Compliance testing for wrapper-based API migration

Thiago Tonelli Bartolomei
University of Waterloo
Canada

Krzysztof Czarnecki
University of Waterloo
Canada

Ralf Lämmel
University of Koblenz-Landau
Germany

ABSTRACT

Wrapping is an established technique for API migration: the use of a given API within the system under migration is replaced by the use of a wrapper-based re-implementation of said API while using a different (preferred) API underneath. Except for some special cases, wrapper development is a craft. In particular, the compliance of a wrapper with the original API is hard to assess and guidance of wrapper development is very limited. In this paper, we describe a method for wrapper development that is essentially inspired by notions of scenario-based differential testing, API contracts as well as selective capture and replay of program executions. The method supports compliance testing of the wrapper under development against the original API; it guides the developer in improving compliance incrementally; it also allows for precise capture of unresolved differences between original API and wrapper. The method is evaluated by a study of wrapper development with different wrappers in the domains of XML processing, byte-code engineering, and GUI programming.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion checkers*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing; Testing tools*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms

Design, Verification

Keywords

API migration, wrapping, compliance testing, API contracts

1. INTRODUCTION

API migration [1, 29, 26, 25, 27, 3, 2, 18, 33, 11, 30, 17] is concerned with the evolution of software systems with regard to their dependencies on particular APIs in all domains of programming, e.g., XML processing, database access, byte-code engineering, or GUI programming. For instance, a system could be subject to 'modernization' such that a deprecated API is to be replaced by a state-of-the-art API. Also, two systems could be subject to 'integration' such that diverging options of APIs for a specific domain (e.g., two different GUI APIs) need to be consolidated by using only one API in the integrated system.

Wrapping is an established technique for API migration: the use of a given API within the system under migration is replaced by the use of a wrapper-based re-implementation of said API while using a different (preferred) API underneath. Except for some special cases, wrapper development is a craft. In particular, the correctness of wrappers is hard to assess and guidance of wrapper development is very limited.

In this paper, we are concerned with API migration based on wrapping and specifically the challenging case in which original API and replacement API are not related in any obvious manner, as, for example, in the case of upgrade transformations [26, 25]. Instead, the two APIs may differ in significant ways as far as protocols, contracts, type hierarchies and other aspects are concerned. In this context, our previous work [3, 2] has focused on the analysis of API differences as they cause compliance issues and implementation challenges in API migration; we have also proposed design patterns for developing wrappers [2].

The focus of this paper is on the development process for API wrappers.

Contributions of the paper

- We introduce a notion of scenario-based compliance testing for API migration such that the execution of scenarios—under the scrutiny of assertions for API contracts—is used to establish compliance between original API and wrapper-based re-implementation.
- We describe a method for developing compliant wrappers, which involves scenario design, scenario execution, selection of API contracts including tuning, data collection, reporting of compliance violations, and incremental wrapper evolution.
- We report on the evaluation of the method for wrapper development by means of a study in the domains of XML processing, GUI programming, and byte-code engineering. Our results show that guidance can be effective in improving wrapper compliance.

Submitted to ISSTA '12 Minneapolis, Minnesota, USA

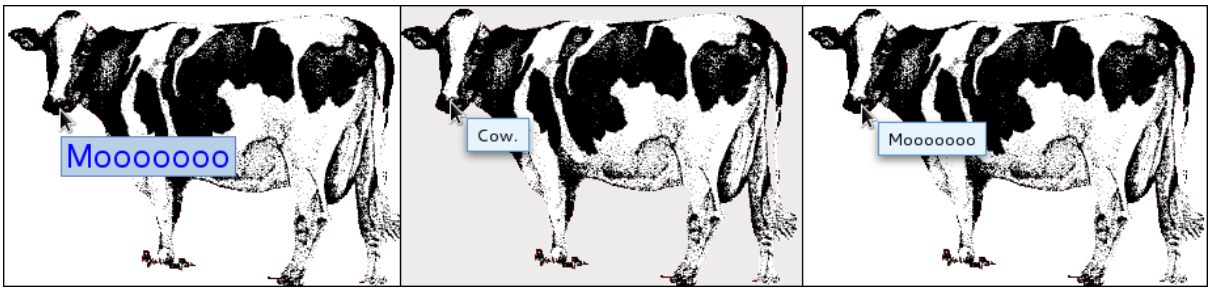

Figure 1: SwingSet2's Tooltip demo in Swing, SwingWT and evolved SwingWT versions.

The aforementioned method has been implemented in a toolkit called *Koloo*. The paper's website[1] provides access to accompanying material such as the wrappers and applications of the study as well as additional data.

Road-map of the paper. §2 presents a *motivating example* for wrapper development based on an open-source wrapper for Java's Swing API for GUI programming. §3 reports on *interviews* conducted with developers experienced in API migration, providing requirements for an API migration method. §4 develops the overall notion of compliance testing for wrapper-based API migration. §5 describes a *method for API migration* based on compliance testing. §6 describes validation. §7 analyzes related work. §8 concludes the paper.

2. MOTIVATING EXAMPLE

We will illustrate here *compliance issues* between original API and its wrapper-based re-implementation. The aim of wrapper development is to reduce such issues up to the point that the wrapper is 'good enough' for use in applications. Original API and replacement API (i.e., the API used internally by the wrapper) may enjoy 'arbitrary' differences [3, 2]; the wrapper must neutralize these differences.

Our example concerns two GUI APIs: i) the Swing API with the help of the AWT API, which are both part of JDK ('the Java platform'), and ii) the SWT API[2], which is part of the Eclipse project. There exists the open-source project SwingWT[3], which is a wrapper-based re-implementation of Swing in terms of SWT.

Using the method of this paper (see §5–§6) and designated tool support (i.e., the *Koloo* toolkit), we have developed a revision of SwingWT[4], to which we also refer as 'evolved SwingWT' in the sequel.

Fig. 1 illustrates compliance issues of SwingWT versus Swing; it also illustrates how our systematically evolved wrapper reduces the compliance issues. In the figure, we exercise one scenario of the *Tooltip* demo of SwingSet2[5], which is a set of Swing demos originally distributed by Sun. The scenario is concerned with displaying a specific tooltip, as it is triggered when the user positions the mouse over the cow's mouth. The scenario was chosen because the *Tooltip* demo is clearly concerned with triggering tooltips.

On the left, the reference behavior is that the cow moos. In the middle, the SwingWT wrapper, prior to our efforts, is at work. Two problems are noticeable in the image: the background color is gray instead of white, and the tooltip shows a different message: 'cow' as opposed to 'Mooooooo'. On the right, the evolved SwingWT wrapper is at work. The background color complies with the original API; the tooltip's messages complies with the original API as well. We have chosen the example to be visually striking, but we must emphasize that the method proposed in this paper specifically reveals issues that are not easily spotted by looking at particular screenshots; it also applies to domains other than GUI programming.

Our method reveals behavioral differences in an automated manner on the grounds of checked assertions for API contracts. Revealed differences are also referred to as *violations* (of compliance). Our method can be exercised with varying levels of scrutiny in terms of API contracts and corresponding assertions to be imposed on scenario execution. *Koloo* supports recording and replaying scenarios as well as comparing results of execution across original API and wrapper.

If we enable API contracts for return values of calls to API methods exercised by the application, then the violation is detected that SwingWT does not reproduce the return value for the method `contains(Point)` of class `java.awt.Polygon`. An investigation reveals that the *Tooltip* demo uses the method to map areas of the image to the corresponding messages. The violation is directly linked to the observable fact that a wrong message is displayed in the middle of Fig. 1.

If we also enable API contracts for asynchronous calls from the API to the application, then the violation is detected that SwingWT does not execute an asynchronous callback to the method `contains(int, int)` of class `java.awt.Component`. An investigation reveals that the *Tooltip* demo overrides the method to select the appropriate tooltip to display—specific to the component. The violation is also directly linked to the observable fact that a wrong message is displayed in the middle of Fig. 1.

If we also enable contracts for state integrity for method-call receivers of API types, then a violation is also flagged for the diverging background color, which cannot be revealed though by looking at return values of method calls because the scenario's design is such that the application does not exercise the (getter of the) background color in any way. It must be noted that additional scrutiny in terms of API contracts may also imply noise in reporting violations (see §6) in so far that elimination of these violations does not need to be reasonably expected by a 'good enough' wrapper.

[1] `http://gsd.uwaterloo.ca/issta2012`
[2] `http://eclipse.org/swt`
[3] `http://swingwt.sf.net`
[4] `http://swingwt.svn.sf.net/viewvc/swingwt/swingwt/trunk/CHANGELOG?revision=86`
[5] Source code see Sun JDK folder `demo/jfc/SwingSet2/`

3. INTERVIEWS

In order to better understand what process developers employ for API migration in practice we conducted interviews with developers with experience in some form of API migration (wrapping or rewriting).

We performed 5 guided interviews by phone and 1 informal interview by email. The subjects came from 3 different countries, had at least 5 years of professional development experience and participated in at least one API migration effort, in domains such as banking, gaming, GUI, web services, and logging. We asked general *open questions* about i) the used development process, ii) the used techniques and criteria for verification of results, and iii) the actual or desired use of tool support. The outcome of the interviews can be summarized as follows.

Specific applications — all reported API migration efforts used specific applications to drive the development process. Even wrapping projects aiming at general wrappers initially targeted a single application. For instance, SWTSwing[6], which is one of the wrappers covered by the interviews, was developed initially to migrate *Eclipse* from SWT to Swing. That is, SWTSwing essentially flips the role of original and replacement GUI APIs, when compared to the SwingWT wrapper that we discussed in §2. No interview was conducted though with the developer of SwingWT.

Simple samples — even with specific applications in mind, developers usually start with basic samples. In the case of API migration by rewriting, developers reported to target samples initially so that they can assess the migration effort. In the case of API migration by wrapping, developers reported to create small samples that imitate API usage in the application. Such samples are then used to evolve the wrapper up to the point that the original applications start executing with it. (In addition to hand-coded samples, developers also reported to leverage (simple) open source applications that exercise the original API.)

Limited test automation — developers reported that, in most cases, they simply try to informally compare the behavior of the application before and after migration, for example, by looking at the resulting GUI snapshots, files, or the return value of some important functions. In fact, a simple 'crash-driven' method is often used, such that developers try to run the application under migration and incompleteness or non-compliance of the wrapper reveals itself as a crash to be debugged and analyzed by developers.

One interviewee reported an approach for a more rigorous testing process in banking applications. Developers first recreate requirements documents to account for missing, incomplete, or outdated requirements documents. Then, developers derive use cases from which they in turn derive high-level, automated test cases manually so that essential business flows can be verified automatically.

Relaxed behavioral equivalence — all developers of wrappers mentioned that strict behavioral equivalence between the original API and wrapper was never a goal. More generally, all developers did not even strive for strict behavioral equivalence between the original and the migrated application. These expectations are also aligned with our experience that the differences between APIs are often too hard to be neutralized completely. Strict equivalence is reportedly considered unnecessary. Developers are content with a migration that is 'good enough'.

Related to the issue of behavioral equivalence, in the case of wrapper-based API migration some developers reported that limited, manual adaptation of the application is also applied in some cases, when this simplifies the wrapper implementation or enables better utilization of the replacement API. Migrated SWTSwing applications, for example, may benefit from added initialization code related to threading.

Conceivable tool support — developers varied in their opinions about conceivable tool support. Some developers expressed interest in automated comparison of results. In certain domains, e.g., in the GUI domain, developers expressed concerns about the feasibility of automated comparison. For instance, developers of SWTSwing used a tool for opening side-by-side the same application—once with SWT, once with SWTSwing, thereby simplifying the manual comparison of states of the GUI including details of appearance. A general, automated comparison was considered infeasible.

One developer said that automated rewriting tools should not be trusted for complex migration tasks because the code may become unmaintainable while the same could be true for generated wrappers. Finally, one developer expressed that debugging and regression testing may benefit from the selective use of injected observation functionality to be applied to objects of interest.

4. COMPLIANCE TESTING

We introduce the notion of compliance testing for API migration, as a form of automated testing, to explicitly capture the behavior of the original API in a manner that behavioral compliance can be asserted for the wrapper. Compliance testing can be seen as a blend of regression, differential [15] and compatibility testing [12] based on developer-designated test cases (as opposed to test-data generation) that are enriched by the generation of assertions, also common elsewhere in the testing area [31].

While regression testing is focused on preserving test-based contracts of an earlier version in a later version, compliance testing is concerned with with alternative implementations (the original API is now the point of reference). Differential testing is more general than regression testing in that it helps comparing different implementations, akin to compatibility and compliance testing. However, differential testing traditionaly focuses on random input generation and uses application output as oracles whereas compatibility testing compares behavior via inferred boolean invariants, and compliance testing generates assertions. When compared to other assertion generation approaches, the proposed form of compliance testing is specifically focused on the interaction between an application and an API with coverage of callbacks, state observation, and tuning of any sort of comparison.

4.1 Trace Capture

The execution of a compliance test is captured as a trace, which consists of a sequence of events that represent method calls, constructor calls and field accesses to *API-defined* elements. An element is API-defined if it is declared by the API or it is an application extension of an element declared by the API. Thus, a trace is meant to abstract from execution events that do not involve API types or subtypes thereof in the application.

Fig. 2 defines the structure of events more precisely. An event is an eight-tuple uniquely identified by its execution

[6] `http://swtswing.sf.net`

event	E	::=	$(e_{id}, t_{id}, p_{id}, d, T, sig, P[], R)$
execution id	e_{id}	$\in$	$\mathbb{N}$
thread id	t_{id}	$\in$	$\mathbb{N}$
parent id	p_{id}	$\in$	$\{e_{id}\}$
direction	d	$\in$	$\rightarrow \mid \hookleftarrow$
event entity	T,P,R	::=	$(type, content)$
entity content	$content$	::=	$\_ \mid value \mid id \mid (id, entity[])$
signature	sig	$\in$	$string$
type name	$type$	$\in$	$string$
object id	id	$\in$	$\mathbb{N}$
content value	$value$	$\in$	$string \mid primitive$

Figure 2: Event structure of captured traces.

```
e1, t0  → _:Robot.<init>():0 ...
e16, t0 → _:ToolTipDemo.<init>(_):1 ...
e22, t0    → _:JPanel.<init>():2 ...
e29, t0    → _:ToolTipDemo$Cow.<init>(1):3 ...
e32, t0    → _:Polygon.<init>():4 ...
e62, t0    → 3:ToolTipDemo$Cow.setToolTipText("Cow"):V ...
e66, t0    → 2:JPanel.add(3):3 ...
e68, t0 → _:JFrame.<init>("ToolTip Demo"):5 ...
e72, t0 → 5:JFrame.getContentPane():6
e73, t0 → 6:Container.add(2, "Center"):_ ...
e77, t0 → 5:JFrame.show():V ...
e85, t0 → 0:Robot.mouseMove('342', '312'):V
e86, t0 → _:Thread.sleep('3000'):V
e87, t1 ↩ 6:Container.contains('155', '112'):'true'
e88, t1    → _:Point.<init>('155', '112'):7
e89, t1    → 4:Polygon.contains(7):'true'
e90, t1    → 3:ToolTipDemo$Cow.setToolTipText(
      "<html><center><font color=blue size=+2>
      Mooooooo
      </font></center></html>")):V
```

Figure 3: API trace for the example in Fig. 1.

id, which captures the order in which events were detected. A thread id identifies the active thread. The parent id indicates the state of the execution stack–event e_{id} occurred in the control flow of its parent event p_{id}. The event signature identifies the target API element, including the static target type. From the viewpoint of the application an event is either outgoing ($\rightarrow$), if it originates from an application type, or otherwise incoming ($\hookleftarrow$). Incoming events express callbacks, when the API calls a method implemented by the application. A callback is *synchronous*, if it is the response to a call in the same thread. It is asynchronous otherwise, i.e., if it is a top-level event of a thread that is controlled by the API. (In this case, the trace contains only callbacks at the top-level, meaning that the API created the thread.)

The remaining items correspond to the target object, a sequence of parameters and a return object, all encoded as event entities. An entity represents a runtime value or object as a tuple with a string (the runtime type of the entity) and a content item. The content can be _ (null or void), a value (primitive or string), an object reference id or the reference to an array (which has an id and points to a sequence of entities). Exceptions thrown by events are encoded by the return value being a reference with the exception type.

Fig. 3 shows an excerpt for the actual trace underlying the motivating cow example of §2. No parent id is included as it is clear from indentation. This is the trace as captured with the original Swing API. (The trace is edited for readability. For instance, type names are abbreviated.) The trace contains calls to the two different `contains` methods that we encountered during the discussion of behavioral differences in §2: one for the area check (e_{89}) and one for the missing callback (e_{87}). The last event assigns the specific tooltip's text using HTML format. As an aside, Swing is capable of handling HTML, whereas SWT does not support HTML here. This API difference eventually led to an observable GUI difference, which was resolved during wrapper development such that HTML was stripped off.

4.2 API Contracts

Trace re-execution is subjected to API contracts, which are checked by assertions. Different forms of API contracts can be selected by the developer. (This is similar to control available in other forms of testing, e.g., the different controls for test-data generation in randomized testing [14]). The generation of assertions is also used in other forms of testing [31]. Technically, *Koloo* provides a corresponding DSL to select contracts.

Value Equality — when a method call or field access returns a value (or a string) it must *equal* the value observed at capture time.

Reference Integrity — when a method call or field access returns a reference (i.e., an object) its identity must *equal* the identity observed at capture time modulo consistent renaming of identities.

Value equality and reference integrity are also applied to arrays and their contents. That is, the array must preserve reference integrity and its contents must all preserve value or reference integrity, depending on the element type.

Exception Conformance — if a method call throws an exception during trace capture, it must also throw an exception of the same type during re-execution. Furthermore, re-execution must not throw any additional exceptions.

Callback Conformance — if a callback was triggered during trace capture, it must also be triggered during re-execution. To this end, the event for triggering a callback must be verified by associating it with an actual callback. Invocation of callbacks is tracked globally so that they can be matched with triggering events.

In the case of synchronous callbacks, the association between callbacks and the (preceding) triggering event is unambiguous. In the case of asynchronous callbacks, the triggering event could date back more arbitrarily. In fact, there is no guarantee that (multithreaded) callbacks re-execute deterministically in the order of trace capture [8] and deviations from the captured order do not imply incorrect behavior. Assertions are checked hence conservatively such that a check for re-execution of the callback is issued at the preceding event (which is not necessarily the trigger) and the check does not need to succeed immediately, but it is attempted repeatedly and in parallel to re-execution (in a separate, auxiliary thread), subject to a large timeout.

Set/get Integrity — subject to name-based patterns, certain pairs of API methods for object modification and observation are subjected to integrity assertions. For instance, a *set* followed by a *get* should retrieve the argument of the set. More precisely, if a method for assumed object modification is executed and there is an associated object observer, then the observation must retrieve the argument of the modification. These are the assumed pairs of modifiers and observers that can be selected for such contracts; we use X for variable postfixes of method names and T for type names:

Modifier	**Observer**	Comment
`set`X(... v)	`get`X()	getter returns v
`set`X(`boolean` v)	`is`X()	predicate returns v
`add`X(... v)	`get`X`s`()	result contains v
`add`(T v)	`get`T`s`()	result contains v
`put`(... k, ... v)	`get`(k)	getter returns v

Such state integrity is only required for the re-execution of the trace if it was valid during trace capture. It may be reasonable that APIs do not satisfy 'laws' as this. For instance, a setter may handle nulls or perform normalization.

State Integrity — if an object on a receiver position during trace capture admits observations based on name patterns, then value equality and reference integrity must hold for the corresponding observations during re-execution. Method names for observations are assumed to start with "get", "has", "is", or "count". Such observation can be controlled by opt-in or opt-out for types. Such observation may also be controlled to be iterated in depth such that results of observers are observed again etc. (We assume 'shallow' observation by default.) Finally, such observation may also be declared in a 'point-wise' manner such that specific receiver types are associated with specific observations including arbitrary closures over the receiver object.

DSL-based declaration of the tuning

```
org/apache/bcel/classfile/
  JavaClass.getBytes : BytecodeTuning
```

Match operation for the tuning

```
public class BytecodeTuning implements ... {
  public boolean matches(byte[] expected,
                         byte[] actual) {
    return serialize(expected).equals(
             serialize(actual));
  }
  private static String serialize(byte[] bytecode)
    StringWriter w = new StringWriter();
    new ClassReader(bytecode).accept(
      new TraceClassVisitor(new PrintWriter(w)), 0);
    return w.toString();
  }
}
```

Figure 4: BytecodeTuning implementation.

4.3 Assertion Tuning

The major source of control in compliance testing is selection of contracts for which assertions are to be checked, as we discussed above. A further source of control is tuning of the semantics of assertions, as we will discuss now. For instance, value equality can be relaxed. Also, non-deterministic, non-repeatable behavior can be accounted for. Technically, *Koloo*'s DSL, which we mentioned above in the context of API contracts, also serves the declaration of tunings: declarations are of the form Package/Type.Method : NameOfTuning.

Tunings can be applied to results of a certain reference type or in a more point-wise manner to results of specific methods. Conceptually, a tuning consists of a match operation for expected and actual result (i.e., the results captured originally and during re-execution).

Fig. 4 demonstrates an example in more detail. The 'BytecodeTuning' occurred in a wrapper study in the domain of byte-code engineering (see §6). The example is concerned with the comparison of the result returned by the central `getBytes` method of the API, which is supposed to return the byte-code of a given Java class; see the upper part of the figure for the DSL phrase that applies a tuning to `getBytes`.

```
org/apache/bcel/generic/
  InstructionHandle.getTargeters : ArrayIsSet
```

Compliance of byte-code engineering APIs: a specific method's result of an array type is declared to have set semantics.

```
java/awt/
  Component.getHeight : Percentage 10
```

Compliance of GUI APIs: a percentage-based divergence between expected and actual height of a component is admitted.

```
java/awt/
  Point.x : Margin 15
```

Compliance of GUI APIs: a margin-based divergence between expected and actual value of a point's coordinate is admitted.

```
javax/swing/
  JFrame.pack : SkipPackCallbacks
```

Compliance of GUI APIs: an API-specific tuning that turns off checking of certain callbacks (not listed here) that are triggered by calling `pack()`.

Figure 5: Diverse tuning examples.

The challenge is that the original API and the replacement API may use different orders for the constant pool that is part of the returned byte code sequence. When serializing the bytecode into strings, then constants are essentially inlined and hence this difference is neutralized. The replacement API readily provides a visitor for serialization, which is accordingly used by the `match` operation in the figure.

5. METHOD

We propose a method for wrapper development, which is motivated by our prior experiences with API migration [3, 2] and the results of the interviews with experienced developers; see §3. Accordingly, the input of the corresponding process is an application (or several applications) subject to wrapper-based migration with regard to a specific API couple; the output is a wrapper, which is supposed to be 'good enough'. Also, the method is iterative to improve the wrapper in a stepwise manner. Overall, the method provides a level of automated testing that improves state of the art in API migration; see, again, §3.

Each iteration is composed of three phases (see Fig. 6) inspired by capture and replay approaches [20, 24, 12]. Developers first extract a dynamic trace of the interaction between the application and the API; see §5.1. (Of course, an iteration could also deal with multiple traces at once.) Then, an interpreter is used to re-execute the trace so that it can be validated in the sense that the trace mocks the behavior of the original application—still using the original API; see §5.2. In this phase, settings are aggregated to decide on API contracts to be asserted. Ultimately, the interpreter is used with the wrapper under development and activities of wrapper evolution and configuration alternate; see §5.3. In this phase, the final decisions on applicable API contracts and corresponding tunings are made. The violations of contracts guide the evolution of the wrapper. The method is supported by the *Koloo* toolkit.

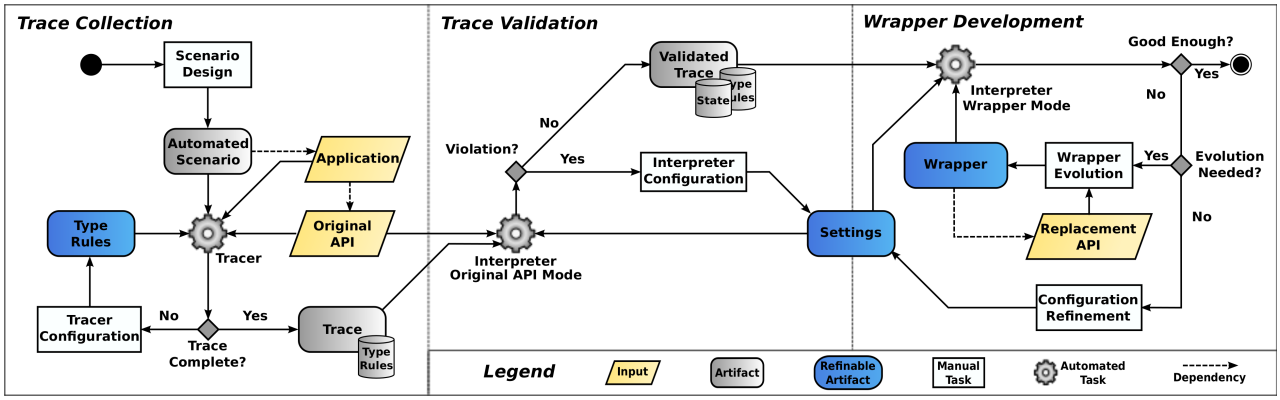

Figure 6: An overview of an iteration according to the *Koloo* method.

5.1 Phase: Trace Collection

Scenario Design — Trace collection starts with the design of scenarios that can be automated as tests eventually. A scenario is an execution of the application using the original API. It is important to design scenarios that cover the desired uses of the application since they represent the requirements for the migration: the wrapper will be considered compliant if it properly replaces the original API in these scenarios with respect to the verified contracts.

Scenarios usually target the applications at hand (as opposed to directly using the API). Tests for an application, if available, can be used as scenarios. Such tests usually contain assertions related to application logic, which may or may not also check (indirectly) some aspects of API usage. The interpreter of the method enriches scenarios—both with and without assertions for application logic—with assertions for selected API contracts.

Traces that are derived from scenarios must not depend on external actions, and all results should be reproducible—this is common requirement for automated testing. In the context of API migration, this requirement may imply extra challenges though. Consider, for example, API migration for GUI APIs: GUI actions would need to be automated by using a record&replay tool (such as GUITAR [16]) or by manually writing scripts that automate actions upon the GUI. In the example of Fig. 1 we designed a scenario that positions the mouse over various areas of the image, and automated the scenario with AWT's `Robot` class.

It is the responsibility of the developer to design scenarios that explore interesting interactions with the API. Depending on the selected contracts, those interactions are more or less strongly verified. As a baseline, let us assume that these contract forms are selected: value equality, reference integrity, exception conformance, and callback conformance—we also refer to these contracts as 'basic contracts'. With these contracts, verification will essentially not look into objects (such as receivers of method calls). Scrutiny of verification is hence increased when state integrity is also selected, but this may also be problematic, as we will discuss in §6.

Tracing — The tracer executes the scenarios and collects the API interactions as traces. (*Koloo* uses the bytecode engineering framework ASM[7] for tracing.) To this end, the tracer is also configured with a description of the types that belong to the application and the API; this metadata is also incorporated into the traces. Trace collection should abstract away details of the underlying application that are not important from an API compliance testing point of view. (This also helps with automating the scenario and making it reproducible.) Such abstraction is similar to other approaches for extracting regressions tests from application traces [7, 20, 24, 23].

Type Rules — Developers need to identify application and API types. Such identification boils down to 'type rules' such as regular expressions over their fully qualified names, e.g., *java.awt.**, *javax.swing.** and *javax.accessibility.** in the case of Swing as the original API. However, the interaction between application and API may be mediated by *auxiliary types* that are essential for the scenario even though they should not be migrated. For example, APIs often use collections, iterators, and files. In order to reproduce the scenario in subsequent phases, events for such auxiliary types must also be incorporated into the traces. The tracer reports on unbound object references in API calls, and developers need to add type rules for auxiliary types until traces can be constructed completely. (In the study of §6, only few such rules were needed: one scenario needed 8 auxiliary types but most did not need any).[8]

Additional type rules are needed for rewriting type names such that the interpreter can eventually deal with possibly divergent namespaces for wrappers, when compared to the original API. For instance, the SwingWT wrapper of §2 requires the type rule `java.awt` ⇒ `swingwt.awt`.

5.2 Phase: Trace Validation

Traces must be validated before they are even attempted with the emerging wrapper. By validation we mean that interpretation of the trace is checked to reproduce the captured behavior. We also include the effort for contract selection, including the corresponding aggregation of state that is needed by contracts but not readily available from the trace, in particular state for checking state integrity; see §4.2.

Interpretation — The interpreter mocks the application by reproducing its role in the interactions recorded in the trace. The interpreter builds a model of all types and methods from the trace so that mock applications types and methods can be provided for trace execution; also, assertions for the selected contracts (see 'Settings' in Fig. 6) can be injected. In particular, each incoming event is associated with a mock application type; each mock application method can

[7] `http://asm.ow2.org`

[8] *Koloo* defines Java's `Thread` and `Runnable` types to be auxiliary types by default, thereby being able to mock multi-threaded applications.

be seen essentially as a sequence of API interactions, i.e., outgoing events. Additional rules apply to model parent relationships for events. Events without parents are stored in a special `main` method. (*Koloo* uses a custom classloader that generates byte-code for the mock application types with method implementations that call back into the interpreter.)

Interpretation starts with the `main` method and proceeds sequentially. Each outgoing event is executed in turn. The interpreter keeps a map from trace object id to runtime object so that it can determine the target and parameters of events; primitive values are used directly. After returning from an event execution, the interpreter registers the return value in the object map and uses the available contracts to verify the result. Contracts can issue fatal violations that immediately interrupt interpretation or can register the violation but allow interpretation to continue. After the last event is executed the interpreter reports the set of detected violations. If no violations were detected, the trace is considered 'validated'.

Reproducibility — Validation of reproducibility for the original API assures interpretation does not attempt to verify in the wrapper behavior that is even unreproducible with the original API. For instance, if the trace contains a call to a `time()` method, then the corresponding event will not be validated and will be accordingly omitted from the verification of the wrapper. Such omission of events that fail to be reproducible also decreases the chance of attempting to verify undeterministic behavior common in multi-threaded APIs. The reproducibility may obviously depend on the selected contracts. For instance, the broad use of contracts for state integrity may potentially invalidate objects from the trace, thereby providing feedback to developers so that they may backtrack on their selection.

5.3 Phase: Wrapper Development

Trace collection and validation prepares actual development (evolution) of the wrapper. The same interpreter as in trace validation is used to verify the traces for the wrapper, while type rules for type-name rewriting may now apply, and aggregated state from trace validation may be used as well.

If interpretation does not result in a violation, the validated trace is deemed 'asserted', meaning that the wrapper is known to comply with the corresponding scenario, modulo tunings, with respect to the verified contracts. If desired, the level of scrutiny can now be increased by selecting stronger contracts and repeating the process. The set of asserted traces constitutes a regression suite that should be used any time the wrapper evolves, hence guaranteeing that no regression goes undetected. *Koloo* contains a corresponding regression running tool to automate this part of the process.

Wrapper Evolution — Any given violation may imply that developers need to evolve the wrapper. However, violations may also be addressed in other ways. That is, developers may also decide to add a tuning to relax the API contracts; see §4.3. Alternatively, developers may decide to reconfigure the setting such that the scrutiny of API contracts is lowered by selecting less API contracts or opting out of specific cases of asserting the contracts.

By the end of an iteration the wrapper demonstrably complies with the original API in the exercised scenarios. The specification of contracts in use documents the level of scrutiny while the specification of tunings capture precisely the unresolved differences between wrapper and original API.

6. EVALUATION

6.1 Research Questions

We designed a study to address the following overarching research questions:

RQ1 — How does our method compare with alternatives in terms of the ability to produce a compliant wrapper?

RQ2 — How do the various API contracts help in driving the evolution of wrappers in practice?

RQ3 — Which types of tunings are necessary and how often are they used?

6.2 Subjects

Table 1 presents general information for the applications, wrappers and APIs that were subjects of our study. (These artifacts and additional data are available from the paper's website.) We chose representatives of 3 domains and sought to cover a wide range of development methods for comparison. XOM2JDOM is a wrapper we developed driven by the test suite for XOM; it failed at around 40% of XOM's test cases [3]. Memoranda[9] is a scheduling tool that uses XOM to store data in XML files. Quilt[10] uses BCEL[11] to modify the byte-code of applications to perform code coverage analysis. We selected Quilt because it contains a large test suite, allowing the comparison against a method driven by application tests. Finally, *SwingWT* was developed in the traditional crash-driven style and offered the opportunity of consulting its developers for validation of our results.

6.3 Methodology

The study was conducted following the steps of our method.

6.3.1 Trace Collection

We designed scenarios for each application, collected traces and iterated over the process to resolve unbound references.

Memoranda — we designed a single scenario in which we opened and closed a complex schedule document. We performed the scenario manually in the GUI while tracing the interaction between Memoranda and XOM.

Quilt — we executed the 65 test cases in Quilt's suite. Out of those, we used the 33 that actually used BCEL.

SwingSet2 — we designed 59 scenarios, 2 to 6 for each of the 16 applications in the set, trying to simulate a user exploring the GUIs. For example, in the Button demo we selected all possible push buttons, check boxes and radio buttons. We automated the scenarios using AWT `Robot`.

6.3.2 Trace Validation

We validated the collected traces against the respective original APIs. We started using basic contracts and incrementally increased the scrutiny, first with set/get contracts and then with contracts for state integrity. Whenever necessary, we created tunings for the original API.

6.3.3 Wrapper Development

XOM2JDOM — we evolved XOM2JDOM based on the detected violations. We started with basic contracts, and subsequently used contracts for set/get and state integrity. After the wrapper was compliant, we re-executed XOM's test

[9] `http://memoranda.sf.net`
[10] `http://quilt.sf.net`
[11] `http://commons.apache.org/bcel/`

Subject$_{version}$ Type Definitions	**Types**	**Methods**	**NCLOC**
XML			
Memoranda$_{1.0-RC3.1}$ net.sf.memoranda.*	461	1684	23891
XOM$_{1.2.1}$ nu.xom.*	110	884	19395
XOM2JDOM$_{0.1}$ nu.xom.*	43	602	5203
JDOM$_{1.1}$ org.jdom.*	73	908	8755
Bytecode			
Quilt$_{0.6-a-5}$ org.quilt.*	67	579	4974
BCEL$_{5.2}$ org.apache.bcel.*	408	3386	27384
ASM$_{3.3.1}$ org.objectweb.asm.*	174	1526	23348
GUI			
SwingSet2$_{1.4}$ swingset2.*	131	425	6422
Swing$_{1.4}$ java.awt.* javax.swing.* javax.accessibility.*	2032	18702	234731
SwingWT$_{0.90}$ swingwt.awt.* swingwtx.swing.* swingwtx.accessibility.*	864	6537	33948
SWT$_{3.3.2}$ org.eclipse.swt.*	696	9607	99205

Each domain has an application on top followed by two APIs; *XML* and *GUI* also have a wrapper between the APIs they wrap. NCLOC stands for non-comment lines of code. SwingSet2 originally used the default package; we moved it to its own package.

Table 1: The subjects of the study.

suite. The goal was to check whether the suite was able to spot the same behavioral differences as *Koloo*.

BCEL2ASM — we created a BCEL2ASM wrapper from scratch. We started with an exception-throwing wrapper such that every method throws an `UnsupportedOperationException`. The chosen application, Quilt, can already be compiled with the initial wrapper. We then developed the wrapper using one trace at a time and only basic contracts. For each trace we would execute the interpreter, pick a violation and either evolve the wrapper or create a tuning. After each of these iterations we executed the suite of asserted traces to detect regressions. Furthermore, because each trace corresponds to the execution of a Quilt test case, we executed the original suite, which let us compare the contract checking of manually written test cases against *Koloo*.

We used mostly basic contracts; sometimes, however, we resorted to selective state-based contracts to verify the state of specific types of objects. In two occasions, we decided to add a call to an API object in the original Quilt test case because we wanted to observe a certain behavior at a particular point in the trace; the relevant behavior was not exercised by the application. When all traces were asserted for basic contracts, we collected compliance data for contracts for set/get and state integrity.

SwingWT — we used SwingSet2 validated traces in two ways. First, we verified the initial compliance of SwingWT with respect to all kinds of contracts, which provides an overview of the overall compliance achieved by SwingWT developers for each category of contracts. Second, we selected two applications of the set, Button and Tooltip demos, to evolve SwingWT. We proceeded as usual, evolving by resolving violations and tuning, using only basic contracts. We validated that the contracts were indeed valuable by submitting a patch to the developers of SwingWT.

6.4 Results

Table 2 shows an overview of the results.

XOM2JDOM — we needed 3 auxiliary types and validation occurred without violations. *Koloo* detected 4 violations that were corrected in the wrapper. By executing XOM's suite before and after the modifications we noticed that a single test case accounted for one of the state contract violations; 3 violations had no corresponding test case. Two of these relate to reference integrity and one to the execution of an unexpected operation (`getParent()` without parent).

BCEL2ASM — we needed 8 auxiliary types for Quilt because BCEL makes heavy use of JRE types like streams and classloaders. During validation we found one trace that our tool could not interpret (due to classloader use) and it was discarded. The BCEL reference integrity violation is due to a returned array with set semantics (Fig. 5). State contracts were violated twice due to side-effects caused by `MethodGen.getLocalVariables()`, and were tuned with *Skip*.

During the development of BCEL2ASM we resorted to 7 additional tunings. One was the BytecodeTuning presented in Fig. 4. The other 6 were methods with reference integrity problemas that we ignored (with Skip) because they were caused by our decision not to implement reuse of handles. In 2 occasions, state violations helped detect problems in the wrapper. Repeated re-execution of asserted traces resulted in detection of 19 regressions.

In all traces that presented violations (some passed directly due to previous evolution of the wrapper) the corresponding Quilt test case started to pass while *Koloo* still detected violations. That is, while fixing the contract violations detected by *Koloo* we also fixed the assertions manually written by Quilt developers. Furthermore, while trying to reproduce the behavior of BCEL using ASM we discovered 4 problems in BCEL. The problems were reported as 3 bugs and 1 enhancement proposal[12], and all were acknowledged by BCEL developers. Two bugs were detected by comparing the byte-code after BytecodeTuning tuned return value contracts, and the other was a reference integrity problem that detected a memory leak. The enhancement proposal concerns a custom sort algorithm used by BCEL that we had to reproduce (we proposed the use of a standard sort).

SwingWT — only one application of SwingSet2 needed an auxiliary type (from `java.bean`). During Swing validation *Koloo* detected 6 callback violations, all caused by missed asynchronous callbacks in an application that uses threads to paint a canvas. We found that 4 methods in Swing do not follow the set/get rule. When receiving a null parameter, one method sets a default and another ignores the call completely. Two other methods enforce bounds on the parameter value. Most state contract violations refer to side effects of getting size and position information, or state from unprepared objects (e.g., `getLocationOnScreen()` throws an exception if the widget is not showing). We created 2 tunings for callbacks and 10 for the other violations.

Koloo detected many contract violations in SwingWT verification. 53 value equality violations were about size and po-

[12]Complete list in `http://gsd.uwaterloo.ca/issta2012`

Memoranda	1 Scenario	3 Auxiliary Types
XOM Validation		**XOM2JDOM Development**
No violations		Reference 1
		State 3
		Tunings/Regressions 0
Quilt	32 Scenarios	8 Auxiliary Types
BCEL Validation		**BCEL2ASM Development**
Reference	1	Reference 50
State	2	Exception 5
Tunings	3	State 2
		Tunings 7
		Regressions 19
SwingSet2	59 Scenarios	1 Auxiliary Type
***Swing* Validation**		**SwingWT Verification**
Sync	3	Value 72
Async	3	Reference 4
Set/Get	4	Exception 1
State	6	Sync 32
Tunings	12	Async 6
		Set/Get 8
		State 100s
SwingWT Development		
Button	4 Scenarios	***Tooltip*** 5 Scenarios
Value	18	Value 1
Sync	4	Async 1
Tunings	7	Tunings/Regressions 0
Regressions	3	

Table 2: A summary of the study results.

sitioning, 12 about missing functionality and 7 about wrong state or default value in widgets. A total of 38 callbacks were found missing. SwingWT did not comply with the set/get contract in 8 methods: 6 were empty setters, one method wraps its parameter in another object, and another method explicitly deviates from the original API behavior (as per code comments). Hundreds of violations were detected with state contracts. Many were thread invalid access exceptions and 2 applications deadlocked; most violations were value equality problems and the side-effects of state gathering caused the GUIs to be distorted.

Most of the violations that drove the evolution of SwingWT were related to size and positions. We tried to solve them until we were content with the result, and applied tunings if necessary (3 Percentage, 2 Margin and 2 Skip, see Fig. 5). Asserted traces helped us detect 3 regressions while developing for *Button*. Violations detected for Tooltip were discussed in §2. Finally, SwingWT developers used our patch for these issues to release version 0.91.

6.5 Discussion

RQ1, Alternative Methods — of the 4 violations detected by *Koloo* in XOM2JDOM, only 1 could be detected by the original XOM test suite. This is evidence that applications exercise APIs in ways that even very complete test suites like XOM's may fail to foresee (like calling `getParent()` on a document without parent), or that the test case that asserts this behavior may be failing for other causes already (remember 40% of the test cases still fail). By concentrating on scenarios *Koloo* can better pinpoint the wrapper evolution that is needed by the application at hand. Furthermore, the 2 additional violations revealed internal wrapper bugs that would be seldom target of assertions in API test suites.

Another result of the study is that manually written Quilt test cases were subsumed by *Koloo*'s basic contracts with respect to the behavior they asserted. This suggests that the type of contracts used by developers in practice may be similar to the basic contracts *Koloo* checks, but they are not consistently applied to all interactions with the API.

We also have evidence that *Koloo* leads to a much more compliant wrapper than the customary crash-driven method. The verification of SwingWT showed that many compliance violations were detected, even with respect to an application that is used by the project to showcase itself. The fact that our patch for Button and Tooltip was accepted by SwingWT validates that developers were interested in solving the compliance issues detected by *Koloo*.

Finally, we should note that we observed three different ways of scaffolding wrappers. In BCEL2ASM we used exceptions to signal unimplemented methods (so called *exception throwing* approach); SwingWT uses *best effort* in that it leaves void methods empty and returns bogus values when necessary in hope that it is *good enough*; from our interviews we know that SWTSwing uses a *mixed* approach: best effort but logging unimplemented methods. If SwingWT used exception throwing it would be able to detect some of the problems due to missing functionality. However, that would be too pedantic because many client applications would easily crash. Therefore, we argue that a compromise is a *pedantic switch* that allows incomplete wrappers to be released in best effort while being developed with exception throwing.

RQ2, Contracts — the study showed that basic contracts represent a minimum level of compliance to be achieved by wrappers, modulo tunings. Basic contracts allowed us to develop BCEL2ASM to be compliant with Quilt's test suite, to find bugs in BCEL and to successfully evolve SwingWT. Value equality contracts helped detect missing functionality and deviations in size and positioning of widgets in SwingWT, and differences in the generated byte-code in BCEL2ASM. A reference integrity contract helped detect a small memory leak in BCEL. Callback conformance contracts were essential to understand which functionality of the API was being overridden by SwingWT applications.

Set-get contracts were useful to better understand the behavior of 4 methods in Swing. None of the corresponding methods in SwingWT complied with that behavior and this deviation did not show in the interaction with applications.

For relatively complete and compliant wrappers, such as XOM2JDOM, state-integrity contracts helped in finding compliance violations. However, state gathering often triggered side-effects that affected other contracts, even when applied to the original APIs (BCEL and Swing). Neither BCEL2ASM nor SwingWT are close to be compliant in terms of state integrity. State contracts naturally demand more of the wrapper than what the application directly needs, and may force the wrapper to implement additional functionality, but appear to be useful for mature, general wrappers.

RQ3, Tunings — we used a total of 15 tunings for validation and 14 tunings for development. In validation, tunings were used to accommodate undeterministic behavior and side-effects of state gathering, and to relax the semantics of strict equality contracts (e.g., `ArrayIsSet`). In development, tunings were additionally used to document unimplemented features (handle reuse in BCEL) and to compromise on features that are difficult to emulate perfectly (positioning and sizes of widgets). The results provide evidence of the need and usefulness of tunings.

6.6 Threats to Validity

The main threats to *internal validity* concern possible bugs in *Koloo*, errors in the collection of data during the execution of the study, and experimenter bias. Because we checked each violation manually it is unlikely that there are false positives; false negatives would not affect our results. We minimized experimenter bias by looking for external confirmation of findings, such as BCEL and SwingWT developers.

Threats to *external validity* include the subjects of our study and interviews. We mitigated this threat by selecting subjects in 3 different domains of API programming. We also sought developers with a varied background to interview, but were constrained by the availability of developers with experience in API migration.

7. RELATED WORK

Methods of API migration. Existing methods leverage only straightforward regression testing where applicable (i.e., "establish that a migrated application passes all available (migrated) tests."), while making orthogonal contributions—not related to testing. One group of methods addresses the implementation of wrappers or transformations for API migration [1, 29, 18, 11]. API mappings may be represented as refactorings, rewrite rules, and other specifications. Another group of methods aims at inference of mappings for API migration [26, 27, 33, 30, 17]. Such inference methods usually rely on additional assumptions. For instance, a readily migrated application may be used to extract mapping rules [33].

Differential and compatibility testing. Differential testing, as initially proposed by McKeeman [15], is a special case of random testing to detect differences between different implementations, e.g., different compilers for the same language. More recently, an approach similar to differential testing has been applied to COTS-based systems. BCT [13] is a technique to capture the behavior of components in a COTS system and generate invariants that represent the I/O and interaction behavior of components. These invariants represent an oracle that can be verified on competing implementations for compatibility [12]. The main differences are that *Koloo* allows for explicit tunings whereas BCT users can only ignore violations, and that BCT uses object flattening to store object state. Thus, BCT cannot check referential integrity contracts and relies on the ability of getting all relevant state from objects. BCT's approach targets large components. It is not clear whether the approach will be effective for APIs that allow many relationships among fine-grained objects. Furthermore, it is not clear whether I/O invariants capture asynchronous callbacks.

Regression testing. There is a large body of work in regression testing; hence, we concentrate our discussion on the approaches closer to *Koloo*. Elbaum et al. [7] propose differential unit tests to detect differences between versions of the same unit. In this approach the system state is serialized prior to the execution of a method. During regression testing, the state is deserialized, the method is executed and results are compared. Hoffman et al. [9] propose a framework for analyzing traces and present RPRISM, a tool that analyzes traces to identify the cause of regressions. Our method and *Koloo* are mostly inspired by the test factoring [24] and SCARPE [20] techniques. These approaches capture the interaction of an application with a certain module and mock the module's behavior via a replay system that reproduces return values and callbacks. These approaches can be used, for example, to abstract costly APIs during regression tests. GenUTest [23] further generates smaller unit tests from the traces, one for each trace event that returns some value. *Koloo* presents many differences with these systems, including the verification of asynchronous callbacks, contracts for state integrity, and support for tunings. The approach of [7] is state-based in that it stores the necessary state of the system prior to a method execution; *Koloo*, test factoring, SCARPE and GenUTest are action-based since they restore the state by reproducing the original events.

GUITAR [16] is a framework for automated model-based GUI testing. GUITAR analyzes a GUI application at runtime and builds a model of the available user interactions. A tool then generates regression test cases by traversing the GUI in various ways and collecting state information. A replayer can re-execute the regression tests to detect differences in the state information across versions of the application. In contrast to *Koloo*, GUITAR is domain specific and needs platform-specific rippers to create GUI models.

Automated test generation. There are methods for test-suite generation for object-oriented programs that readily apply to API implementations including, potentially, wrappers. Jartege [19] performs random generation of unit tests based on JML specifications. The Korat framework [4] generates systematically all non-isomorphic test cases for the arguments of a method under test using an advanced backtracking algorithm that monitors the execution of predicates for class invariants. For these methods to be applicable, the API implementations would need to provide detailed contracts; such contracts are not available for APIs in the migrations we have performed or encountered.

Eclat [21] performs dynamic inference of certain (API) properties (manifested as assertions) from the execution of a given test suite, and it uses the inferred properties in generating further possibly fault-revealing test cases. Randoop [22] generates random sequences of calls to a given API and uses dynamic feedback to guide the generation of additional sequences. Randoop checks default contracts (exception throwing and equals) and allows users to implement additional contract checking code. Orstra [31] is a tool for augmenting generated tests with regression oracle testing. *Koloo* implements the interface-based object-state contract proposed by Orstra; the concrete state contract is not implemented since wrappers usually do not contain the same private fields as the classes they are replacing.

Various approaches were proposed with the main goal of improving structural branch coverage: Ocat [10] captures and stores object state during execution and mutates the state during random testing, Palus [32] uses a combination of static and dynamic analyses to generate behaviorally diverse tests, while DyGen [28] combines dynamic trace analysis with symbolic execution. All these methods are not well aligned with the objectives of wrapper development, where the testing effort should be limited such that the application under migration passes its existing tests enriched by assertions for simple API contracts.

8. CONCLUSION AND FUTURE WORK

In this paper we presented a notion of scenario-based compliance testing for API migration. We described a method for developing compliant API wrappers and its implementation in the *Koloo* toolkit. We evaluated the method in a study involving compliance assessment, evolution and development from scratch of wrappers for object oriented APIs in the domains of XML processing, byte-code engineering and GUI programming. Our evaluation provides evidence that the method is effective in improving wrapper compliance, that API contracts help driving the evolution of the wrapper and that contract tuning is both necessary and useful.

The study showed that the violations uncovered by *Koloo* helped in understanding the deviation from the original API. However, finding the cause of the deviation in the wrapper and fixing it are orthogonal problems for which *Koloo* gives little help. Delta-debugging [5] might be promising in helping determine the root cause, although its applicability to APIs with long start-up times like Swing is not clear. The same work proposes a technique to automatically determine which objects to trace from an initial trace. This technique could also be adapted to support the discovery of needed auxiliary types in the type rules (see §5.1). Finally, techniques for removing non-determinism from replaying of multi-threaded applications [6] could be used to better support the verification of asynchronous callbacks.

9. REFERENCES

[1] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *OOPSLA '05*, 2005.

[2] T. T. Bartolomei, K. Czarnecki, and R. Lämmel. Swing to SWT and Back: Patterns for API Migration by Wrapping. In *ICSM'10*, 2010.

[3] T. T. Bartolomei, K. Czarnecki, R. Lämmel, and T. van der Storm. Study of an API Migration for Two XML APIs. In *SLE'09*, pages 42–61, 2010.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA'02*, 2002.

[5] M. Burger and A. Zeller. Minimizing reproduction of software failures. In *ISSTA'11*, 2011.

[6] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, 1998.

[7] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *FSE'06*, 2006.

[8] M. Fowler. Eradicating Non-Determinism in Tests, Apr. 2011. Blog post `http://martinfowler.com/articles/nonDeterminism.html`.

[9] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *PLDI'09*, 2009.

[10] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. Ocat: object capture-based automated testing. In *ISSTA*, 2010.

[11] P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration. In *OOPSLA '10*, 2010.

[12] L. Mariani, S. Papagiannakis, and M. Pezzè. Compatibility and regression testing of cots-component-based software. In *ICSE'07*, 2007.

[13] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *Int. Conf. on Eng. of Complex Computer Systems*, 2005.

[14] P. Maurer. Generating Test Data with Enhanced Context-free Grammars. *IEEE Software*, 7(4):50–56, 1990.

[15] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 1998.

[16] A. Memon, A. Nagarajan, and Q. Xie. Automating regression testing for evolving gui software. *Journal of Software Maintenance and Evolution*, 2005.

[17] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen. A graph-based approach to api usage adaptation. In *OOPSLA*, 2010.

[18] M. Nita and D. Notkin. Using Twinning to Adapt Programs to Alternative APIs. In *ICSE'10*, 2010.

[19] C. Oriat. Jartege: A Tool for Random Generation of Unit Tests for Java Classes. In *QoSA'05 and SOQUA'05*, 2005.

[20] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *WODA '05*, 2005.

[21] C. Pacheco and M. D. Ernst. Eclat: Automatic Generation and Classification of Test Inputs. In *ECOOP'05*, 2005.

[22] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, 2007.

[23] B. Pasternak, S. Tyszberowicz, and A. Yehudai. Genutest: a unit test and mock aspect generation tool. *Int. Journal on Software Tools for Technology Transfer*, Vol. 11, October 2009.

[24] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE'05*, 2005.

[25] I. Şavga, M. Rudolf, S. Götz, and U. Aßmann. Practical refactoring-based framework upgrade. In *GPCE '08*, 2008.

[26] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE '07*, 2007.

[27] W. Tansey and E. Tilevich. Annotation Refactoring: Inferring Upgrade Transformations for Legacy Applications. In *OOPSLA'08: Proc. of the 23rd Conference on Object-Oriented Programming Systems Languages and Applications*, 2008.

[28] S. Thummalapenta, J. de Halleux, N. Tillmann, and S. Wadsworth. Dygen: automatic generation of high-coverage tests via mining gigabytes of dynamic traces. In *Int. Conf. on Tests and Proofs*, 2010.

[29] V. L. Winter and A. Mametjanov. Generative programming techniques for java library migration. In *GPCE '07*, 2007.

[30] W. Wu, Y. G. Gueheneuc, G. Antoniol, and M. Kim. AURA: A Hybrid Approach to Identify Framework Evolution. In *ICSE'10*, 2010.

[31] T. Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, 2006.

[32] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA'11*, 2011.

[33] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API Mapping for Language Migration. In *ICSE'10*, 2010.