Mining Object Behavior with ADABU

Valentin Dallmeier · Christian Lindig · Andrzej Wasylkowski · Andreas Zeller
Dept. of Computer Science, Saarland University, Saarbrücken, Germany
{dallmeier, lindig, wasylkowski, zeller}@cs.uni-sb.de

ABSTRACT

To learn what constitutes *correct* program behavior, one can start with *normal* behavior. We observe actual program executions to construct *state machines* that summarize object behavior. These state machines, called *object behavior models*, capture the relationships between two kinds of methods: *mutators* that change the state (such as `add()`) and *inspectors* that keep the state unchanged (such as `isEmpty()`): "A `Vector` object initially is in `isEmpty()` state; after `add()`, it goes into `¬isEmpty()` state". Our ADABU prototype for JAVA has successfully mined models of undocumented behavior from the AspectJ compiler and the Columba email client; the models tend to be small and easily understandable.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Class invariants; D.2.1 [**Requirements/Specifications**]: Tools; D.2.5 [**Testing and Debugging**]: Tracing

General Terms

Experimentation, Documentation, Languages, Verification

Keywords

Program Analysis, Object Behavior Model, Mining, Automata, Observer Method, Inspector Method, Purity Analysis, Java, Program Instrumentation, Object State

1. INTRODUCTION

When some object uses the services of another object, it must satisfy a number of constraints—for instance, it can only invoke public methods, and it must provide appropriate arguments. Some constraints, though, are of a *temporal* nature: for instance, the method `Vector.add()` must be called before `Vector.remove()`.

Temporal constraints are often undocumented—but implicit in a correct interaction between a client and an object. This observation led several researchers to *mine* temporal constraints dynamically (Ammons et al., 2002; Yang and Evans, 2004; Weimer and Necula, 2005; Xie et al., 2006); applications include program understanding and error detection (Weimer and Necula, 2005; Whaley et al., 2002). Mining learns a finite-state automaton whose states represent the state of an object and whose transitions are labeled with method names. Such automatons reflect that executing a method like `Vector.clear()` leads a vector object to a new state where calling `Vector.remove()` is illegal. More generally, the automaton approximates all legal method-call sequences and serves as a temporal specification.

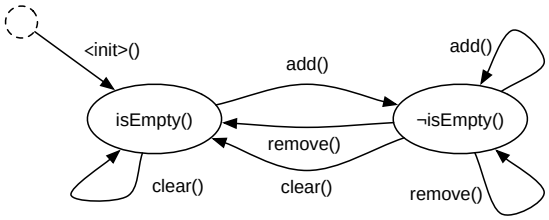

Figure 1: An object behavior model for the JAVA `Vector` class.

While *transitions* between states are easy to obtain, characterizing meaningful *states* has always been an issue: When do two sequences of method calls end in the same state? In this paper, we use a both novel and simple approach to characterize states. Rather than using anonymous states (Cook and Wolf, 1998; Ammons et al., 2002; Weimer and Necula, 2005), or referring to implementation details like variables or branch conditions (Xie et al., 2006; Whaley et al., 2002), we characterize object states by their *externally observable state.* This state is obtained by invoking automatically identified *inspector methods*: executing `Vector.clear()` leads to a state where `Vector.isEmpty()` is false. Our object behavior models thus do not rely on implementation aspects and therefore are aligned with a client's view.

Figure 1 shows an example of an object behavior model, describing the behavior of a JAVA `Vector` object mined from the execution of Columba, a modern email client whose implementation comprises more than 500 classes (Stich and Dietz, 2005). After construction (`<init>`), a `Vector` object is empty, as indicated by the inspector method `isEmpty()` returning true. After adding elements using the `add` method, the state changes to `¬isEmpty()`.

By removing elements, the state can become empty again. As shown in the example, relying on inspectors to characterize state allows the model to rely on *abstractions* as defined in the interface—the distinction between empty and non-empty vectors was important enough to warrant the existence of a dedicated inspector, and therefore is also reflected in the model.

WODA'06, May 23, 2006, Shanghai, China.

A model expresses the relationship between mutator and inspector methods: calling a mutator changes the state of the object as it is observable through its inspector methods. For instance, a call to the `clear()` method always ends up in the `isEmpty()` state. A model also expresses which sequences of mutator calls are possible: an invocation of `remove()` in `isEmpty()` state has not been observed. Therefore, it is uncommon and in fact, incorrect, to invoke `remove()` right after initialization or a call to `clear()`. Like other dynamic analyses, we build on the observation that *common behavior is often correct behavior;* our models thus are likely to represent *universal invariants*—which gives them a great potential for documenting and validating program properties.

Mining object behavior models from JAVA programs is fully automatic and does not rely on special test cases. Indeed, we can mine models from an arbitrary program execution for all its classes simultaneously. Our approach takes the following steps, which are detailed in Section 2:

1. For each class, we statically identify its *inspectors*—public methods that do not change state (such as `isEmpty()`). Any public non-inspector is a *mutator* (such as `add()`).
2. While executing the program, we *invoke all inspectors before and after each mutator* to retrieve the object's state—a vector $\langle x_1, \ldots, x_n \rangle$ of concrete values. For the `Vector` class in Figure 1, this is $\langle$`size()`, `isEmpty()`, `capacity()`$\rangle$.
3. We *abstract* from concrete values by mapping them to small finite domains such as positive/negative/zero for integers. We thus obtain a vector of abstracted values like $\langle$`size()` $> 0, \neg$`isEmpty()`, `capacity()` $> 0\rangle$ for the state of a `Vector` object.
4. Each abstract vector becomes a state which is reached by the mutator. In the `Vector` class, there are two such states: an empty state

   $$\texttt{isEmpty()} \wedge \texttt{size()} = 0 \wedge \texttt{capacity()} > 0$$

   and a nonempty state

   $$\neg\texttt{isEmpty()} \wedge \texttt{size()} > 0 \wedge \texttt{capacity()} > 0 \ .$$

5. The result is a *finite state model*, such as the one shown in Figure 1. The transitions occur between (abstract) object states, summarizing normal object usage.

We have built a prototype called ADABU[1] which realizes the above approach. During a program run (or a set of program runs), ADABU first obtains a model for each individual object; then, it merges these models for all objects of a single class. We thus end up with one model per class, summarizing the normal object usage across all observed runs.

The remainder of this paper details our notion of object behavior models and how we obtain them (Section 2), discusses models that we found (Section 3), surveys related work (Section 4), and concludes with an outlook (Section 5).

2. EXTRACTING MODELS

The process of extracting models from a run consists of two steps: First, a static analysis identifies all side-effect free methods in the program. A subset (defined below) of these methods constitutes the set of inspectors for the program. During the second phase, the program is executed and inspectors are called to extract information about an object's state.

[1]ADABU Detects All Bad Usages. "Adabu" is also the Swahili word for "good behavior".

2.1 Inspectors And Mutators

For the purpose of extracting models, we partition the methods of a class into *inspectors* and *mutators*. An inspector returns information about the state of an object. A method that is to be used as inspector must meet the following criteria:

- **Not Void.** We expect inspectors to pass information to the caller in the returned value, and thus require a return type other than `void`.
- **No Parameters.** An inspector must not take parameters. The reason for this is that if an inspector takes an argument a, the return value may depend not only on the state of the object itself, but also on the state of a. This requirement also makes it much easier to call inspectors at runtime.
- **No Side Effects.** The execution of an inspector must not have side-effects on the state of the program. This is necessary because our tool invokes inspectors to extract the state of objects. If we would allow inspectors to have side-effects, state extraction might have an impact on the program run.

The first two criteria can be checked with a straightforward static analysis, whereas identification of side-effect free methods (also referred to as purity analysis) requires a static whole-program analysis as described by Rountev (2004).

ADABU currently uses the purity analysis by Salcianu and Rinard (2005). This conservative analysis classifies a method as pure only if it is certain not to modify objects that existed prior to the invocation of that method. This definition is precise enough for our purposes and, at the same time, allows an inspector the creation of temporary objects, which may be also returned to the caller. Thus, when we call an inspector to extract an object's state, we can be sure that this has no effects on the program state.

Methods that are not free of side-effects are called *mutator methods*. Invoking a mutator method on an object may change the object's state. Thus, an invocation of a mutator method represents a transition in an object's model, whereas inspector methods are called to extract the state of an object.

2.2 Instrumentation

After static analysis, each method of the program is classified either as an inspector or as a mutator. This information is used by the instrumentation to enframe the body of every mutator with code to extract and store the state of the object.

For instrumentation, ADABU uses the JAVASSIST framework by Chiba and Nishizawa (2003). Instrumentation occurs before execution by rewriting the JAR files of the investigated program. Figure 2 demonstrates instrumentation of the `Vector` class. (We actually instrument the bytecode of a program; the figure shows the equivalent source code instrumentation.)

As a first step, a method `extractState()` is generated and added to the class. As implied by its name, this method extracts the state of an object: it invokes every inspector and stores the result together with the name of the inspector. For demonstration we use only three out of nine inspector methods for `Vector`, namely `isEmpty()`, `size()` and `capacity()`. The results of these three inspectors are encapsulated in a `State` object and returned by the method.

The `extractState()` method is invoked by the code injected into every mutator method, such as `add()`. Method `add()` is a mutator because it stores its argument in a field of `Vector` , thus changing state. Prior to the execution of the original method body, the state of the vector is extracted by calling `extractState()`

```
public class Vector {
  ...
  public State extractState() {
    State s = new State();
    s.add("isEmpty", isEmpty());
    s.add("size", size());
    s.add("capacity", capacity());
  }
  ...
  public void add(Object o) {
    State pre = extractState();
    try {
      ⟨body of add⟩
    } finally {
      State post = extractState();
      model.addTransition(pre, post, "add");
    }
  }
  ...
}
```

Figure 2: Instrumentation for `Vector.add()`. The instrumentation happens at the byte-code level but for clarity an equivalent source code instrumentation is shown.

and the result is stored in a local variable. After the execution of the method body, the state is extracted again and a transition is added to the model for this object. The body of `add()` is surrounded by a `try-finally` block to capture the state at both regular and exceptional method exits.

The overhead of instrumentation is negligible: instrumenting version 1.1b4 of AspectJ (2382 classes) takes only 177 seconds and increases the code size from 5.5 to 12 megabytes. We observed similar values for other applications.

2.3 Model Construction

After instrumentation has finished, we can execute the instrumented program and learn behavior models.

Concrete states. For an object, we define its *state* as a vector $v = (x_1, \ldots, x_n)$, where each x_i is the return value of an inspector.

For simplicity, let us assume a JAVA `Vector` object has just three inspectors $x_1 =$ `isEmpty()`, $x_2 =$ `capacity()`, and $x_3 =$ `size()`. A new `Vector` of capacity 20 might thus have a state $v =$ (`true`, 20, 0), reflecting the inspector values.

Traces. A *trace* for an object becomes a sequence of triples $t = \big[(v_1, m_1, v_1'), (v_2, m_2, v_2'), \ldots\big]$, where each v_i and v_i' is the state before and after invocation of a mutator m_i.

Here is an example trace of a `Vector` object, including its initialization:

$$t = \left[\begin{array}{lll} \big((\bot, \bot, \bot) & \langle\texttt{init}\rangle\texttt{()}, & (\texttt{true}, 20, 0)\big), \\ \big((\texttt{true}, 20, 0), & \texttt{add()}, & (\texttt{false}, 20, 1)\big), \\ \big((\texttt{false}, 20, 1), & \texttt{add()}, & (\texttt{false}, 20, 2)\big), \\ \big((\texttt{false}, 20, 2), & \texttt{remove()}, & (\texttt{false}, 20, 1)\big), \\ \big((\texttt{false}, 20, 1), & \texttt{remove()}, & (\texttt{true}, 20, 0)\big), \\ \big((\texttt{true}, 20, 0), & \texttt{add()}, & (\texttt{false}, 20, 1)\big), \\ \big((\texttt{false}, 20, 1), & \texttt{clear()}, & (\texttt{true}, 20, 0)\big), \\ \big((\texttt{false}, 20, 0), & \texttt{clear()}, & (\texttt{false}, 20, 0)\big) \end{array}\right]$$

Abstract states. If we used the plain return values for inspectors, the model would have a very large number of states. As an example, consider the inspector `size()` of the `Vector` class. If the concrete value of `size()` was used to characterize the state of a vector, the resulting model would have at least as many states as the maximum size of the vector. Therefore, we use *abstractions over the return values of inspectors* rather than the concrete values themselves.

Formally, we use a *state abstraction function* named *abs* which maps concrete values v to abstract states s as follows:

- Concrete numerical values x_i (of type `int`, `double`, etc.), are mapped to three abstract states $x_i < 0$, $x_i = 0$, and $x_i > 0$.

  In the `Vector` example, for instance, the values returned by the `size()` inspector are mapped to two abstract states "`size()` $= 0$" and "`size()` > 0".

- Object references x_i are mapped either to the abstract state $x_i =$ `null`, or to the abstract state x_i `instanceOf` c for class c of the object referenced by x_i.

  If the `Vector` from Figure 1 contained `File` objects, the values returned by the `firstElement()` inspector would be mapped to two abstract states "`firstElement()` = `null`" and "`firstElement()` `instanceOf` `File`".

- Enumerations and boolean values x_i are mapped to one singleton abstract state for each single value.

  In Figure 1, this is how the `isEmpty()` method induces two abstract states.

For the `Vector` trace, above, we would thus obtain three abstract states s_0, s_1, s_2:

	`isEmpty()`	`capacity()`	`size()`
s_0	$\bot$	$\bot$	$\bot$
s_1	`true`	> 0	$= 0$
s_2	`false`	> 0	> 0

—that is, the three states of the model in Figure 1.

Models. The abstract states, as determined in the previous step, form the states s of object behavior models. A transition $e = (s, m, s')$ occurs between two states s, s' and is labeled with a mutator m. A transition e is part of the model if and only if the trace t contains a transition between two concrete states v and v' abstracted by s and s', respectively (formally, $\exists (v, m, v') \in t \cdot$ *abs*$(v) = s \wedge$ *abs*$(v') = s'$ must hold).

For the `Vector` trace, we would obtain seven abstract transitions between the states s_0, s_1, s_2, as described above:

$$T = \left\{\begin{array}{lll} (s_0, & \langle\texttt{init}\rangle\texttt{()}, & s_1), \\ (s_1, & \texttt{add()}, & s_2), \\ (s_2, & \texttt{add()}, & s_2), \\ (s_2, & \texttt{remove()}, & s_2), \\ (s_2, & \texttt{remove()}, & s_1). \\ (s_2, & \texttt{clear()}, & s_1), \\ (s_1, & \texttt{clear()}, & s_1). \end{array}\right\}$$

—that is, the transitions of the model in Figure 1.

Summarizing models. As a last step, we merge all object models for all objects of a class into a single model that summarizes the behavior of all instances of this class. Merging automata is easy, because each state of the automaton is uniquely identified by the object state it represents. The resulting model consists of the union of all states and transitions of all observed object behavior models.

	AspectJ	Columba
Version	1.1b4	1.0
Classes	2382	1513
Models	589	538
Mutators (avrg, per model)	7.7	4.1
Inspectors (avrg, per model)	8.0	3.2
States (avrg, per model)	11.0	4.1
Transitions (avrg, per model)	17.7	5.2

Table 1: Statistics for our subjects and their models.

3. EXPERIENCES

We have implemented our approach for JAVA programs and used it to extract models for some classes of the JAVA API, the AspectJ compiler (Kiczales et al., 2001) and the Columba email client (Stich and Dietz, 2005). Table 1 provides some statistics for our subjects.

3.1 Overhead

In order to instrument a program for model extraction, we first need to know which methods we may use as inspectors. Currently, we use the purity analysis by Salcianu and Rinard (2005), which is the only scalable implementation we are aware of. It is based on the FLEX compiler infrastructure (Rinard, 2002), which unfortunately restricts analysis to programs compiled against the GNU Classpath API 0.08. Besides this limitation, the analysis is sufficiently fast and produces reliable results. Analyzing version 1.1b4 of the AspectJ compiler, for example, takes about 22 minutes. Since the identification of inspectors needs to be done only once and prior to mining, this does not pose a problem.

We have successfully extracted models from about 100 test runs of AspectJ. These test runs were obtained from test suite ajc1.1 included in the source distribution of AspectJ. From each of these tests, we have learned a set of object behavior models, and merged models for the same class from different runs into one model per class.

Using ADABU, a run of the instrumented version of AspectJ takes about 5 times longer than the original version. While we believe that there is still room for optimizations, a large part of the runtime overhead is unavoidable as we have to extract state twice for every method invocation in the original program run.

Our second test subject, Columba, is an email client with a graphical frontend. We ran an instrumented version of Columba, browsed through several folders of an IMAP account, created and deleted folders, and sent an email using an SMTP server. As it is difficult to measure the precise execution time for GUI programs, we can only report that the instrumented version was sufficiently fast to remain usable.

3.2 Models

Altogether, we have mined models for 589 classes of AspectJ and 538 classes of Columba. Unfortunately, the purity analysis failed to analyze parts of Columba. To address this problem, we introduced artificial inspectors that simply return object attributes.

The bottom of Table 1 summarizes the over 1100 models we have mined from our test subjects. The average model for AspectJ has 11.0 states and 17.7 transitions. The largest model, recorded for the `AjParser` class, has 1500 states and 4975 transitions (each state consists of the values for more than 50 integer inspectors). As a first example, we already discussed the `Vector` class in Section 1. In the remainder of this section we present four more (small) models in detail. The corresponding diagrams in Figure 3 to Figure 6 were derived automatically but edited for presentation.

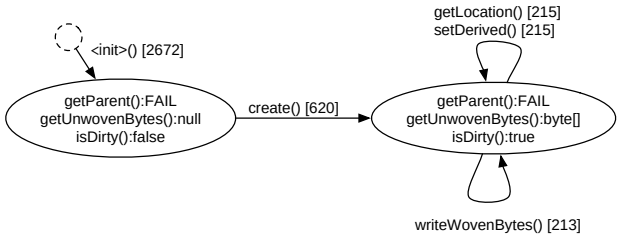

Figure 3: An object behavior model for the `DeferredWriteFile` class. A number in square brackets denotes a transition's frequency.

3.3 Deferred Writes

As an example for a simple model, Figure 3 shows the model for class `DeferredWriteFile` from AspectJ. Instances of this class are used to map the contents of a file into memory, and defer any changes like deletion and modification until the application decides to write the current state to the file system. The `isDirty()` method indicates that the cached content has changed and therefore the file needs to be written.

The object behavior model indicates that the usage of this class in AspectJ is quite limited. After the call to the constructor, a call to `create` causes the file to be loaded from the file system and the dirty flag to be set. After the compiler has finished, the woven class is written to the file system. Thus, AspectJ only uses instances of `DeferredWriteFile` for usual read and write operations, which contradicts the intention of the class.

3.4 Mutex

Figure 4 shows an object model for the `Mutex` class implemented in Columba. A mutex gives a thread exclusive access to a resource. It is used in Columba to ensure the integrity of data transmitted via IMAP and SMTP protocols. Internally, the Mutex class uses a flag named `mutex` which is true whenever the resource is locked and false otherwise. Before a thread may access the shared resource, it must call the `lock()` method. If another thread is currently using the resource, `lock()` waits until the resource is free and locks it for the thread. When a thread has finished accessing the resource, it must call `release()` to release the lock again.

The behavior model has three states. Right after instantiation, the object is in an undefined state. The first method invoked on the new instance is the constructor (`<init>`). After construction, the mutex flag is initially set to false. This is reflected in the model by a transition from starting state to state `mutex:false` labeled `<init> [107]`. The number in square brackets denotes how often a method caused a transition.

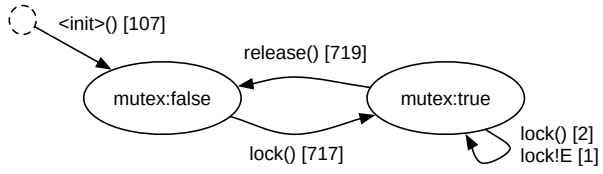

Figure 4: An object behavior model for the `Mutex` class.

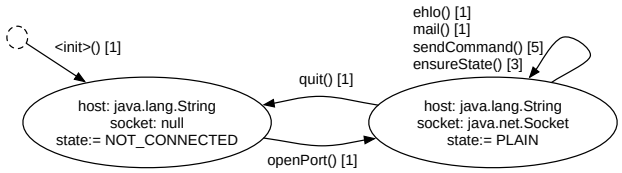

Figure 5: An object behavior model for `SMTPProtocol`.

The model reveals that synchronization is almost never needed since the resource is unlocked. We know this because `lock` was called 717 times when mutex was false, while we observed only two invocations of `lock` when mutex was true. We also observe that the number of `lock()` and `release()` invocations is equal. This strongly suggests that the mutex is used correctly throughout the program. (We cannot say this for sure as the model was learned from multiple `Mutex` instances.)

There is another mysterious transition: The label `lock!E [1]` indicates that one call to `lock` raised an exception. An investigation of the source code revealed that `lock` throws a runtime exception when the thread waiting for the lock is interrupted. This is obviously a flaw in the method design and should be solved by throwing a meaningful exception or a return code indicating that acquiring the lock failed.

3.5 SMTPProtocol

Let us now examine the model for the `SMTPProtocol` class. This class implements communication with a mail server according to the SMTP protocol specification. For presentation, the model in Figure 5 was slightly simplified and shows only the three most important inspectors `host`, `socket`, and `state`.

After a call to the constructor, the server host is set but no socket is opened to connect to it. This is indicated by the socket being `null` and the attribute `state` being set to a constant value `NOT_CONNECTED`. Prior to communicating with the server, the client must call `openPort()`, which causes the socket to be created and the state variable to be set to `PLAIN`. During communication (e.g., `ehlo`, `mail`), the automaton remains in this state until calling `quit` causes a transition to state `NOT_CONNECTED`.

A client that uses this class must adhere to this sequence of calls. As SMTP is not the only protocol used by Columba, there are other classes implementing protocols which have similar requirements, but the documentation does not mention them. Assuming that the models capture correct usage, they do not only reveal these requirements but also explain them: It is only through an invocation of `openPort()` that the state changes from `NOT_CONNECTED` to `PLAIN`.

3.6 IMAPProtocol

Columba also supports accessing emails through the IMAP protocol, implemented in the `IMAPProtocol` class. Figure 6 shows the behavior model for this class. After the call to the constructor, the protocol is in state `NOT_CONNECTED` and the `openPort()` method has to be called to open a connection to the server, just like in the SMTP model.

In order to access data on the server, the client must call the `login()` method, causing a transition to `AUTHENTICATED` state. In this state, the server may be queried for its status and capabilities, but it is not yet possible to access mails on the server. This requires the selection of a mailbox using the `select()` method, causing a transition to state `SELECTED`. Now the protocol can be used to look for mails, change the folder structure on the server or log out again.

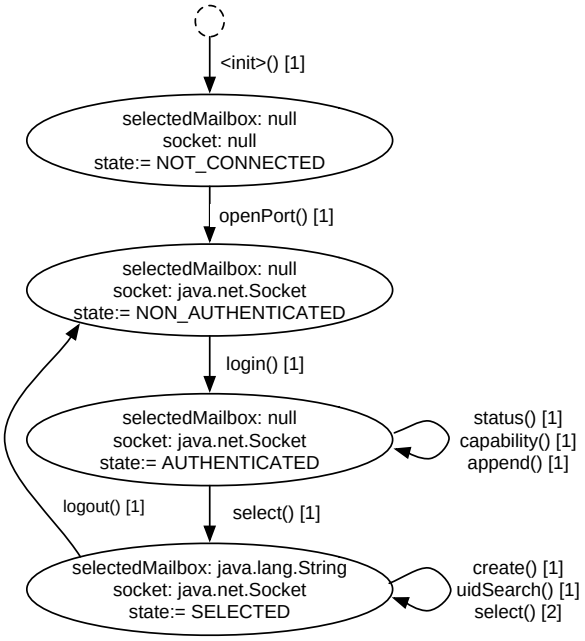

Figure 6: An object behavior model for `IMAPProtocol`.

Obviously, the sequence `openPort()`, `login()`, `select()` must be executed on every instance of `IMAPProtocol` before any further email access. Again, this is not documented anywhere in the source code of Columba. The behavior model not only reveals this requirement, but also helps to understand why it is needed.

All in all, these four examples highlight the expressiveness of object behavior models: they associate method invocations with changes to the state of an object. This allows for a better understanding of objects than previous models that only considered the sequence of method invocations.

4. RELATED WORK

Concrete program executions as a source of abstractions have inspired many researchers and gained enormous interests in the past years. Most approaches express the behavior of programs or parts thereof as finite state machines. Object behavior models are unique in using automatically identified inspector methods to characterize object state independent from its implementation.

Learning from Sequences. Cook and Wolf (1998) is the seminal work about inference of finite-state machines from event sequences. The paper compares neural networks, grammar inference, and a Markov method. While not described in the paper, this could be applied to method-call sequences to obtain models for objects. However, states are unlabeled and this purely syntactic approach tends to produce models that are very detailed.

Mining Specifications. The concept of learning models from actual program runs was first explored by Ammons et al. (2002), applying a probabilistic NFA learner on C traces. Their approach relies on manual annotations to relate functions to objects (such as C sockets or X11 selections) and to distinguish object definers from object users.

To mine a specification like the one in Figure 1 using the work of Ammons et al., one would have to distinguish mutators and inspectors manually. In the set of C functions, one would also have to identify the one parameter which denotes the actual object being accessed. Finally, the states would not be associated with inspectors like `isEmpty()`, but remain anonymous—as "the state which is reached after calling `add()`".

Dynamic Invariants. Dynamic invariants, as conceived by Ernst et al. (2001), express properties of data that hold at specific moments during the observed executions. Applied to the `Vector` class above, Ernst's DAIKON tool could detect a dynamic invariant `this._size > 0` at the end of `add()`, thus modeling the postcondition of `add()`.

Dynamic invariants, as mined by DAIKON, could be used to characterize states in an automaton, and we may end up in a model like the one shown in Figure 1; the invariants would also characterize whether a method serves as mutator or inspector. The difference is that the states would refer to internal attributes such as `this._size`. DAIKON may also use inspector methods provided by a user, but it does not identify them automatically. In contrast to DAIKON, we do not check a set of pre-defined abstractions on internal data, but rather rely on abstractions as given by public inspectors. In this way, we do not only reflect the user's view, but can also express more high-level properties.

Mining Temporal Relations. Unlike most approaches, Yang and Evans (2004) don't build a model that captures all behavior of a system. Instead, they *check* for the presence of certain temporal patterns in a squence of events. This leads to very abstract characterizations and avoids the problem of labeling states. At the same time, the approach is limited by the a-priori knowledge of patterns it looks for.

Mining Pairs. Weimer and Necula (2005) also avoid learning complete automata and instead learn pairs of matching function calls (like `open` and `close`) from method-call traces. Using this knowledge they find errors in programs where the pairing is violated in error handlers. Like the previous one, this work benefits from its restriction to partial temporal specifications; in addition, it is a great showcase for an application of mined specifications.

Permissive Interfaces. Henzinger et al. (2005) learn finite state automata that describe legal method call sequences for an API. Their approach is based on repeatedly generating candidate graphs and checking them against an abstract program representation. Each state in the candidate graph corresponds to an internal state of an object.

In contrast to mining object behavior models, their approach works purely static on a JAVA subset. Henzinger et al. (2005) only present interfaces learned from 4 classes of the JAVA API. They also need to manually specify the set of predicates needed to track the state of an object. In contrast, our approach relies on purity analysis to identify inspectors and mutators.

Mining Object Interfaces. Whaley et al. (2002) suggest to learn automata that capture observed sequences of method calls for an object. These automata are then sliced by the fields each method accesses during its execution, thus creating *submodels* describing call sequences of methods that affect the same attribute of a class.

While we use side-effect free methods to extract the state of an object, Whaley et al. (2002) ignore them in order to avoid pollution of the automata. Unlike object behavior models, their automata do not provide information about the state of the object and thus cannot relate state transitions to method calls.

Object State Machines. The work closest to ours is by Xie and Notkin (2004): they instrument JAVA programs to extract the state of an object, which is later abstracted by calling all methods on the object and inspecting the returned values. Unlike us, they don't restrict calls to pure methods. Therefore they have to cope with side effects. To remedy this, mining relies on automatically generated test cases where the object is discarded after its state was extracted; these test cases also provide arguments for the methods being called. In contrast, we can instrument and mine models from arbitrary programs since pure observers can't affect program execution. We are thus more likely to capture common behavior of objects.

Unlike us, Xie and Notkin (2004) do not abstract from concrete values to avoid state space explosion. Because of this, and the fact that transition labels include method parameters as well as their return values, their models are more accurate than ours—but less general and concise. Also, they do not include state-preserving transitions in their machines, as opposed to our approach, where every method call that happened during the execution is present in the model.

State Abstraction. Finding useful abstractions over state is a challenge in itself. Our approach has been inspired by the work of Liblit et al. (2005), who characterized runs according to, among others, functions returning positive, negative, or zero values.

Applied to the `Vector` class, Liblit's approach could determine that program failures correlate with `isEmpty()` being true. Establishing such correlations is orthogonal to our approach.

Method Call Sequences. In own earlier work (Dallmeier et al., 2005), we showed that *sequences* of method calls could be used to characterize sets of executions, thus helping in defect localization. In contrast to this earlier work, we now mine full-fledged finite state automata rather than fixed-length sequences—automata which have many more potential uses.

Applied to the JAVA `Vector` class, our earlier approach could determine that program failures correlate with a sequence of three method calls ⟨`clear()`, `isEmpty()`, `remove()`⟩. Establishing such correlations is possible for models as well, and orthogonal to the work presented in this paper.

5. CONCLUSIONS AND FUTURE WORK

Object behavior models capture essential properties of an object from the view of the object's client. They can be used for documenting common behavior, and therefore can serve as a base for further specifications, with all the resulting applications.

The key ingredient and central contribution of our approach is the distinction between *mutator* and *inspector* methods. As prior work, we use mutators to label transitions; however, we use inspectors to capture the observable state of objects, which is new.

Mining object behavior models works for real applications like the Columba email client, is scalable, and does not rely on special test cases: We mined models for the 538 classes in Columba at once, all while the application was still usable.

In addition to general issues such as performance or ease of use, our future work will concentrate on the following topics:

Dynamic Checking. In addition to learning models at runtime, ADABU could also check behavior models dynamically. A set of previously learned automata can be used to flag or even prevent uncommon behavior, e.g. in security-related API s. This may be especially interesting for security checkers, such as malware scanners.

Alternative Abstractions. The size of our models depends on our abstraction function *abs* which we apply to values returned by inspectors. Currently, we use a coarse abstraction and consequently our models tend to be concise. We are currently experimenting with different abstraction styles, resulting in models where the level of detail can be scaled arbitrarily.

Deep Models. Some inspectors return objects. Instead of just using the class name as a value, we could inspect the returned object recursively, up to a certain depth. This would allow to express an object's state as the state of its constituents, and lead to models where states are characterized by tree-structured values. By adjusting the depth of the state extraction, we could control the granularity of our models.

Purity Analysis. Right now, the identification of inspector methods critically relies on purity analysis, a static whole-program analysis. With currently only one usable and scalable implementation available, we deem that purity analysis is still in its infancy. We are currently evaluating pragmatic alternatives for the cases where purity analysis is not usable.

Static Analysis. We would like to check programs statically against mined models and thus find deviations from those. Under the assumption that models capture correct behavior such deviations may point to incorrect usage of an API.

All in all, we find that program executions offer a wealth of data that can be mined for recurring patterns and rules. With object behavior models, we give a concise characterization of common behavior—a characterization with which we eventually hope to reach a sweet spot between simplicity and expressiveness. Finding this sweet spot is certainly a challenge for us as well as for other researchers; however, the multitude of potential applications will help us and the community identify the most useful approaches.

For future and related work regarding object behavior models, see

`http://www.st.cs.uni-sb.de/ample/`

Acknowledgments. Silvia Breu and Tao Xie provided valuable comments on earlier revisions of this paper.

References

Glenn Ammons, Rastislav Bodík, and Jim Larus. Mining specifications. In *Conference Record of POPL'02: The 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.

Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *Proceedings of the 2nd International Conference of Generative Programming and Component Engineering*, pages 364–376, 2003.

J. Cook and A. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for Java. In Andrew Black, editor, *European Conference on Object-Oriented Programming (ECOOP)*, pages 528–550, 2005.

Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *Proceedings of the Symposium on the Foundations of Software Enginnering*, 2005.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jorgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–26, June 2005.

Martin Rinard. Flex. `http://www.flex-compiler.lcs.mit.edu/`, 2002. Compiler infrastructure.

Atanas Rountev. Precise identification of side-effect-free methods in Java. In Panos Linos, editor, *20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 82–91, 2004.

Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, number 3385 in LNCS, pages 199–215, January 2005.

Timo Stich and Frederik Dietz. Columba. `http://columbamail.org/`, 2005. Open-source email client, implemented in Java.

Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pages 461–476, Edinburgh, UK, April 4–8 2005.

John Whaley, Michael Martin, and Monica Lam. Automatic extraction of object-oriented component interfaces. In Phyllis G. Frankl, editor, *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA-02)*, volume 27(4) of *SOFTWARE ENGINEERING NOTES*, pages 221–231, New York, July 22–24 2002. ACM Press.

Tao Xie, Evan Martin, and Hai Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), Research Demonstrations*, May 2006.

Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM 2004)*, pages 290–305, November 2004.

Jinlin Yang and David Evans. Dynamically inferring temporal properties. In Cormac Flanagan and Andreas Zeller, editors, *Proceedings of the Workshop on Program Analysis For Software Tools and Engineering (PASTE'04)*, pages 23–28, June 2004.