Extracting Visual Contracts from Java Programs

Abdullah Alshanqiti*
amma2@mcs.le.ac.uk

Reiko Heckel
*Department of Computer Sciences
Leicester University, UK
reiko@mcs.le.ac.uk

***Abstract*—Visual contracts model the operations of components or services by pre- and post-conditions formalised as graph transformation rules. They provide a precise intuitive notation to support testing, understanding and analysis of software. However, due to their detailed specification of data states and transformations, modelling real applications is an error-prone process. In this paper we propose a dynamic approach to reverse engineering visual contracts from Java based on tracing the execution of Java operations. The resulting contracts give an accurate description of the observed object transformations, their effects and preconditions in terms of object structures, parameter and attribute values, and their generalised specification by universally quantified (multi) objects. While this paper focusses on the fundamental technique rather than a particular application, we explore potential uses in our evaluation, including in program understanding, review of test reports and debugging.**

I. Introduction

Visual Contracts (VCs) provide a precise high-level specification of the object graph transformations caused by invocations of operations on a component or service. They link static models (e.g., class diagrams describing object structures) and behavioural models (e.g., state machines specifying the order operations are invoked in) by capturing the preconditions and effects of operations on a system's objects.

Visual contracts differ from contracts embedded with code, such as JML in Java or Contracts in Eiffel, as well as from model-level contracts in OCL. They are *visual*, using UML notation to model complex patterns and transformations intuitively and concisely, and their *executable semantics* based on graph transformation supports model-based oracle and test case generation [1], [2], run-time monitoring [3], service specification and matching [4], state space analysis and verification.

However, creating a detailed model in any language is error-prone. Visual contracts are no exception, and their specification of object states and transformations requires a deeper understanding of a system than models of externally visible behaviour. This limits their applicability in testing, verification and program understanding in general.

In this paper we propose a dynamic approach to reverse engineering visual contracts from sequential Java programs based on tracing the execution of Java operations. The resulting contracts give accurate descriptions of the observed object transformations, their effects and preconditions in terms of object structures, parameter and attribute values, and allow generalisation by *multi objects*. The restriction to sequential Java is due to the need to associate each access to a unique operation invocation.

Given a Java application, the process starts by selecting the classes and operations within the scope of extraction and providing a set of test cases for the relevant operations. We proceed by (A) observing the behaviour under these tests using AspectJ instrumentation and synthesising contract instances as pre/post graphs of individual invocations; (B) combining the instances into higher-level rules by abstracting from non-essential context; (C) generalising further by introducing multi objects; and (D) deriving logical constraints over attribute and parameter values.

First solutions to variants of (A) and (B) were reported in [5], [6], respectively. Apart from general performance improvements in the individual algorithms and their integration in a prototype tool, the dynamic analysis (A) was extended by traceability of contract instances to code, recording access and changes to attribute and parameter values, and producing contract instances in a format that could be fed into the initial step (B) of the learning. The latter originally relied on both positive and negative examples, so had to be adapted to make do with positive examples only as produced by (A). Support for operations with parameters was also added. Steps (C) and (D) extending the learning of basic contracts by multi objects and attribute constraints are discussed here for the first time, as is an experiment on the usefulness of visual contracts for testing and debugging.

Following a general presentation of the notions and techniques of the approach in Sect. II, Sect. III-A describes the prototype tool implementing them. The evaluation in Sect. III discusses the scalability of the extraction as well as the validity of the resulting models and their utility in program understanding in the context of testing and debugging. Apart from their use in validation, case studies and experiments are chosen to exemplify potential applications in this area without claiming that the present tool could support real-world use. After discussing Related Work, Sect. V concludes the paper.

II. Extracting Visual Contracts

This section gives an overview of the approach using a simple case study of a Car Rental Service designed to represent a range of different preconditions and effects of operations over a complex object-structure, including the creation of objects, the creation and deletion of links as well as attribute updates

```
public interface IRental extends Serializable{
    public String registerClient(String city, String clientName);
    public String makeReservation(String ClientID, String pick−up, String drop−off);
    public void cancelReservation(String Reference);
    public void cancelClientReservation(String clientID);
    public void pickupCar(String Reference);
    public void dropoffCar(String Reference);
    public Reservation[] showClientReservations(String clientID);
    public Client[] showClients (String city);
    public Car[] showCars (String city);
}
```

Listing 1: Interface of a Car Rental Service

and constraints. Basic concepts of graph transformation are introduced, following [7].

An interface with the relevant operations is given in Listing 1. The class diagram in the top left of Figure 1 shows the selected classes, whose instances will be observed. Classes and data-valued attributes in the diagram map to classes and attributes in Java. Associations with cardinality 1 at the target represent object-valued attributes in their source class and associations with cardinality $*$ are implemented by containers.

Formally, a class diagram is represented as an *attributed type graph* TG, i.e., a distinguished graph defining vertex, edge, attribute and data types from which object graphs can be constructed. An *object graph* over TG is a graph G equipped with a structure-preserving mapping $G \to TG$ assigning every element in G its type in TG.

A. Observing Access and Synthesising Contract Instances

We adopt a dynamic approach to extract, for each operation invocation, a *contract instance* capturing the observed behaviour. Observations are made by weaving instrumentation code using AspectJ. This results in a trace recording the object creation, read and write access to objects and attributes caused by the active invocation. Concurrent invocations create the problem of identifying for each observation the relevant invocation and are therefore not considered. We aggregate observations into a contract instance capturing the overall precondition and effect of the invocation (see [6] for more details). Along with the instance we collect traceability data for its elements, such as the line numbers in the code causing for their access. This is used later to validate the extraction, e.g., to assess which code fragments are captured by which contracts.

Consider the contract instances in Figure 1. Instance *registerClient* creates a new client object, registers it with the branch at *city*, and updates attribute *branch.cMax*. Instance *makeReservation* books a car for a client by creating a new reservation object *r* with links *pickup, dropoff, made* and *for*. Links *of* and *at* indicate that a client can reserve cars from the *pickup* branch they are registered with. Instances *pickupCar* and *dropoffCar* describe the movement of a car from the *pickup* to the *dropoff* branch.

As can be seen in the example, a contract instance consists of a pair of object graphs representing the situation before and after the operation. We write $b = op(a_1, \ldots, a_n) : G \Rightarrow H$ to indicate the invocation $op(a_1, \ldots, a_n)$ of an operation with signature $op(x_1 : T_1 \ldots, x_n : T_n) : T$ leading to a transformation of G into H. We assume that G, H live in a common name space given by unique object identities, so the elements deleted, preserved and created by the transformation are $G \setminus H, G \cap H$ and $H \setminus G$, respectively.

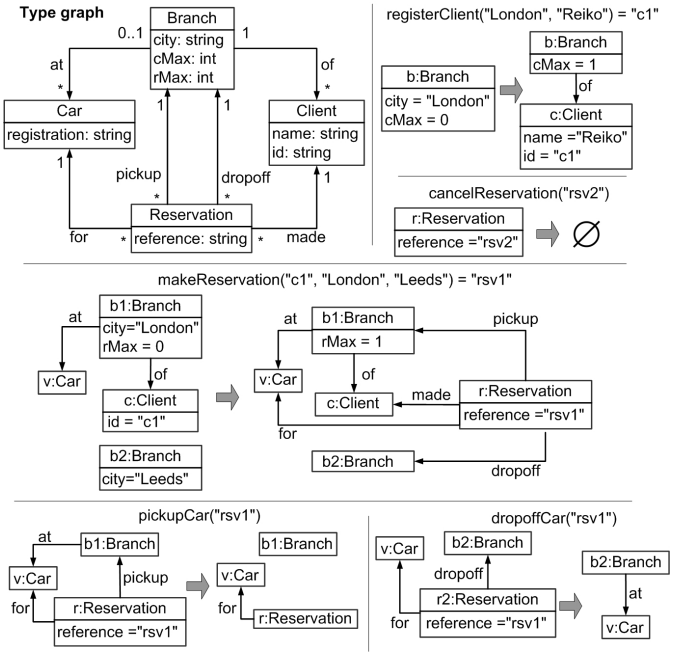


Fig. 1: Type graph and rule instances, extracted from car rental service

B. Deriving Minimal Contracts and Shared Context

Each contract instance only represents one invocation, but of course our aim is to derive a small set of contracts that describe the overall behaviour as precisely as possible. Such a general contract is given by a set of *parametrised rules* $op(x_1, \ldots, x_n) = y : L \Rightarrow R$ over the same operation signature with graphs L and R, called the *left-* and *right-hand side* of the rule, expressing the pre- and postconditions of the operation. As before $L \setminus R$, $L \cap R$ and $R \setminus L$ represent the elements deleted, preserved and created by the rule.

To derive such a general model we consider all instances representing executions of the same operation. First, we generate a *minimal rule* for each instance, i.e., the smallest rule containing all objects referred to by the operation's parameters and able to perform the observed object transformation. The construction has been formalised in [8] and implemented (without considering parameters) in [5]. Formally, given a contract instance $b = op(a_1, \ldots, a_n) : G \Rightarrow H$ its minimal rule is the smallest rule $L \Rightarrow R$ such that $L \subseteq G, R \subseteq H$ with $a_1, \ldots, a_n \in L$ and $b \in R$ as well as $G \setminus H = L \setminus R$ and $H \setminus G = R \setminus L$. That means, the rule is obtained from the instance by cutting all context not needed to achieve the observed changes nor required as input or return.

The result is a classification of instances by effect: All instances with the same minimal rule have the same effect, but

possibly different preconditions. These are in turn generalised by one so called *maximal rule* which extends the minimal rule by all the context that is present in all instances, essentially the intersection of all its instances' preconditions. Figure 2 shows an example of this generalisation where maximal rule (C) results from instances (A) and (B) of *cancelClientReservation(..)*. The shared effect in both cases is the deletion of the Reservation object connected to the Client and the minimal rule is identical to (B). The isolated *r1:Reservation* object in (A) arises from an unsuccessful test on $r1$ when searching for the reservation object to be cancelled.

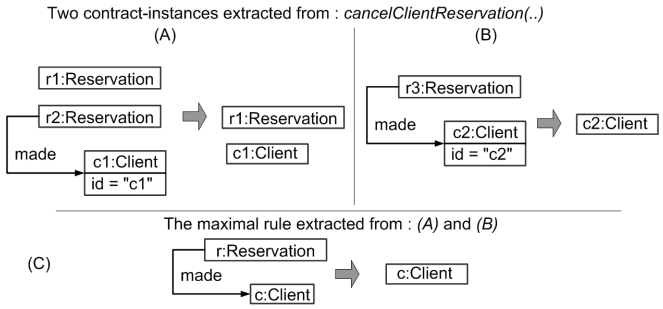


Fig. 2: Extracting maximal rules from contract instances

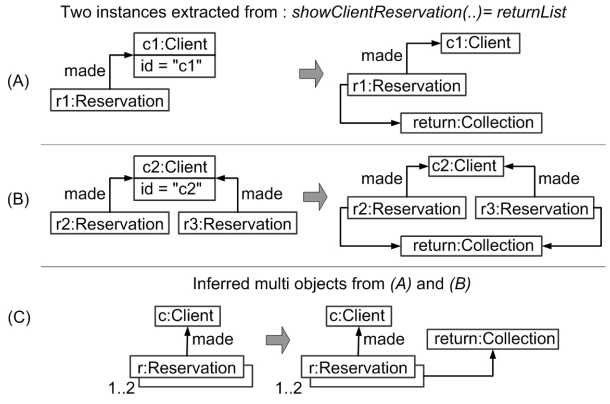


Fig. 3: Inferring MOs from contract instances

But minimal or maximal rules are not just generalisations of instances, but provide a constructive specification. Given an object graph G, a rule can be applied if there is a match $m : L \to G$, such that L is (isomorphic to) a subgraph of G and removing (an image of) $L \setminus R$ from G, the resulting structure is a graph. The derived object graph H is obtained by adding a copy of $R \setminus L$. Unsurprisingly, applying a rule extracted from a contract instance $b = op(a_1, \ldots, a_n) : G \Rightarrow H$ to the pre-graph G of that instance, we obtain its post-graph H, but we can also apply the same rule to other given graphs deriving transformations not previously observed.

C. Introducing Universally Quantified Multi Objects

The contracts extracted so far may use a number of rules to describe the same operation. In the case of iteration over containers, for example, the set of minimal rules is potentially unbounded, but some only differ in the number of objects manipulated while performing the same actions on all of them. Rules with multi objects (MOs) provide a concise way to specify constraints and actions across sets of objects of different cardinalities.

A *multi-object (MO) rule* distinguishes a set $M \subseteq L$ of MO nodes, with cardinality constraints $card : M \to \mathcal{P}(\mathbb{N})$ stating how many concrete objects each MO can be instantiated by. Application of MO rules is defined by expanding MO nodes into sets of regular nodes. An expansion of an MO rule is a (regular) rule obtained by successively replacing each MO node $m \in M, card(m) = C$ by $c(m)$ copies for some $c(m) \in C$. This includes copying all incoming and outgoing edges so that for each node $m \in M$ and chosen $c(m)$ we get L^m as

- $L^m_V = L_V \setminus \{m\} \uplus \{m\} \times \{1, \ldots, c(m)\}$ and
- $L^m_E = L_E \setminus L_E(m) \uplus L_E(m) \times \{1, \ldots, c(m)\}$

where $L_E(m) = \{e \mid src^L(e) = m \vee tar^L(e) = m\}$ is the set of edges attached to node m. Sources, targets and types of new edges and nodes are inherited from L. The expansion extends to R on the MO nodes shared with L. (Due to the associativity of the product $\times$ up to isomorphism, the resulting rule is essentially independent of the order of the MO nodes expanded.) Note that for two MO nodes m_1, m_2 connected by an edge we will create $c(m_1) * c(m_2)$ edges between the copies of m_1 and m_2. An *application of an MO rule* to an object graph G is an application of a maximal applicable expansion.

For example, node *Reservation* in Figure 3 (C) is an MO node (shown with a 3D shadow) with cardinality 1..2, applicable to object graphs with 1 or 2 *Reservation* nodes connected to the *Client*. Contract instances of two corresponding transformations are shown in Figure 3 (A) and (B).

To derive MO rules from such instances we have to discover sets of nodes that have the same structure and behaviour, then represent them by a single multi-object node. We only consider multi-object nodes that are part of the minimal rule because their typical use is to describe universally quantified effects (rather than preconditions). In the rule instance Figure 3 (B), for example, both *Reservation* nodes have the same context, i.e., they both point to the same *Client* node by a *made* edge, and they are both connected to *return: Collection* on the right-hand side, so share the same behaviour. Therefore they are substituted by one multi-object, as shown in Figure 3 (C), which also generalises Figure 3 (A) with only one occurrence. After inferring multi objects within individual rules, if two MO rules are isomorphic, the two original rules can be replaced by a single MO rule with appropriate cardinalities reflecting the generalised cases.

Two objects are *equivalent* if they are (1) of the same type; (2) part of the minimal rule; and (3) have the same context (incident edges of the same type connected to the same nodes) in the pre- and postcondition (thus specify the same actions). Assuming for every operation op a set of maximal rules $R(op)$ as constructed in Sect. II-B, we derive MO rules in two steps. *Merge equivalent objects:* For each rule $m \in R(op)$ and each non-trivial equivalence class of objects in m, one object is chosen as the representative for that class and added to the set

of MO nodes for m, while all other objects of that class are deleted with their incident edges. The cardinality of the MO node is defined to be the cardinality of its equivalence class (the number of objects it represents). The resulting set of MO rules is $MOR(op)$.

Combine isomorphic rules: A maximal set of structurally equivalent rules in $MOR(op)$, differing only in their object identities and cardinalities of their MO nodes, forms an isomorphism class. For each such class we derive a single rule by selecting a representative MO rule and assigning to each of its MO nodes the union of cardinalities of corresponding nodes in all the rules in the class. The resulting set of combined MO rules is $CMOR(op)$. An example is the derivation of Figure 3 (C), a combination of basic rule Figure 3 (A) with the MO rule derived from (B) whose cardinalities of 1 and 2 for the *Reservation* node are merged to 1..2.

D. Deriving Constraints on Attributes and Parameter Values

So far we have focussed on structural preconditions and effects, disregarding the data held in objects' attributes or passed as parameters. However, at implementation level, manipulation of object structure and data are tightly integrated. While we have seen that the structural view is naturally expressed by graphical patters, constraints or assignments over basic data types are more adequately expressed in terms of logic.

The contract for *cancelClientReservation(cid: String)* describes the removal of a *Reservation* object linked to the *Client* whose id matches the parameter cid. In the contract this is expressed by the equality $id = cid$ in the *Client* object. Formally, $c.id$ and cid, as well as the right-hand side counterpart $c.id'$ of $c.id$, are local variables of the contract that get instantiated by the match as part of an application. In particular, given a graph object G and match $m : L \to G$, $c.id$ is instantiated by the value of the id attribute of $m(c)$, i.e., $m(c.id) = m(c).id$. In a similar way we can extend m to evaluate complex expressions and use these in assignments to update attributes. The formalisation in attributed graph transformation assumes an abstract data type A as attribute domain linking it to the structural part by attribution maps.

Let us consider how attribute constraints for contracts can be learned. Say, an instance $i = [b = op(a_1, \ldots, a_n) : G_i \Rightarrow H_i]$ has attribute and parameter values A_i (i.e., these values were either read or written during the corresponding invocation). A maximal rule $r = [op(x_1, \ldots, x_n) = y : L \Rightarrow R]$ generalising a number of instances with shared effects is given a set X of local variables for all formal parameters $x_1, \ldots, x_n$ and all attributes read or accessed by all its instances. Since maximal rule r is embedded by a match m_i into every instance i it subsumes, this extends to an assignment of the local variables $m_i : X \to G_i$.

Fixing an order on the variables X, each m_i becomes a vector of values to be fed into a machine learning tool capable of driving logical constraints. We use the Daikon tool [9] designed for the derivation of invariants over program variables. From the assignments m_i for all instances i that contributed to the construction of rule r Daikon generates a set of constraints that are valid for all assignments. These constraints are fed back into the graphical part of the contract, where each becomes part of the pre- or postcondition depending on wether the variables used occur only in L or in L, R and the parameters. This approach allows the separation of structural and constraint learning.

III. Evaluation

In this section, we illustrate the implementation of the approach by a proof-of-concept prototype and discuss correctness and completeness of extracted contracts. We report on experiments to assess the utility of visual contracts and the scalability of the extraction as implemented by the prototype.

A. Prototype Tool

The approach is implemented by a tool whose high-level architecture is shown in Figure 5. It consists of a Tracer observing the behaviour of selected classes using AspectJ and constructing contract instances (cf. subsection II-A), a Generaliser learning minimal, maximal and MO rules (cf. subsection II-B and II-C) using Daikon [9] to learn constraints (cf. subsection II-D) both supported by a database connection and a Visualiser for selective display and analysis of contracts. An export to the graph and model transformation tool Henshin [10] is used to simulate contracts for validation. First we focus on the Visualiser to illustrate how results are presented and how they could be used to aid program understanding.

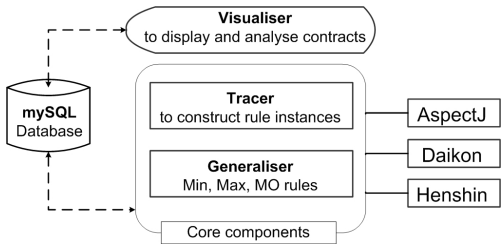
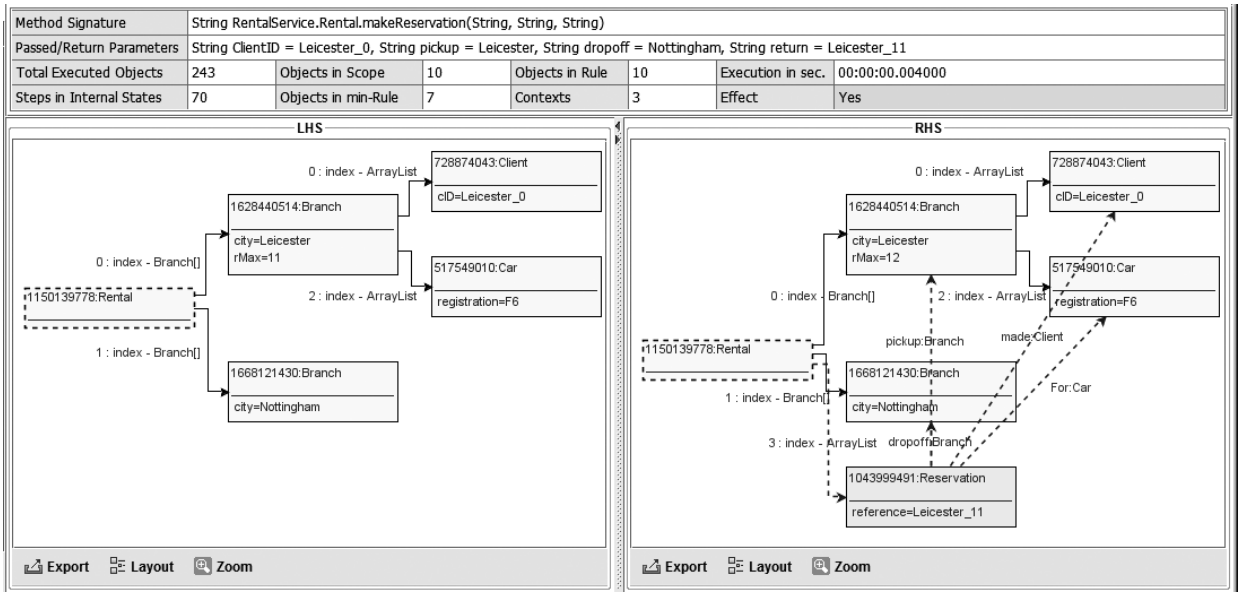


Fig. 5: Architecture of the Tool

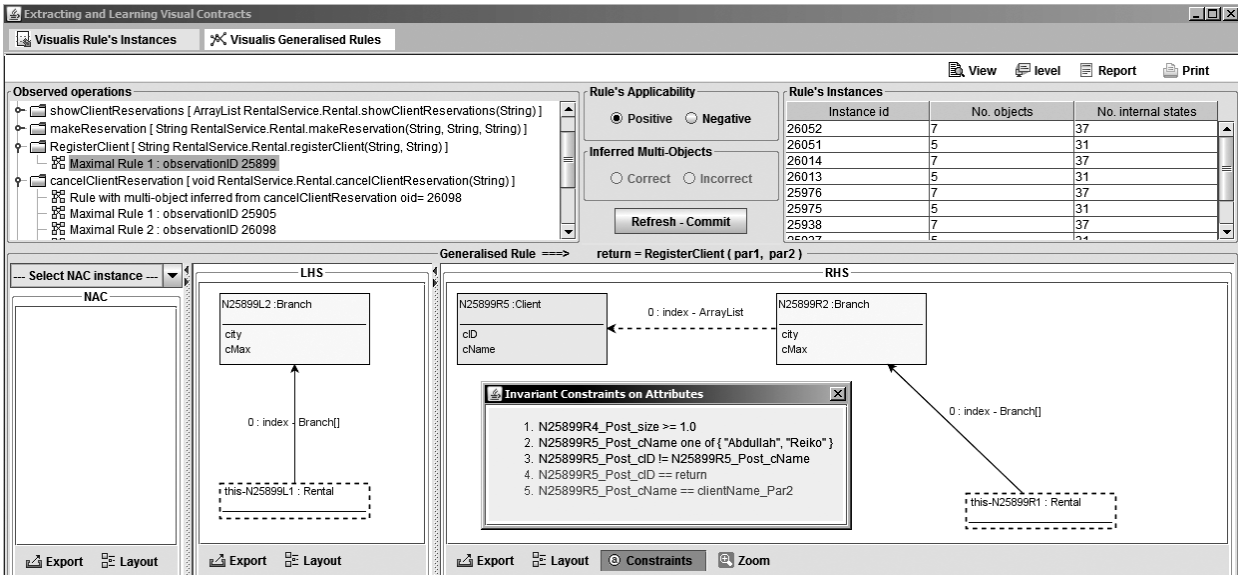
The main task of the Visualiser (see Figure 4) is to organise, browse and display extracted contracts. To this end we support

- the distinction in colour and style between elements of the minimal and larger maximal rule, e.g., dotted edges and nodes with grey background represent elements of minimal rules, while nodes with white background and solid edges are context elements;
- the alternative display of collections as to-* associations or using explicit collection objects;
- the selective visualisation of rules, for example of the minimal rule or the precondition only.

Figure 4 shows two screenshots of the main interface. In (a), we present an instance extracted from *makeReservation()* in Figure 1. The upper part of (a) gives information on the operation signature, actual parameters and the extraction process. Apart from the rule showing the precondition and effect at a high level, we provide information on the access to individual objects with the corresponding locations in the



(a) Rule instance



(b) Generalised rules interface

Fig. 4: Visualiser interface

code. They are available through a pop-up window like the one in Figure 6 activated by clicking on the *:Reservation* node in the right-hand side of the contract.

Figure 4 (b) shows how generalised (maximal and MO) rules are displayed. The top left shows a list of the rules organised by their operation signatures. When selecting, e.g., a maximal rule, all its rule instances will appear in the table, see the top right of (b). The lower part shows the maximal rule with multi objects and 5 attribute constraints for *RegisterClient()* that describe the relation between attribute values, input and return parameters. For example, the 4th constraint states that the value of the *cID* is returned while the 5th requires that the *cName* attribute of the new Client

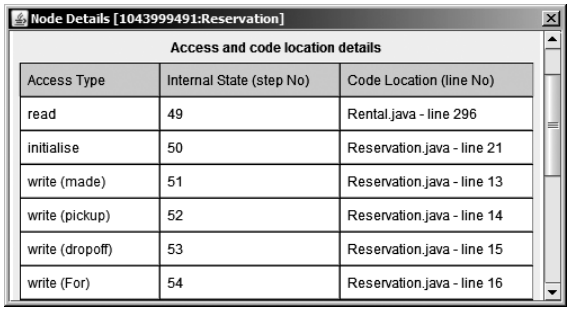
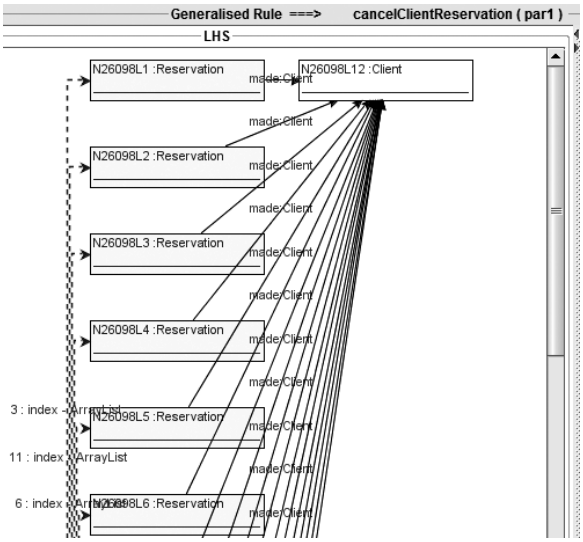
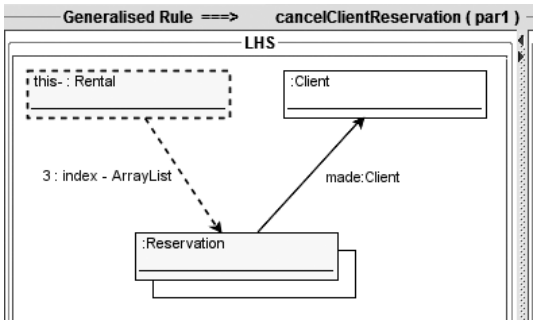


Fig. 6: Object access and code locations



(a) Left-hand side of maximal rule *cancelClientReservations()*



(b) Left-hand side with multi object extracted from (a)

Fig. 7: Extraction of rule with multi object

object has the same value as the 2nd parameter. An example of a rule with multi object is shown in Figure 7 (b) as extracted from the maximal rule in (a) for *cancelClientReservations()*.

B. Correctness and Completeness

In order to establish to which extent the contracts extracted provide an accurate description of the software's behaviour we consider two directions, the *correctness* and *completeness* of the contracts. For every state s in the implementation there exists a corresponding object graph $G(s)$ at model level obtained by representing all objects in the scope of observation (i.e., that are instance of the classes selected for tracing, cf. start of section II) as nodes, object-valued attributes as edges and data-valued attributes as node attributes. Then, a model is *correct* if for every valid state s and invocation in, a step $in : G(s) \Rightarrow H$ in the model implies a step in the implementation from state s to a new state s' such that $H = G(s')$. That means, the model does not allow behaviour that is not implemented by the system. Conversely, *completeness* means that for each valid state s, a step caused by an invocation in of the implementation leading to a state s' must be matched by a step $in : G(s) \Rightarrow G(s')$ in the model, i.e., all the system's behaviour is captured by the model.

In general, the models extracted will be neither correct nor complete. Correctness fails because the model is extracted for a certain part of the system only as identified by the implementation classes selected for tracing. Anything outside this scope of observation is not recorded and therefore not represented by the model. That means, if the implementation checks a condition on the state of an object outside scope, this check is not reflected in the precondition of the contract. If this check fails, a step in the model may not be reflected by a step in the implementation. A weaker condition taking into account this limitation is that of *moderated correctness*. It states that, if both preconditions are satisfied, the observable effect of the implementation-level step should match the effect of the model-level step. Here the comparison is moderated via the the mapping $G(\_)$ of implementation states to object graphs, which also takes account of the scope.

Completeness fails for the same reason that test cases cannot prove the correctness of a system. The dynamic approach to extracting contracts is inherently dependent on the range of behaviours observed, and behaviours that have not been observed will not be reflected in the model. So what can we realistically hope to achieve? A minimal notion of completeness should require that all observed behaviours are represented in the model, i.e., when executing the tests the model was extracted from, all steps steps in the implementation should be matched by the model.

We used manual inspection on the Car Rental Service case study to validate if the models extracted by the tool satisfy the baseline/moderated notions of correctness and completeness. The limited amount of code and our familiarity with the application allowed us to perform a detailed review for every method in the interface, validating for all execution paths that there exists a rule in the corresponding contract capturing the path's combined precondition and effect, and vice versa for every rule that the behaviour described is fully implemented. This process was aided by the export of extracted contracts to the Henshin model transformation tool [10], which provides a facility to simulate contracts based on their operational semantics as graph transformation rules.

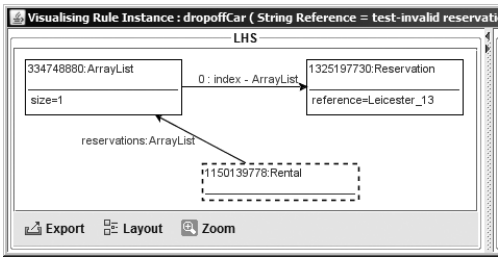
Consider the source code fragment in Listing 2 implementing the *dropoffCar()* method. There are three possible paths leading to at least three different contracts, depending on the evaluation of the two *if* statements in lines 4 and 10. When executing this method by three test cases that cover all statements, the extracted rules reflect the expected behaviours. This is confirmed by tracing the line numbers in the code responsible for the access to objects in the contracts.

Figure 8 shows the left-hand sides of the three rules extracted from *dropoffCar()*. For example, (a) reflects the behaviours of statements 1-6 as we pass an invalid *reservation id* and, accordingly, the execution breaks at line 5. The rule correctly describes the access to *this:Rental* and the *Reservation* container. In (b) the parameter is valid, i.e., the *Reservation* object *Leicester_13* exists, but the execution breaks at line 11 since the car has not been picked up yet. This can be seen from the *pickup* link which would have been

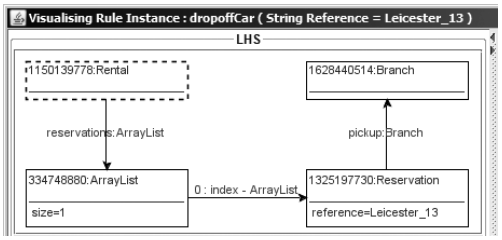
deleted otherwise. The rule in (c) reflects correctly the third path, i.e., the conditions in 4 and 10 are false so there is no return from the method there.

```
1  public void dropoffCar(String Reference){
2
3      int iIndex = getReservationIndex(Reference);
4      if (iIndex==-1){
5          return;
6      }
7
8      Reservation getReservation = this.reservations.get(iIndex);
9      // check if reserved car has been picked up already
10     if (getReservation.pickup!=null){
11         return;
12     }
13
14     // return reserved car to the drop-off branch
15     getReservation.dropoff.at.add(getReservation.for);
16     // remove reservation object
17     this.reservations.remove(iIndex);
18 }
```

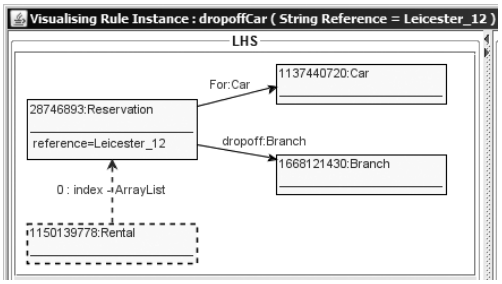
Listing 2: Implementation of dropoffCar() method



(a) Rule instance extracted from lines (1-6)



(b) Rule instance extracted from lines (1-12) without line (5)



(c) Rule instance extracted from all lines except (5,11)

Fig. 8: Rule instances for *dropOffCar()*

More generally, due to the method of model extraction (and assuming it was correctly implemented in our prototype tool) we can assert that model and implementation should show the same behaviour at least for the test cases used. In particular

- contract instances capture precisely the preconditions and effects relevant to the invocation they are derived from, within the scope of observation;
- minimal rules capture exactly the effect of contract instances they are extracted from;
- maximal rules subsume all contract instances they derive from, i.e., every contract instance can be replicated as an application of the maximal rule;
- rules with multi-objects are (more concise, but) equivalent to the sets of maximal rules they derive from, i.e., by retaining the original rules' cardinality information, they describe exactly the same set of transformations;
- the parameter and attribute constraints derived do not invalidate any of the contract instances their maximal rule originates from.

The fact that, in general, models are only representative of the behaviour they were extracted from is an obstacle to some applications, such as their use in verification, where automated extraction has to be followed by a manual review and completion of contracts. In the following section we demonstrate an application to program understanding in the context of testing and debugging that does not rely on completeness or correctness beyond the set of tests executed.

C. Utility in Assessing Test Reports and Localising Faults

Using the Car Rental Service case study we conducted an experiment to evaluate the utility of visual contracts extracted from the execution of test cases for analysing test reports and identifying faults. In this paper-based exercise our hypothesis was that "visual contracts, rather than textual representations of the same information, improve recall and accuracy of detecting faults in test reports". Generally, we wanted to find out how visual contracts help developers, and for which kinds of faults they are most effective.

To conduct the experiment, an implementation of the Rental Car Service was documented in natural language, seeded with 8 faults and provided with several short test cases able to detect them. Tests were executed and results recorded in two different formats: (A) as sequences of invocations and returns of operations from the interface, with queries added to display details of the internal state after each step and (B) as sequences of visual contracts extracted from the same invocations. Students were asked to (1) identify invocations where the observed behaviour deviated from the expected based on the documentation and (2) locate the faults responsible in the code provided. Both groups received reports from 4 tests of 4-5 invocations each, containing a total of 20 failures to be traced down to the 8 seeded faults.

The 66 participating students were volunteers from an MSc module on (UML-based design, implementation and testing of) Service-oriented Architectures running February-May at the University of Leicester. We could use data from previously submitted coursework, one on modelling and one on implementation and testing, to check that the average level of qualification of participants in both groups was comparable. The groups A and B were selected randomly (handing out worksheets A and B alternatingly), resulting in 32 students

in group A with an average coursework mark of 67.4% and 34 students in group B with an average coursework mark of 68.1%. From the module, the students were broadly familiar with the concept of specification-based testing of service interfaces like the one provided. The Car Rental Service interface, its documentation and the two types of assignments were introduced to all students in a 50 min session prior to the experiment. The participants then had 50 mins under exam conditions to analyse test reports, detect and document failures and locate the corresponding faults in the code provided.

Group A achieved an avg. recall of 0.215 (identifying 1.7 out of the 8 faults) and an avg. precision of 0.232 (with 1.7 correct out of 7.4 responses). Group B had an avg. recall of 0.3 (correctly identifying 2.41 out of 8) and an avg. precision of 0.35 (with 2.41 correct out of 6.88 responses). This represents a factor of improvement *recall B / recall A* of $0.3/0.215 = 1.4$ and *precision B / precision A* of $0.35/0.232 = 1.5$. In both cases, the t-test for independent two-sample experiments (for unequal variances and population sizes) showed that the results are statistically significant with a probability (p-value) of 0.033 for recall and 0.013 for precision. The p-value was calculated using an online tool[1] for a degree of freedom of 64 (the sum of population sizes -2), a significance level of 0.05, and a one-tailed hypothesis (there is a reasonable expectation that group B would perform better than group A). That means, assuming the null hypothesis that "the different representations of test reports in both groups have no effect on the resulting scores" is true, there is a 0.033 resp. 0.013 probability of observing the same results due to random sampling error. The key figures are summarised in Table I.[2]

TABLE I: Statistical data for groups A and B

	recall	precision
A mean	0.215	0.232
A std dev.	0.196	0.212
B mean	0.3	0.35
B std dev.	0.18	0.209
t-test	1.875	2.284
p-value	0.033	0.013

We investigated more closely which faults in which operations were detected more frequently by which group. The numbers are too low to have statistical significance, but suggest that the differential benefit of using visual contracts is greater with faults that involve structural features rather than those that concern attributes and parameter values only, such as

- *makeReservation()* does not check the *of* link between *Branch* and *Client* object;
- *dropoffCar()* does not remove the *Reservation* object.

The visual representation seems to be less effective for detecting faults in postconditions than in preconditions. In fact, there are two examples of structural postcondition faults that were detected with higher frequency by group A than B, i.e.,

- *cancelReservation()* deletes all reservations for the relevant client, rather than only the one specified by the parameter;
- *pickupCar()* does not delete the *pickup* link.

Indeed to understand the structural effect of a rule we have to spot the differences between its left- and right-hand side, which can be difficult if the structure is complex and there are several changes. This could be addressed, for example, by using different colours to highlight changes.

The highest relative benefit of visual contracts (13 discoveries in group B vs. 1 in group A) was observed for *registerClient()* (see top right of Figure 1) where according to the documentation, the client id returned should have been formed as *city* + "_" + *Branch.cMax* while in fact was computed as *city* + "_" + *Branch.of.size()* using the size of the client list rather than the next free client number *cMax*. To detect this problem requires matching information from pre and postcondition, including the navigation of the link between *Client* and *Branch* object, and the return value. Indeed, one advantage of visual representations is that they are not linear, and so able correlate items of information across more than one dimension.

Threats to Validity: While it is unlikely (see above) that results are due to random error, the design of the experiment itself could have biased the outcome. The (self) selection of participants may have resulted in groups that are not representative of the software developers normally concerned with testing tasks or could have provided an advantage to one of the groups. However, testing is often performed by junior developers. Many of our MSc students, mostly international with a broad range of backgrounds, would expect to go into entry level developer roles after graduation. As stated earlier we checked that both groups were equally capable based on their academic performance on a related MSc module that matched well with the expertise required in this task.

The relatively poor performance overall is a cause for concern. We believe this is due to the limited time to understand and perform a quite complex task, and the lack of practical experience of the participants, but also caused by the paper-based nature of the exercise, where a debugging tool providing similar representations in a more interactive, navigable way could improve outcomes. It is worth stressing, however, that the study does not claim the visual approach to be effective in absolute terms, only that it works better than the textual one in this artificial setting. This indicates that it might provide advantages in related practical tasks as well, but this is yet to be demonstrated.

There could be bias in the representation of information to both groups. Of course, since the hypothesis claims that the visual representation is more useful, this "unfair advantage" is intended. Apart from that the information provided is equivalent: invocations with actual parameters and returns are shown textually in both cases, only information on the internal state (object structure and attribute values) is represented differently, in group A by query operations listing all accessed objects and their state and in group B by visual contracts extracted.

[1]Social Science Statistics, P Value from T Score Calculator, http://www.socscistatistics.com/pvalues/tdistribution.aspx

[2]All documents and instructions handed out to both groups as well as the raw data and detailed calculations are available at http://www.cs.le.ac.uk/people/amma2/experiment

The choice of case study, with its dominance of structural features and their manipulation rather than computations on data, limit the validity of results to just such applications. This is justified by the fact that this is the natural domain for visual contracts. The NanoXML and JHotDraw case studies provide further examples of that nature.

D. Scalability

We use two case studies to evaluate scalability to large numbers of invocations and large object graphs. The case studies are based on NanoXML and JHotDraw[3] , both popular benchmarks for software testing and analysis, and representative of the kind of system our method would be appropriate for, i.e., with significant and dynamic object structures in their core model. In NanoXML this is the object representation of the XML tree, for JHotDraw that of graphics' objects.

NanoXML is a small non-validating XML parser for Java, which provides a light-weight and standard way to manipulate XML documents. We use version 2.2.1 which consists of three packages and 24 Java classes. We focus on two classes, *XMLElement* and *XMLAttribute*, which provide the functionality to manipulate XML documents. We monitor all *XMLElement* methods, executing 5605 test cases in order to evaluate the handling of large numbers of invocations. The original test cases were generated by CodePro[4], some modified and completed manually to improve coverage. These tests cover 2099 out of 5836 instructions. In Figure 9 we plot the time taken to execute different batch sizes of tests, from 59 to 2183. Each test generates a single rule instance from which minimal and maximal rules, multi-objects and constraints are extracted. Tracing, contract instance construction and extraction of minimal rules are essentially linear, as is the derivation of constraints and multi objects. The construction of maximal rules requires to compare all rule instances with shared minimal rules, which is quadratic in the number of rule instances that share the same effect.

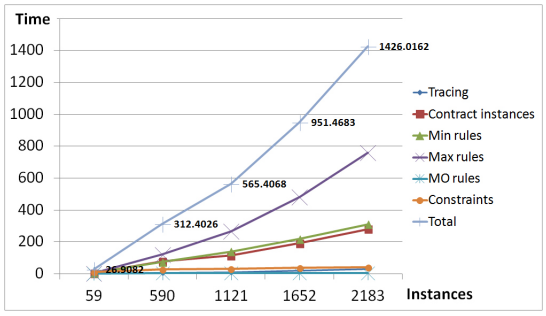


Fig. 9: Scalability for extracting contracts from NanoXML

JHotDraw is a Java GUI framework for technical and structured graphics, developed as an exercise in good software design using patterns. We used version 5.3 which has 243 classes, focussing on the top level methods for the manipulation of graphs, such as **.addFigure(..), *.DeleteFigure(..), *.copyFigure(..), *.DecoratorFigure(..)* and all undoable actions in **.CommandMenu.actionPerformed(comExe)*. We use GUI testing using WindowTester[5] to generate test cases by recording user interactions. We executed 405 test cases that cover 9284 of 34710 instructions. Based on the recorded test cases, the total runtime of the extraction is about 3 hours 15 mins. Scalability is analogous to NanoXML, see Figure 10, but the quadratic component of maximal rule extraction is less significant due to the smaller overall number of rule instances.

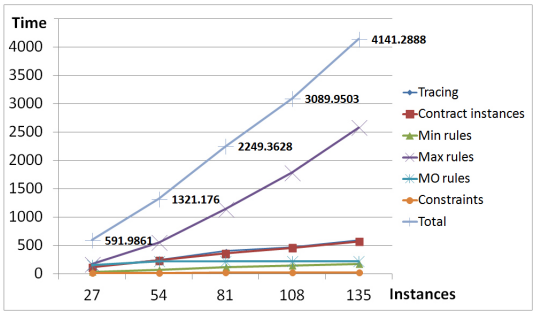


Fig. 10: Scalability for extracting contracts from JHotDraw

Unlike NanoXML where the number of invocations / contract instances is large but the size of each contract instance small, JHotDraw produces contract instances up to several hundreds of objects. Table II shows the number of objects accessed, number of instances, maximal rules, and rules with MO created (with total size in terms of numbers of objects).

TABLE II: JHotDraw objects accessed and processed for construction of contracts

Executed method signature	executed objects	instance rules	max rules	MO rules
CopyCommand.execute()	20150	16(400)	3(80)	0
add(Figure)	11106	24(332)	2(26)	0
DeleteCommand.execute()	494971	15(6259)	2(828)	1 (207)
DecoratorFigure.decorate(Figure)	2215	20(90)	2(10)	0
UndoableCommand.execute()	651671	60(19060)	10(2088)	1 (209)
	number (and size) of rules			

Based on these results we conclude that scalability may be acceptable for batch processing moderately sized test suites, but not necessarily for interactive testing. In applications to program understanding and debugging, however, where the human effort is significant, the time taken to prepare a more effective representation for inspection is likely to pay off, and our user study indicates that such benefits may be expected. The number of cases where multi objects could be identified is relatively small (2 out of 19 maximal rules) but they covered a large number of objects that may be hard to survey without this added level of abstraction.

The overall evaluation provides some confidence in the validity of the technology, the usefulness of the results and the scalability of the tool, but these aspects were evaluated

[3]See http://nanoxml.sourceforge.net/orig/ and www.jhotdraw.org/

[4]A JUnit test case generator https://developers.google.com/java-dev-tools/codepro/doc/features/junit/test_case_generation

[5]A tool to record GUI tests for Swing applications, https://developers.google.com/java-dev-tools/wintester/html/gettingstarted/swing_sampletest

through separate experiments on a range of different cases. There is no direct evaluation of the usability of the tool or of the absolute effectiveness of the approach in applications to program understanding and testing because such claims are beyond the scope of the paper.

IV. Related Work

Reverse engineering visual contracts is a process of learning rules from transformations. This has been suggested in a number of areas, including the modelling of real-word business processes [11], biochemical reactions [12] and model transformations [13]. Although related in the aim of discovering rules, the challenges vary based on the nature of the graphs considered, e.g., directed, attributed or undirected graphs, the availability of typing or identity information, etc. We organise the discussion in two levels: tracing to construct rule instances and learning to infer high-level rules with advance features

Model Extraction: Automated reverse engineering is based on static or dynamic analysis. The static approach, exemplified by [14], [15], [16], examines the source code only, with the intention of extracting all possible behaviours. This is useful for incomplete systems, e.g., components that cannot be executed independently [15], but limited in its ability to detect dynamic object-oriented behaviours such as dynamic binding. The drawback of a dynamic approach, such as ours but also [17], [18], [19], is that the extracted model represents only those behaviours that are actually executed. In particular [19] uses AspectJ for extracting a context-free graph grammar but 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.

Learning Models from Observations: [11] propose mining algorithms for graph transformation systems from transition systems. Their *context algorithm* provides similar outputs to our inferred maximal rule, but we differ in the strategy used. Their construction relies on extending the minimal rule by adding matched context elements. Our approach is the opposite, based on cutting down unmatched contexts from a chosen contract instance, which makes it easier to maintain the graph structure as valid against the type-graph. To the best of our knowledge, no work has been done on inferring multi-objects for visual contracts.

In [12] source and target graphs represent networks of biomolecules. The authors aim to discover rules modelling reactions. They extract the minimal rule by best sub-graph matching and adopt a statistical approach to rate context. Our approach is simpler in that the minimal rule is determined by tracing and we do not deal with uncertainty of context.

Considering approaches to learning model transformations [20], we distinguish *in-place* where source and target have the same metamodel and *out-place* transformations where the metamodels are different [21]. For learning *out-place* transformations, [13] use input-output pairs representing the result of a transformation process rather than a single step. [22], [23], [24] also address the learning of *out-place* transformations, while our approach focusses on *in-place* transformations.

[25] also addressing the learning of *in-place* transformations is interactive, requiring confirmation of the rules proposed. Our approach does not rely on direct user involvement and, significantly, is not based on a small number of carefully hand-crafted examples, but on large numbers of observations extracted from a running system. Therefore, scalability and the ability to deal with example sets providing incomplete coverage are important.

Graph pattern mining: An algorithmic problem closely related to the extraction of rules from example transformations is graph pattern discovery. Current approaches are statistical or node signature-based. Finding graph patterns by statistical means is popular in machine learning [26], but can produce a large variance depending on the frequency of a pattern. For instance, an object that is not accessed, but always present in the context, is considered an important element of the rule.

[27], [28] discuss research in exact and best graph pattern matching. A crucial point is how to distinguish nodes as candidates for possible matches. In [29], a node signature for attributed graphs encodes node/edge types and node attributes. We use a node signature-based approach with added structural information and metadata, extended from subgraph to subrule matching, taking into account shared minimal rules and parameters.

V. Conclusion and Future Work

We presented an integrated approach and tool for learning visual contracts, from instrumentation of Java code and observation of tests to the derivation of general rules with multi objects and attribute constraints. It supports the analysis of tests based on a concise, visual and comprehensive representation of operations' behaviour. We have evaluated the validity of the resulting models, usability and scalability in experiments on three case studies.

Currently we work on improving the integration of our tool with Henshin [10] to evaluate extracted contracts more widely. This involves invoking the model alongside the original implementation with the same set of tests, comparing outputs for consistency. Executing the tests the contracts were extracted from adds to their validation of correctness, but more interestingly we can try a range of additional cases to evaluate how well contracts capture the wider behaviour, beyond the directly observed. A related idea is the use for adaptive testing [30] where test cases are generated from contracts in a cycle of test generation, execution, and contract extraction.

We also plan to use contract extraction to support testing and debugging, evaluating their effectiveness for these tasks more comprehensively. Additionally, we will be investigating techniques such as Hyper/J [31] to support tracing and extraction for multithread Java applications.

Acknowledgment

We would like to thank Michel Chaudron and Neil Walkinshaw for the valuable advice and feedback on conducting user experiments.

REFERENCES

[1] T. A. Khan, O. Runge, and R. Heckel, "Testing against visual contracts: Model-based coverage," in *Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings*, 2012, pp. 279–293.

[2] O. Runge, T. A. Khan, and R. Heckel, "Test case generation using visual contracts," *ECEASST*, vol. 58, 2013.

[3] G. Engels, M. Lohmann, S. Sauer, and R. Heckel, "Model-driven monitoring: An application of graph transformation for design by contract," in *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, ser. Lecture Notes in Computer Science, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, Eds., vol. 4178. Springer, 2006, pp. 336–350. [Online]. Available: http://dx.doi.org/10.1007/11841883_24

[4] J. H. Hausmann, R. Heckel, and M. Lohmann, "Model-based development of web services descriptions enabling a precise matching concept," *Int. J. Web Service Res.*, vol. 2, no. 2, pp. 67–84, 2005. [Online]. Available: http://dx.doi.org/10.4018/jwsr.2005040104

[5] A. M. Alshanqiti, R. Heckel, and T. Khan, "Learning minimal and maximal rules from observations of graph transformations," *Electronic Communications of the EASST*, vol. 58, 2013.

[6] A. M. Alshanqiti and R. Heckel, "Towards dynamic reverse engineering visual contracts from java," *Electronic Communications of the EASST*, vol. 67, 2014. [Online]. Available: http://journal.ub.tu-berlin.de/eceasst/article/view/940

[7] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer, *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. pp 12, 21,22, Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[8] D. Bisztray, R. Heckel, and H. Ehrig, "Verification of architectural refactorings: Rule extraction and tool support," *Proceedings of the Doctoral Symposium at the International Conference on Graph Transformation - Electronic Communications of the EASST*, vol. 16, 2009.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 13, pp. 35 – 45, 2007, special issue on Experimental Software and Toolkits. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016764230700161X

[10] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced concepts and tools for in-place EMF model transformations," in *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds., vol. 6394. Springer, 2010, pp. 121–135. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16145-2_9

[11] H. Bruggink, "Towards process mining with graph transformation systems," in *Graph Transformation*, ser. Lecture Notes in Computer Science, H. Giese and B. Knig, Eds. Springer International Publishing, 2014, vol. 8571, pp. 253–268. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09108-2_17

[12] C. h. You, L. B. Holder, and D. J. Cook, "Learning patterns in the dynamics of biological networks," in *Proceedings of the 15th ACM SIGKDD International conference on Knowledge discovery and data mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 977–986.

[13] X. Dolques, A. Dogui, J.-R. Falleri, M. Huchard, C. Nebut, and F. Pfister, "Easing model transformation learning with automatically aligned examples," in *Proceedings of the 7th European conference on Modelling foundations and applications*, ser. ECMFA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 189–204.

[14] M. K. Sarkar, T. Chatterjee, and D. Mukherjee, "Reverse engineering: An analysis of static behaviors of object oriented programs by extracting uml class diagram," *International Journal of Advanced Computer Research*, vol. 3, no. 3, 2013.

[15] A. Rountev, O. Volgin, and M. Reddoch, "Static control-flow analysis for reverse engineering of uml sequence diagrams," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 96–102, Sep. 2005.

[16] P. Tonella and A. Potrich, "Reverse engineering of the interaction diagrams from c++ code," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, 2003, pp. 159–168.

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

[18] T. Ziadi, M. A. A. Da Silva, L. M. Hillah, and 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)*. IEEE, 2011, pp. 107–116.

[19] C. Zhao, J. Kong, and K. Zhang, "Program behavior discovery and verification: A graph grammar approach," *IEEE Transactions on Software Engineering*, vol. 36, no. 3, pp. 431–448, 2010.

[20] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer, "Conceptual modelling and its theoretical foundations," in *Conceptual Modelling and Its Theoretical Foundations*, A. Düsterhöft, M. Klettke, and K.-D. Schewe, Eds. Berlin, Heidelberg: Springer-Verlag, 2012, ch. Model transformation by-example: a survey of the first wave, pp. 197–215.

[21] T. Mens and P. Van Gorp, "A taxonomy of model transformation," *Electron. Notes Theor. Comput. Sci.*, vol. 152, pp. 125–142, Mar. 2006.

[22] M. Faunes, H. Sahraoui, and M. Boukadoum, "Generating model transformation rules from examples using an evolutionary algorithm," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 250–253.

[23] D. Varr, "Model transformation by example," in *In Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML)*. Springer, 2006, pp. 410–424.

[24] Z. Balogh and D. Varr, "Model transformation by example using inductive logic programming," *International Journal - Software and Systems Modeling*, vol. 8, no. 3, pp. 347–364, 2009.

[25] P. Langer, M. Wimmer, and G. Kappel, "Model-to-model transformations by demonstration," in *Proceedings of the Third international conference on Theory and practice of model transformations*, ser. Lecture Notes in Computer Science, L. Tratt and M. Gogolla, Eds. Springer Berlin Heidelberg, 2010, vol. 6142, pp. 153–167.

[26] M. Qiu, H. Hu, Q. Jiang, and H. Hu, "A new approach of graph isomorphism detection based on decision tree," in *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*, vol. 2. IEEE, 2010, pp. 32–35.

[27] D. Conte, P. Foggia, C. Sansone, and M. Vento, "Thirty years of graph matching in pattern recognition," *International journal of pattern recognition and artificial intelligence*, vol. 18, no. 03, pp. 265–298, 2004.

[28] N. Dahm, H. Bunke, T. Caelli, and Y. Gao, "Efficient subgraph matching using topological node feature constraints," *Pattern Recognition*, vol. 48, no. 2, pp. 317 – 330, 2015.

[29] S. Jouili, I. Mili, and S. Tabbone, "Attributed graph matching using local descriptions," in *Advanced Concepts for Intelligent Vision Systems - Acivs 2009*, ser. Lecture Notes in Computer Science, SEE. Springer, 2009, pp. 89–99.

[30] K.-Y. Cai, T. Y. Chen, Y.-C. Li, W.-Y. Ning, and Y. T. Yu, "Adaptive testing of software components," in *Proceedings of the 2005 ACM Symposium on Applied Computing*, ser. SAC '05. New York, NY, USA: ACM, 2005, pp. 1463–1469. [Online]. Available: http://doi.acm.org/10.1145/1066677.1067011

[31] H. Ossher and P. Tarr, "Hyper/j: Multi-dimensional separation of concerns for java," in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 734–737. [Online]. Available: http://doi.acm.org/10.1145/337180.337618