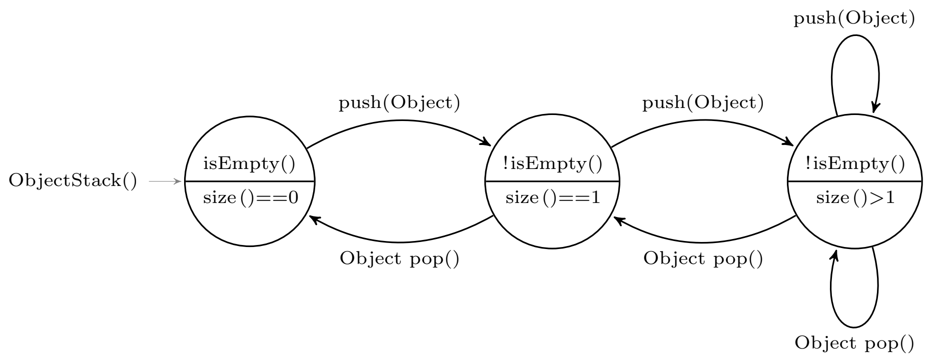

Figure 4: An unfolded version of the behavior model of ObjectStack

method: { top()==**null**, top()==obj_1, top()==obj_2, top() ==obj_3, top() $\notin I_{Object}$}. This approach leads to a significant state unfolding if the class exposes several observers returning Object, and a significant number of non-deterministic transitions. Fig. 5 shows an example of an unfolded model of ObjectStack' using an instance pool of three values for Object. Solid arcs represent push() operations and dashed arcs represent pop()s; a state labeled with $s_{i,j}$ means that in that state size()==i and top()==obj_j [5]. Transitions representing methods with formal parameters are labeled not only with the name of the method, but also with the actual parameters taken from the instance pool. For example, the transition from state $s_{1,1}$ to state $s_{2,1}$ is labeled with push(obj_1).

The case of observers with side effects can be treated as follows. After applying the method, we check whether the object has changed (by applying a simple hashing function to the serialized state). In case of change, we retrieve the state of the object before executing the impure observer by re-applying the corresponding sequence of operations that generated it. With this approach, Fig. 5 represents a behavioral model not only for ObjectStack' (with the pure observer top()), but also for the original ObjectStack data abstraction described in Fig. 1, using the impure pop() method both as a modifier and an observer. Observers with parameters can also be handled by our method, but their treatment is omitted for space reasons.

4.3 Driving the Equality Engine

In the previous section we described an approach that improves the number of terms to be compared for equivalence with respect to `Heureka`. In this section we show how the step of stub generation in `Heureka`, which was reviewed in Section 3, can also be driven by a behavioral model.

If we consider two terms to be checked for equivalence, we must check that for every possible test stub they observationally behave in the same way, by checking the result of observers. The stub check stops when a limit on stub length is reached.

[5]The isEmpty() return value is omitted for brevity because of the invariant size()==0 $\Leftrightarrow$ isEmpty() $\wedge$ size()>0 $\Leftrightarrow$!isEmpty()

With our approach, we are sure that every pair of terms to be compared belong to the same ALEC. First, we must check that the actual values returned by the observers are equal; if they are not, the two terms are not equivalent. Otherwise, we adopt a heuristic method which more likely tries to uncover non equivalent pairs of terms. To do so, the stub is created by appending first the methods labelling non-deterministic transitions exiting the state. The motivation is that the behavioral model introduces non-deterministic transitions either because the observers are not powerful enough to distinguish as non equivalent two terms that lead to the same abstract state or because the abstraction we provided in the behavior model collapses non equivalent concrete state in the same abstract state.

For example, consider the behavioral model for ObjectStack' shown in Fig. 5, and consider two terms in the ALEC associated with state $s_{2,1}$:

- push(push(ObjectStack(), Object@3), Object@1)
- push(push(ObjectStack(), Object@2), Object@1)

The stub generation heuristic selects a pop() operation, which is non-deterministic, instead of a push(obj_j), which would yield two undistinguishable resulting terms. Instead, after one application of pop() as a stub, the two objects are distinguishable, for example by applying a further pop() as an (impure) observer.

As a last, critical optimization, we use the behavioral model to stop stub generation if we reach concrete deterministic states. If we are testing two terms for inequality and we apply a stub such that both terms reach a deterministic state, then every other stub term generated by applying other operation from this stub cannot introduce any further information about that inequality. The reason is simple: if after a particular stub we reach a deterministic state, then the two terms are undistinguishable. Consider the ObjectStack data abstraction and suppose that we add an additional clear() operation, which eliminates every object from the stack. In this case, the behavioral model in Fig. 5 must be modified by adding transitions labeled clear() which would connect each state with state $(s_0, null)$. For every pair of terms, if we apply the clear() method they surely become undis-

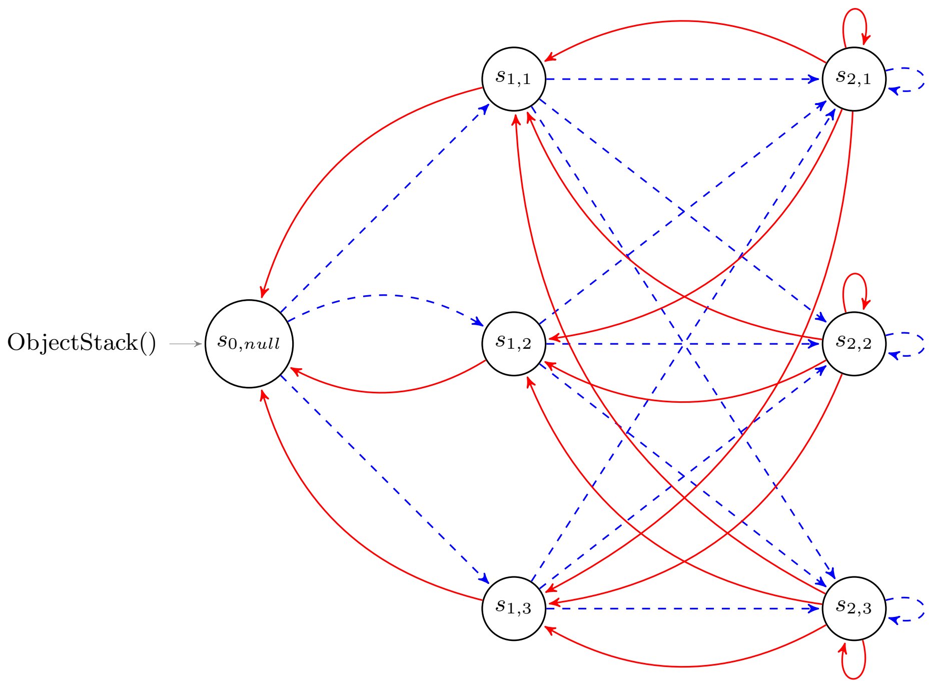

Figure 5: Unfolded model of ObjectStack with instance pool abstraction (Solid arcs represent pop() transitions; dashed arcs represent push(. . .)).

tiguishable, and it is useless to apply any other subsequent operation, because its result will always generate the same equivalent object. The same situation applies to states $s_{1,k}$ in Fig. 5. In fact, all operations labelling transitions exiting these states are deterministic (note that the three dashed transitions are labeled push(obj$_j$) with different values of j).

5. PRELIMINARY ASSESSMENT

Our optimization of `Heureka` mainly regards the generation of state equations and axioms between states. Our approach provides a smaller number of comparisons between terms during equation generation and a better performance of the equality engine. To provide a quantitative assessment of our optimization, we propose three performance metrics:

1. the total number of comparisons (i.e. the number of calls to the equality engine) needed by the equation generation;
2. the total number of method invocations on the target class; in the case of our approach, this includes the overhead invocations needed for the generation of the behavioral model;
3. the average number of method invocations on the target class for each invocation of the equality engine.

Table 1 shows the test results for the ObjectStack class and a SymbolTable class implementing a symbol table abstract data type. The former considers the state equation generator with minimal state optimization, while the latter uses the state equation generator without the minimal state term optimization (used for axioms completeness).

In the case of the ObjectStack class, `Adiheu` significantly reduces the average number of method invocations used by the equality engine. This reduction shows the benefits of using a behavioral model to drive both the proof of inequality, which is discovered sooner, and the assumption of equality, where many test cases for indistinguishable terms are ignored. The reduction varies from 45% in the worst case to 72% in the best case. A reduction is also noticeable on the total number of method invocations, despite the additional invocations due to behavior model generation. In this case, the degree of reduction varies from 65% to 83%. Analogous reduction of an order of magnitude were obtained by applying `Adiheu` to the SymbolTable class, even if this experiment did not exploit the minimal state term optimization.

6. CONCLUSIONS

Recovering specifications from run-time observation of the bevavior of existing code is a practically relevant and theoretically challenging problem. This paper focused on recovering algebraic specifications from Java classes implementing data abstractions. Our approach is based on the `Heureka` method defined by [9], which is significantly modified and optimized by applying techniques inspired by [3]. Our method has been implemented by a prototype, called `Adiheu`. A preliminary assessment of `Adiheu` shows that it significantly improves `Heureka`.

We view this work as an initial step along a path that tries to identify sound methods and useful heuristics that can help

Class ObjectStack											
	Heureka					Adiheu					
6	17	4	136	3567.0	1891523	13	17	4	13	1158.84	601599
7	45	6	990	11891.06	2.4×10^7	13	45	6	123	3283.5	4.5×10^6
8	54	15	1431	9209.47	7.5×10^7	13	54	15	321	4997.42	2.6×10^7
9	138	21	9453	33137.54	1.1×10^9	13	138	21	1218	8470.50	1.7×10^8
Class SymbolTable											
5	51	192	1275	4711.8	4.6×10^7	59	51	192	194	213.39	3.8×10^6
6	131	1119	8515	9273.84	1.2×10^9	59	131	1119	1147	264.5	3.6×10^7
maxSizeOfTerms	# of Terms	# of Equations	# of Comparisons	Avg. Invocations on Equality Engine	Total Method Invocations	# of States	Total # of Terms	Total # of Equations	Total # of Comparisons	Avg. Invocations of Equality Engine	Total Method Invocations

Table 1: Tests on ObjectStack and SymbolTable classes.

recover specifications by observing the run-time behavior of software. This problem has an important practical relevance, as it could help establish trust on third-party software used as a service.

7. REFERENCES

[1] Briand, L. C., Labiche, Y., and Miao, Y. Towards the reverse engineering of uml sequence diagrams. In *IEEE Working Conference on Reverse Engineering (WCRE'03)* (2003), pp. 57–66.

[2] CoFI (The Common Framework Initiative). *Casl Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[3] Dallmeier, V., Lindig, C., Wasylkowski, A., and Zeller, A. Mining object behavior with Adabu. In *WODA 2006: ICSE Workshop on Dynamic Analysis* (May 2006).

[4] Diaconescu, R., Futatsugi, K., and Iida, S. Component-based algebraic specification and verification in CafeOBJ. In *World Congress on Formal Methods* (1999), pp. 1644–1663.

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

[6] Goguen, J. A., Thatcher, J. W., and Wagner, E. W. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology, Volume 4:Software Specification and Design*, R. T. Yeh, Ed. Prentice Hall, 1978, ch. 5, pp. 80–149.

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

[8] Guttag, J. V., and Horning, J. J. The algebraic specification of abstract data types. *Acta Informatica 10*, 1 (1978).

[9] Henkel, J., and Diwan, A. Discovering algebraic specifications from Java classes. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference* (Darmstadt, July 2003), L. Cardelli, Ed., Springer.

[10] Leavens, G. T., Baker, A. L., and Ruby, C. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, H. Kilov, B. Rumpe, and I. Simmonds, Eds. Kluwer Academic Publishers, 1999, pp. 175–188.

[11] Meyer, B. Design by Contract: The Eiffel Method. In *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems* (1998), p. 446.

[12] O'Callahan, R., and Jackson, D. Lackwit: a program understanding tool based on type inference. In *ICSE '97: Proceedings of the 19th international conference on Software engineering* (New York, NY, USA, 1997), ACM Press, pp. 338–348.

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

[14] Sagiv, M., Reps, T., and Wilhelm, R. Parametric shape analysis via 3–valued logic. In *Symposium on Principles of Programming Languages* (1999), pp. 105–118.