Electronic Communications of the EASST
Volume 67 (2014)

Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014)

Towards Dynamic Reverse Engineering Visual Contracts from Java

Abdullah Alshanqiti and Reiko Heckel

12 pages

Guest Editors: Frank Hermann, Stefan Sauer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Towards Dynamic Reverse Engineering Visual Contracts from Java

Abdullah Alshanqiti[1] **and Reiko Heckel**[2]

[1] amma2@mcs.le.ac.uk [2] reiko@mcs.le.ac.uk
Department of Computer Sciences, Leicester University, UK

Abstract: Visual contracts provide a concise and intuitive representation of pre- and postconditions for operations in object-oriented or component-based systems, which can be used for documentation, testing, or simulation. However, defining visual contracts to correctly describe the behaviour of existing classes or components requires a deep understanding of their data model and behaviour.

We propose an approach to automatically extract instantiated versions of visual contracts, or *contract instances*, by observing the changes an operation performs on the objects in a system. We describe and evaluate the approach and tool to extract contract instances using the case study of Java-based DOM implementation NanoXML.

Keywords: graph transformation, rule learning, rule extraction

1 Introduction

Visual contracts model the pre- and postconditions of operations of classes or components by graph transformation rules. They are based on an operation's signature, but go beyond interface description in that they specify the transformation of a hypothetical data state, just as protocol state machines use a hypothetical control state to describe sequences of invocations. Visual contracts provide a precise yet high-level behavioural model that can be used for documentation, model-based testing [KRH12, RKH13], simulation and formal verification. However, the correct definition of visual contracts for an existing component requires a deep understanding of its implementation. Automated reverse engineering can enable their wider use in testing and verification but also provide a valuable tool for program understanding.

The extraction of behavioural models from implementations can be performed statically by examining the source code to capture all possible behaviours [BLL06]. However, dynamic binding in object-oriented software affects the accuracy of static methods, often leading to an over-approximation of effects or dependencies. We propose a dynamic reverse engineering approach, extracting instantiated versions of visual contracts by observing runtime changes in object structures when an operation is executed.

Each contract instance will only capture the behaviour of a specific invocation of the operation. It can therefore be described by a single rule, whose precondition captures the objects that were read only, while the postcondition describes how the object structure changed during the invocation. While each contract instance only represents one possible outcome of the operation's invocation, jointly they can be input to an algorithm [AHK13] (proposed at last year's GT-VMT) for learning graph transformation rules from examples.

Dynamic reverse engineering requires to trace a system's execution, e.g., by instrumentation of the code, as well as the analysis of the traces to extract behavioural models. We instrument Java bytecode using AspectJ to observe the internal state of the system and its changes during execution. The recorded sequence of elementary read and write operations is then analysed to construct the contract instance.

In Sect. 2 the intended mapping between the internal structure and its external representation in the model is discussed. In Sect. 3 we explain in detail how tracing the execution and analysing the resulting log we can generate contract instances. These are evaluated in Sect. 4 before we discuss the related work in Sect. 5 and conclude with Sect. 6.

2 Representing Object-oriented Structure and Behaviour

We explain informally how Java class, object structures and methods are represented by class, object diagrams and visual contracts, respectively. We will use the NanoXML API[1] as example and refer to Figure 1 for illustration.

2.1 Java Classes and Objects Structures

NanoXML is a small non-validating XML parser for Java, which provides a light-weight and standard way to manipulate XML documents. It consists of three packages and 24 Java classes. The left of Figure 1 shows code fragments of Java classes `XMLElement` and `XMLAttribute` and their corresponding classes in the class diagram under (A). Fields of Java classes are represented as UML attributes or associations depending on their type, i.e., fields of primitive types lead to UML attributes, while fields of object type turn into associations. For example, the fields declared in lines 8 to 12 are represented as attributes while the `parent` field in line 5 becomes a reflexive 0..1 association. Fields of collection type such as `children` in line 7 could be represented as $*$ associations, but we prefer to represent collections explicitly because they provide information about the organisation of their elements. For example, elements of a `vector` in (A) are known to be ordered.

The choices made in representing class structures determine the representation of object structures. For example, (B) shows fragments of two states representing objects referred to as p and c and their changes due to the invocation `p.addChild(c)`.

2.2 Behavioural Code

Each visual contract describes the pre- and postcondition of a possible way of executing a method. Consider (B), which shows relevant objects of two states resulting from an execution of the `addChild()` operation in lines 14 to 21. The rule extracted from this execution is shown in (C). It describes how the objects and their attributes and links change from one state to the other. In the code this is achieved by assignments, which represent the creation and deletion of objects and links or the update of attributes. For instance, the assignment in line 19 represents the creation of an edge. If accessed objects remain unchanged, they become context elements in the

[1] http://nanoxml.sourceforge.net/orig/

rule. Such objects can be part of conditions, such as in line 15. Note that while we can extract information about deletion of links, we are unable to detect object deletion, which is handled implicitly by Java's built-in garbage collector.

```
1  package net.n3.nanoxml;
2
3  public class XMLElement implements IXMLElement {
4
5     private IXMLElement parent;
6     private Vector attributes;
7     private Vector children;
8     private String name;
9     private String fullName;
10    private String namespace;
11    private String content;
12    private String systemID;
13    ...
14    public void addChild(IXMLElement child) {
15       if (child == null) {
16          throw new IllegalArgumentException("child must
             not be null");
17       }
18       ...
19       ((XMLElement)child).parent = this;
20       this.children.addElement(child);
21    }
22
23    public void setAttribute(String name, String value) {
24       XMLAttribute attr = this.findAttribute(name);
25       if (attr == null) {
26          attr = new XMLAttribute(name, name, null, value,
             "CDATA");
27          this.attributes.addElement(attr);
28       } else {
29          attr.setValue(value);
30       }
31    }
32    ...
33 }
34 ...
35 class XMLAttribute
36 {
37    private String fullName;
38    private String name;
39    private String namespace;
40    private String value;
41    private String type;
42
43    XMLAttribute(String fullName,
44                 String name,
45                 String namespace,
46                 String value,
47                 String type)
48    {
49       this.fullName = fullName;
50       this.name = name;
51       this.namespace = namespace;
52       this.value = value;
53       this.type = type;
54    }
55    ...
```

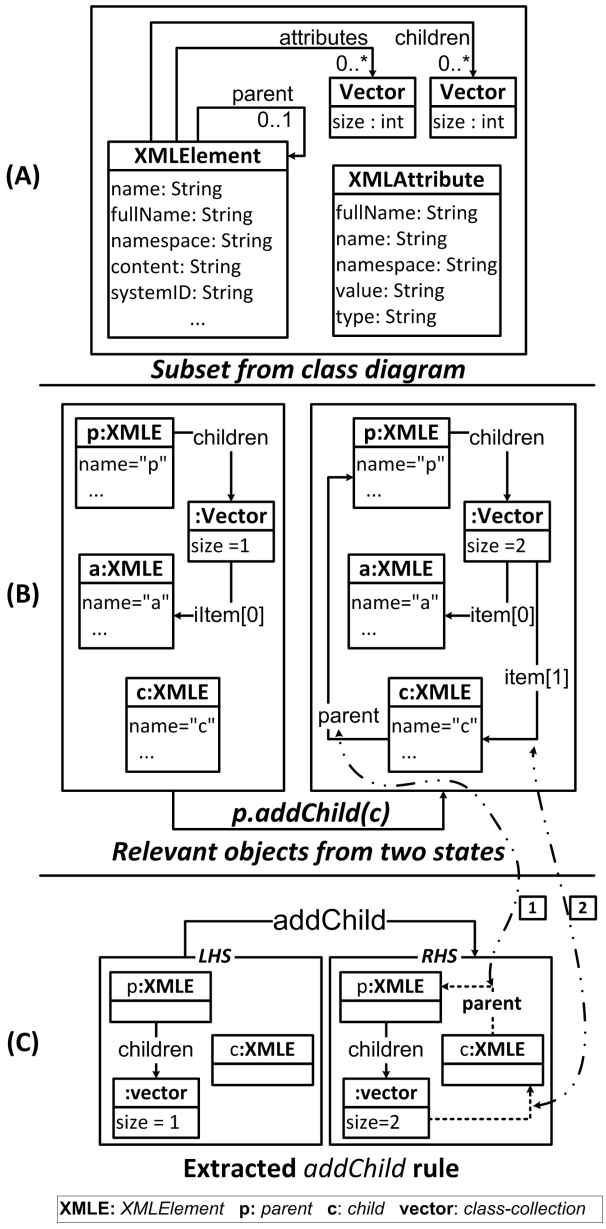


Figure 1: Mapping classes and objects and representing operations

3 Dynamic Extraction of Visual Contracts

After having clarified *what* we want to extract, we turn to the details of the *how*.

3.1 Tracing

Using the AspectJ concepts of *join point* and *point cut* we can control what actions on which objects to react to in order to generate a trace. Join points are actions such as method calls and executions, object instantiations, constructor executions, field references and updates, etc. Point cuts select specific join points based on the objects involved, the method body executed, etc. We are interested in actions that test or change existing objects and fields or create new objects. An *advice* implements the observation mechanism to be executed at each join point. It can be invoked before and/or after the join point, which enables us to observe state changes caused by the execution of the actions.

Observing all actions that involve read or write access to any part of an object, including invocations and executions, we produce a large number of join points. These are filtered by the classes defining the scope of our observation, i.e., our class diagram in Figure 1 (A), so that we only record join points relating to instances of these classes. The class diagram was defined directly from the source code using the ObjectAid tool[2] and then manually reduced to the classes and associations we wanted to observe.

The result is a sequence of nested join points as shown in Listing 1, tracing the `addChild()` operation of the NanoXML case study. Depending on the action performed, join points translate into basic rule instances describing the relevant state transformations. The idea is to aggregate these rule instances into a single rule such as presented in Figure 1 (C).

3.2 Analysis

We analyse traces at runtime, i.e., once the invocation of a method of interest has completed. In order to aggregate the rule instances in the sequence we identify elements across the sequence that refer to the same objects. Then the basic rules are composed along these shared objects to form the overall contract instance.

Figure 2 describes the overall process of analysing a sequence of join points, see lines 4 to 17 in Listing 1: We restrict the relevant join points based on the class diagram, define the scope of the operation as the set of objects potentially affected and match them to the objects that have actually been accessed based on the join points. These are the elements of the rules constructed as a result. We discuss these steps in more detail below.

Scope of Operation. The scope of an execution contains all objects potentially needed for the construction of the rule. It is defined by navigating *this()* and *target()* references from each relevant join point, recording also the values of attributes that may change during execution.

In addition, the scope prevents us from considering unnecessary objects, which are of the right class but unrelated to the trace. For example, Listing 2 shows the query method `enumerate-AttributeNames()` which does not affect any member of its object but writes to a local

[2] http://www.objectaid.com

```
1  // Tracing addChild(..) operation from nanoxml api
2   { XMLElement.addChild(myElement); }

3  // The following is a sequence of nested joinpoint outputs from tracing the above operation
 ...
4      1 before: execution(void net.n3.nanoxml.XMLElement.addChild(IXMLElement))
5          2 before: call(String net.n3.nanoxml.IXMLElement.getName())
6              3 before: execution(String net.n3.nanoxml.XMLElement.getName())
7                  4 before: get(String net.n3.nanoxml.XMLElement.name)
8                  4 after: get(String net.n3.nanoxml.XMLElement.name)
9              3 after: execution(String net.n3.nanoxml.XMLElement.getName())
10         2 after: call(String net.n3.nanoxml.IXMLElement.getName())
11         5 before: set(IXMLElement net.n3.nanoxml.XMLElement.parent)
12         5 after: set(IXMLElement net.n3.nanoxml.XMLElement.parent)
13         6 before: get(Vector net.n3.nanoxml.XMLElement.children)
14         6 after: get(Vector net.n3.nanoxml.XMLElement.children)
15         7 before: call(void java.util.Vector.addElement(Object))
16         7 after: call(void java.util.Vector.addElement(Object))
17     1 after: execution(void net.n3.nanoxml.XMLElement.addChild(IXMLElement))
...
```

Listing 1: Sequence of nested join points obtained by tracing

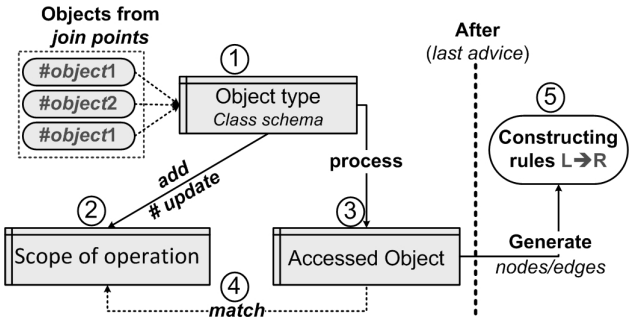


Figure 2: Overview of our trace analysis

`Vector` object created in line 2. The same class exists in the class diagram, but write access to the local `Vector` object is not relevant to the rule to be extracted, i.e., `enumerateChildren()` in the center-bottom of Figure 4.

Accessed Objects. By observing read and write access to objects we are able to identify which elements of our state are required or modified by the operation. Information about objects and attributes created, deleted or modified allows us to create a minimal rule. Read access determines the additional context whose existence is required in order to apply the operation.

There are two explicit join point types for handling read and write access at the field level: the *get-field* and *set-field* join points. In lines 7 and 11 of Listing 1 we show examples of read and write access, respectively. In contrast, access at the object level is implicit in the *method call* join point, where it needs specific restrictions to deal with collection object operations. For example,

```
1  public Enumeration enumerateAttributeNames() {
2     Vector result = new Vector();
3     Enumeration enum = this.attributes.elements();
4
5     while (enum.hasMoreElements()) {
6        XMLAttribute attr = (XMLAttribute) enum.nextElement();
7        result.addElement(attr.getFullName());
8     }
9
10 return result.elements();
11 }
```

Listing 2: enumerateAttributeNames() from NanoXML

adding elements to a collection represents write access, see line 15. Analogously, the calls in the following example require the execution to read the elements of the collection.

```
call(Enumeration java.util.Vector.elements())
call(int java.util.Vector.size())
```

We rely on Java object identifiers to find the accessed objects in the scope.

Rule Construction. Table 1 sketches the cases used for constructing rules, successively building up graphs *LHS* and *RHS* by navigating the trace. For instance in the first row, the *initialization* join point indicates the creation of a new object. Accordingly, we add a new vertex to *RHS*. The second row indicates the creation of an edge, which means that the necessary source and target vertices must be added to both *LHS* and *RHS*. We translate any object attribute *variable-in-class* that points to another object and has a valid type in the class diagram into an edge. The initial value of an *object attribute* will be added to *LHS* and the last value of a relevant write access to *RHS*.

In the last column we state for each element if it is minimal (required for the specification of the effect) or context (shared between pre- and postcondition) based on the access type. Note that there is no join point type for destroying an object in Java, as the garbage collector automatically destroys objects that are not reachable by any reference.

	join point	**step of execution**	**constructing rule**	
			add to	type
1	Initialization *(constructor signature)*	createNode(*id:type*)	RHS	minimal
2	Set *(Field write access)* as object-field	createEdge(*id:type*)	LHS & RHS	minimal
3	call *(collection.add(object))*			
4	get *(Field read access)*	readNode(*id:type*)	LHS & RHS	context
5	Set *(Field write access)* as data-field	updateNode(*id:type*)	LHS & RHS	minimal
6	call *(collection.set(object))*			
7	Set *(Field write access)* as null object-field	deleteEdge(*id:type*)	LHS & RHS	minimal
8	call *(collection.remove(object))*			

Table 1: Cases for rule construction

4 Evaluation

This section presents in detail two experiments on the NanoXML case study to evaluate our approach. The first one is to validate the correctness of our solution of extracting rules and to estimate its performance across a range of different operations, including the benefits of constructing the scope of an operation. The second experiment explores scalability for larger XML documents for one simple operation.

We traced the main operations of the NanoXML API using AspectJ post-compile (binary) weaving. We executed test cases and for each invocation of an operation to be extracted translated the resulting traces into rules. Table 2 lists execution times with and without extraction, as well as a breakdown of the time taken to extract rules into tracing and analysis. The last column of Table 2 shows the ratio of instructions covered for each tested operation as measured by the coverage tool[3]. The coverage gives an indication of the degree to which the constructed rules describe an operation's behaviour. If there are several different execution paths in an operation's implementation, its behaviour can only be described by several rules with different pre- and postconditions.

The operations listed in Table 2 cover the range of different elementary behaviours as presented in Table 1, such as creating and deleting edges, creating nodes, updating attributes. We show some of the extracted rules in Figure 4. All extracted rules are valid according to the class diagram in Figure 1 and, more significantly, they describe the functionality as documented in the NanoXML API documentation[4]. For example, the `addChild()` operation adds an XML element as a child to an existing one. The rule for this operation shown in the top left of Figure 4 has a pre-condition including the prospective child and parent elements and the collection used by the parent to store its children. The postcondition described the creation of the links caused by adding the new child to the collection and linking back to its parent, as well as updating the size of the collection.

The benefit of constructing the scope of an operation becomes apparent by comparing columns 4, 5 and 7 in Table 2. Compared to the total number of objects involved in the operation, the scope reduces significantly the number of objects to be considered in the second step of the analysis. For instance in the first row scoping discards 39 irrelevant objects leaving only 3 to be considered further.

If we compare the effort with and without rule extraction, we find that the overhead ranges between a factor of about 21 for `getParent()` to about 348 for `createElement()`. The main factors are the total number of objects and the number of objects in scope, which are a measure of the complexity of the operation. Despite the overhead, the extraction is manageable for all operations we encountered in this case study.

We conducted the second experiment to validate scalability to different XML documents, especially in terms of the number of children an element contains. The results are summarised in Table 3, where execution times are averages over 10 executions. Operations most affected by larger numbers of children are those that potentially have to read many of them. The most interesting case is `removeChild()` where the number of elements read depends on the position in

[3] http://www.eclemma.org
[4] http://nanoxml.sourceforge.net/orig/NanoXML-2-JavaDoc/index.html

the vector of the child to be removed.

For example the rules in the top right of Figure 4 specifies that the link to the second child element at index 1 is to be removed, while the link to the one with index 0 is part of the context, i.e., occurs in both left- and right-hand side. The link to the element 0 is required because, in order to remove an element from the vector we have to find it by searching from the first index. In case the required element is not present, the rule returned would be without effect, just specifying read access to all the elements in the vector.

To explore how the effort for extracting the rule depends on the position of the element, we compare the case where the child element is at index 3 against cases where the element is at the end of vectors of 100 - 400 elements, as shown in Table 3. Figure 3 visualises the performance, plotting execution time vs. size. It shows that for elements removed at index 3 the time taken is independent of the size of the vector while for elements at the end of the list it is linear in relation to the length of the list.

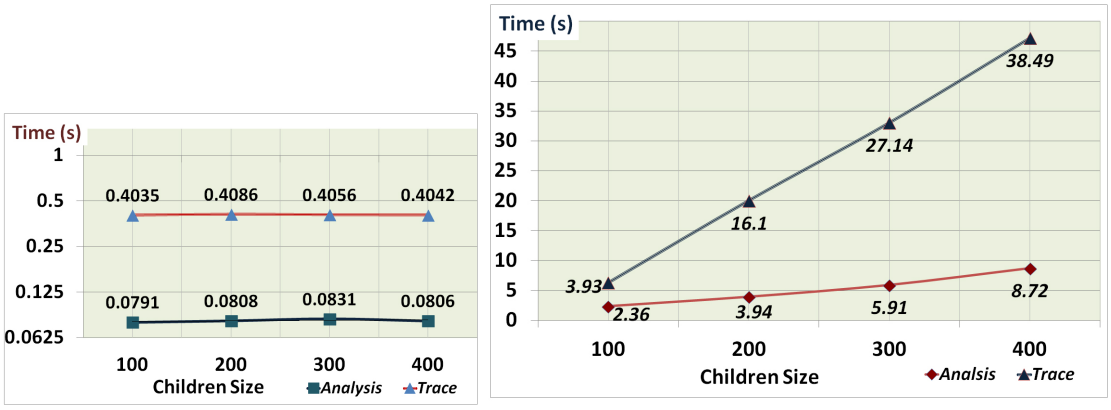
In conclusion, there is a significant overhead associated with the extraction of rules which cause problem if the approach should be used for continuous monitoring of applications, but can be neglected for applications in reverse engineering for program understanding.

no	**operation**	**exec. time without extraction**	**tracing**			**analysis**		**covered instruct ions**
			all objects	scope	exec. time	objects in rule	exec. time	
1	`createElement()`	2.94e-5	42	3	0.346	3	0.044	45/62
2	`addChild(XMLElement)`	1.42e-5	32	6	0.055	3	0.027	14/44
3	`insertChild(IXMLEl- ement, int index))`	1.51e-5	34	6	0.046	3	0.020	15/45
4	`removeChild(XMLEle- ment child)`	4.59e-5	166	12	0.153	7	0.051	8/13
5	`removeChildAtIndex (int index)`	1.20e-5	18	3	0.026	2	0.017	5/5
6	`setName(String name)`	1.20e-5	15	3	0.081	1	0.009	10/10
7	`getParent()`	0.93e-5	11	3	0.020	1	0.006	3/3
8	`enumerateChildren()`	0.93e-5	16	3	0.024	2	0.011	4/4

Table 2: Extracting rules from individual executions of NanoXML operations (times in *sec*.)

no	**children**	**child found at index**	**tracing**			**analysis**	
			all objects	scope	exec. time	objects in rule	exec. time
1	100	3	166	12	0.403	7	0.079
2		100 *(last index)*	3791	375	3.931	104	2.364
3	200	3	166	12	0.408	7	0.080
4		200 *(last index)*	5879	603	16.096	204	3.941
5	300	3	166	12	0.405	7	0.083
6		300 *(last index)*	8779	903	27.138	304	5.906
7	400	3	166	12	0.404	7	0.080
8		400 *(last index)*	11679	1203	38.492	404	8.716

Table 3: Results for extracting rules for the `removeChild()` operation (times in *sec*.)



(a) child-element exists at the third index

(b) child-element exists at the last index

Figure 3: Overheads for extracting rules for `removeChild(child)`

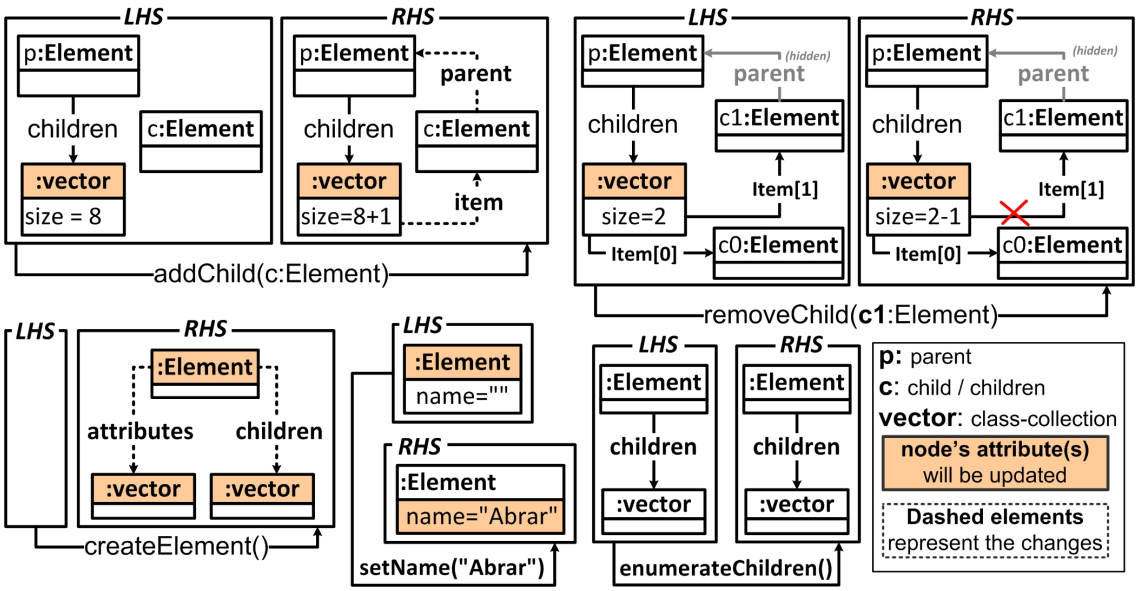


Figure 4: Example of extracting 5 instantiated visual contracts *(rules)*

Discussion and Threats to Validity. The evaluation is potentially biased by the selection of the case study and the choice of operations and test cases within it. The NanoXML case study is one of a selection of benchmarks in the Software-artifact Infrastructure Repository[5] frequently used for evaluating automated testing and program analysis in Java.

At 7646 LOC, 24 classes it is not a large API, but what we have seen from the evaluation suggests that it is the complexity of the individual operations, in terms of the number of objects involved, that determines the effort. This in turn depends on the number of classes we choose to

[5] http://sir.unl.edu

observe. The case study was selected because it offers a range of operations of significant structural complexity, creating objects and creating and destroying links, sometimes with complex structural preconditions. In that sense it is the kind of example for which an approach like ours may be useful. The choice of operations to be traced and the test cases used for it is motivated by trying to explore all aspects of this complexity, and our test cases do indeed cover all the cases of Table 1.

5 Related Work

Automated reverse engineering is typically based on static or dynamic analysis. The static approach, exemplified by [SCM13, RVR05, TP03], examines the source code only, with the intention of extracting all possible behaviours. It is useful with incomplete systems, e.g., components that cannot be executed independently [RVR05]. However, it is limited in its ability to detect dynamic object-oriented behaviours such as dynamic binding or method calls. For example, [TP03] propose a static approach for generating sequence and collaboration diagrams from C++ code, thereby potentially over-approximating the actual behaviour. Our solution is dynamic, based on running the system, as are others [BMT+12, ZDHZ11, ZKZ10]. The potential drawback here is that the extracted model represents only those behaviours that are actually executed.

Many dynamic reverse engineering approaches take advantage of aspect oriented concepts for instrumentation at different levels. We use AspectJ to instrument low-level Java bytecode and generate visual contract instances at runtime. A similar strategy lies behind [BMT+12] generating object graphs or [ZDHZ11] extracting sequence diagrams.

Using Java bytecode instrumentation for extracting a context-free graph grammar is [ZKZ10]. Their use of graph grammars is for representing nested hierarchical call graphs, not to model the behaviour of the system in terms of transformations on objects. Their tracing focusses on method calls only. A mining approach infers the graph grammar from the set of extracted call graphs.

At the level of models [ALCN10] discovers traceability relationships [GG07], e.g., the relation between elements of source and target models in a model transformation. The approach is based on aspect-oriented programming, tracing access to objects by creation, update and deletion actions based on transformation events and aggregating them into a relational model. While the observed actions are similar, our approach works at the level of Java programs.

6 Conclusion

We propose a dynamic approach to reverse engineering of instantiated versions of visual contracts from Java code. We use instrumentation based on AspectJ to observe executions of existing test cases, record these observations in traces and analyse them to filter out irrelevant objects and aggregate their basic steps into rules covering the overall precondition and effect for that execution.

We have validated the correctness and scalability of our approach by applying it on the NanoXML case study. The results are consistent with the class diagram and the expected behaviour of the operations based on the documentation. We observe an overhead factor of up to 348 compared to the execution of the same tests without rule extraction, increasing with the number of objects

accessed, but manageable for all operations we encountered in the case study.

One limitation of the approach, due to the semantics of Java, is the inability to detect the deletion of objects. This is handled implicitly by Java's garbage collector. As long as the main application of our technique is in program understanding, this aspect of Java's semantics is reflected correctly in the extracted contracts, Where a more high-level, language-independent model is sought, this limitation may have to be addressed.

Each extracted rule (contract instance) describes one possible behaviour of the operation, covering only the part executed by one test case. Therefore, we usually obtain more than one rule for a specific operation, which then need to be combined or generalised. As future work, we plan to apply our approach to rule learning to this problem [AHK13]. We also intend to infer advanced features such as NACs and multi objects, which will increase the accuracy of the specification. The present the solution may already be useful for program understanding, e.g., as part of testing or debugging, but does not deliver a general specification of the operation's behaviour.

Bibliography

[AHK13] A. M. Alshanqiti, R. Heckel, T. Khan. Learning Minimal and Maximal Rules from Observations of Graph Transformations. *Electronic Communications of the EASST* 58, 2013.

[ALCN10] B. Amar, H. Leblanc, B. Coulette, C. Nebut. Using Aspect-Oriented Programming to Trace Imperative Transformations. In *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International*. Pp. 143–152. 2010.

[BLL06] L. Briand, Y. Labiche, J. Leduc. Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software. *IEEE Transactions on Software Engineering* 32(9):642–663, 2006.

[BMT+12] H. Brito, H. Marques-Neto, R. Terra, H. Rocha, M. Valente. On-the-fly extraction of hierarchical object graphs. *Journal of the Brazilian Computer Society*, pp. 1–13, 2012.

[GG07] I. Galvao, A. Goknil. Survey of traceability approaches in model-driven engineering. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. Pp. 313–313. 2007.

[KRH12] T. A. Khan, O. Runge, R. Heckel. Testing against Visual Contracts: Model-Based Coverage. In *ICGT*. Pp. 279–293. 2012.

[RKH13] O. Runge, T. A. Khan, R. Heckel. Test Case Generation Using Visual Contracts. *ECEASST* 58, 2013.

[RVR05] A. Rountev, O. Volgin, M. Reddoch. Static Control-flow Analysis for Reverse Engineering of UML Sequence Diagrams. *SIGSOFT Softw. Eng. Notes* 31(1):96–102, Sept. 2005.

[SCM13] M. K. Sarkar, T. Chatterjee, D. Mukherjee. Reverse Engineering: An Analysis of Static Behaviors of Object Oriented Programs by Extracting UML Class Diagram. *International Journal of Advanced Computer Research* 3(3), 2013.

[TP03] P. Tonella, A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. Pp. 159–168. 2003.

[ZDHZ11] T. Ziadi, M. A. A. Da Silva, L. M. Hillah, M. Ziane. A fully dynamic approach to the reverse engineering of UML sequence diagrams. In *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. Pp. 107–116. 2011.

[ZKZ10] C. Zhao, J. Kong, K. Zhang. Program Behavior Discovery and Verification: A Graph Grammar Approach. *IEEE Transactions on Software Engineering* 36(3):431–448, 2010.