Noname manuscript No.
(will be inserted by the editor)

Inferring Visual Contracts from Java Programs

Abdullah Alshanqiti · Reiko Heckel · Timo Kehrer

Received: date / Accepted: date

Abstract Visual contracts model the operation 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. Their detailed specification of internal data states and transformations, referred to as deep behavioural modelling, is an error-prone activity. 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, patterns, and invariants. 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.

Keywords Visual contracts · graph transformation · model extraction · dynamic analysis · reverse engineering · specification mining

Abdullah Alshanqiti
Department of Computer Sciences, University of Leicester, UK
E-mail: a.m.alshanqiti@gmail.com

Reiko Heckel
Department of Computer Sciences, University of Leicester, UK
E-mail: reiko@mcs.le.ac.uk

Timo Kehrer
Institut für Informatik, Humboldt-Universität zu Berlin
E-mail: timo.kehrer@informatik.hu-berlin.de

1 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,
- *abstract:* providing a specification of object transformations at a high level of granularity to aid readability and scalability,
- *deep:* capturing the transformation of internal object structures besides input / output behaviour,
- *executable:* based on graph transformation they support model-based oracle and test case generation [31,39], run-time monitoring [17], service specification and matching [23], state space analysis and verification.

However, creating a detailed behavioural model in any language is error-prone. Visual contracts are no exception, and their specification of internal 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 and patterns* and general invariants. 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 test cases for the relevant operations. We proceed by (A) observing the behaviour under these tests using AspectJ instrumentation and synthesising rule instances as pre/post snapshot 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 patterns; (D) deriving logical constraints and assignments over attribute and parameter values; and (E) identifying universally shared conditions and structures as invariants captured separately.

First solutions to variants of (A) and (B) were reported in [4,2], respectively, extended and elaborated in [1,3] which also provided performance improvements of the algorithms and their integration in a prototype tool. In this

paper, we raise further the level of abstraction by supporting, in addition to previous work, the inference of multi patterns in (C), attribute assignments in (D) and universal context in (E), and exploiting subtyping throughout the process. We support the use of visual contracts as *deep oracles* by exporting contracts to the model transformation tool Henshin and controlling the (otherwise non-deterministic) model execution by comparing the effect of each test invocation with the rules in the operation's contracts. This technology is used in a new evaluation of completeness (recall) and correctness (precision) of extracted contracts. Finally, we report on improved tool support.

Following a general presentation of the notions and techniques of the approach in section 2, section 3 describes the Visual Contract Extractor tool implementing them. The evaluation in section 4 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. In section 5, we discuss two further application scenarios for our approach and tool in the field of model-based software engineering, namely the use of visual contracts for model-based (or visual) debugging as well as the automated learning of complex model editing operations by examples. After discussing related work in section 6, section 7 concludes the paper.

2 Deep Behaviour Modelling and Extraction

This section gives an overview of our approach using the simple case study of a Car Rental Service designed to represent a range of preconditions and effects for operations over a complex object structure. These include the creation of objects, the creation and deletion of links, attribute updates and constraints. First, visual contracts and their semantic foundations in graph transformation are introduced, following [16].

2.1 Visual Contracts

A visual contract describes an operation by means of a set of rules, each representing the pre- and postcondition of a possible behaviour. Imagine a simple operation that can execute along one of three behaviours: success, handled failure (returning an error code) or exception, chosen according to the state in which the operation is invoked. Then, the first two are represented by individual rules capturing their respective conditions and effects. The exception behaviour corresponds to the absence of a suitable rule, i.e., the failure to satisfy any of the existing rule's preconditions. We will use the term *behaviour* to refer to the set of actions and conditions executed along a path or a set of paths through the control flow graph of the system, to be captured in the contract by a single rule.

```
public interface IRental extends Serializable{
    public String registerClient(String city, String clientName);
    public String makeCarReservation(String ClientID, String pick−up, String drop−off);
    public String makeVanReservation(String ClientID, String branch);
    public void cancelReservation(String Reference);
    public void cancelClientReservation(String clientID);
    public void pickupVeh(String Reference);
    public void pickupFleet(String branch, String ClientID);
    public void dropoffVeh(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

Contracts are based on a class diagram and operation signatures. An interface with operation signatures is given in Listing 1. The class diagram in the top left of Figure 1 shows the classes whose instances are considered within the scope of the specification. Classes and data-valued attributes in the diagram map to classes and attributes in Java. Associations with cardinality 0..1 at the target end 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: a distinguished graph defining node, edge, attribute and data types over which instance graphs can be formed. An *instance 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. We allow type graphs with inheritance, which specify a subtype relation between node types. Instance graphs are to type graphs with inheritance as object diagrams are to class diagrams, and adopt the same notation.

Our aim is to derive rules of the form $r : L \Rightarrow R$. Graphs L and R, called the *left-* and *right-hand side* of the rule, provide a declarative specification of pre- and postconditions (effects), where $L \setminus R$, $L \cap R$ and $R \setminus L$ represent the elements to be deleted, preserved and created by an operation.

Formally, a rule according to [16] is a span of graph homomorphisms $L \leftarrow K \rightarrow R$, where we think of K as representing $L \cap R$, the structures read but not consumed by an application of a rule. Rules can be augmented by conditions on attributes and input parameters and they can specify attribute updates and return values. If the left-hand side L_r of rule r finds an occurrence in graph G, formally expressed as a type-compatible injective graph homomorphism $o : L \to G$, it can be applied to this graph leading to a transformation $G \overset{r,o}{\Longrightarrow} H$. In the presence of subtyping, the types of rule elements in L or R must be equal to or more general than the types of their corresponding graph elements in G or H.

We use rule schemata for the concise specification of transformations over recurring model patterns. A *rule schema* (s, M) consists of a *kernel rule* s and a set M of *multi rules*. Each multi-rule $m \in M$ is an extension of s by additional preconditions and effects. Each multi rule m specifies a *multi pattern* $L_m \setminus L_s$ and its transformation into $R_m \setminus R_s$. When a rule schema is applied,

kernel rule s is executed once like a "normal" rule while each multi rule m is applied to all distinct occurrences of its multi pattern simultaneously.

In summary, given an attributed type graph TG with inheritance and an operation signature $op(x_1 : T_1 \ldots, x_n : T_n) : (y : T)$, a visual contract vc for op is a set of rules $r : L \Rightarrow R$ over TG such that parameters $x_1, \ldots, x_n$ occur in L and the return y occurs in R.

2.2 Observing Object Access and Synthesising Rule Instances

Visual contracts are inferred by observing executions of Java methods. For each operation invocation we extract a *rule instance* capturing the recorded behaviour. Observations are made by weaving instrumentation code using AspectJ. This results in a trace recording object creation, read and write access to objects and attributes caused by the invocation.

In a sequential execution all actions observed between the start and the return of the method can be attributed to one invocation. Concurrent invocations make it more difficult to identify the relevant invocation for each action and are therefore not considered. We aggregate all observations into a rule instance capturing the overall precondition and effect of the invocation (see [2] for more details). Along with the instance we collect traceability data for its elements, such as the line numbers of corresponding statements in the code. This is used later to validate the extraction, e.g., to assess which code fragments are captured by which contracts.

Consider the rule instances in Figure 1. Instance *registerClient* creates a new client object, registers it with the branch at *city*, and updates attribute *branch.cMax*. Instance *makeCarReservation* 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 reserves a car from the *pickup* branch they are registered with, but with a different *dropoff* branch. Instance *makeVanReservation* performs a similar operation for vans, where *pickup* and *dropoff* are the same. Rule instances for *pickupVeh* and *dropoffVeh* record the movement of a car from the *pickup* to the *dropoff* branch. Note that while these two operations are defined for the abstract class *Veh*, these instances use *Car* objects.

As can be seen in the example, a rule 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 $op(x_1 : T_1 \ldots, x_n : T_n) : (y : T)$ 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.

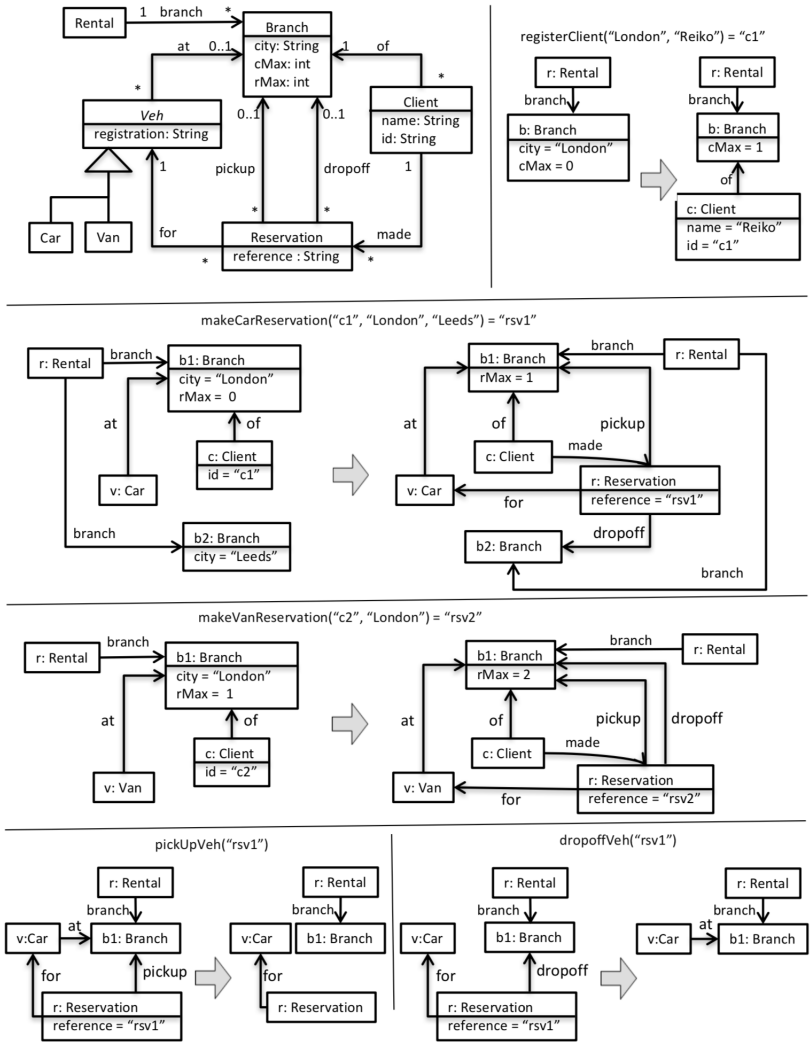


Fig. 1: Type graph and rule instances, extracted from car rental service

2.3 Deriving Minimal Contracts and Shared Context

Each rule instance only represents one invocation. Our aim is to derive a small set of rules that describe the overall behaviour of the operation, i.e., the operation's contract. Thus, a contract is 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 infer a general specification 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 first formalised in [11] and implemented (without considering parameters) in [4]: Given a rule 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. In [11] this is expressed more abstractly using category theory. Specifically the notion of initial pushout allows us to capture the differences of graphs $G \setminus H$ as the part deleted and $H \setminus G$ as the part created by the transformation, which is then extended by the necessary context to form a rule. The implementation in [4] identifies elements shared between G and H by their object identities.

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.

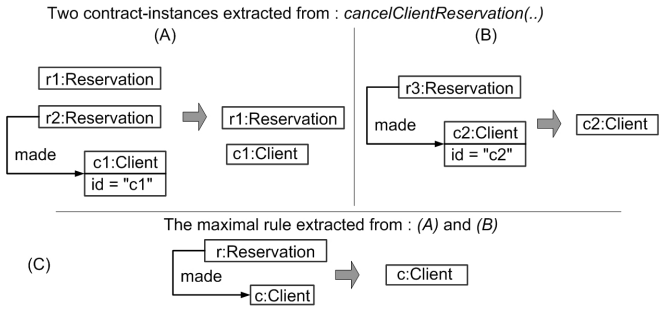


Fig. 2: Extracting maximal rules from rule instances

But minimal or maximal rules are not just generalisations of instances. They provide an executable 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 rule instance

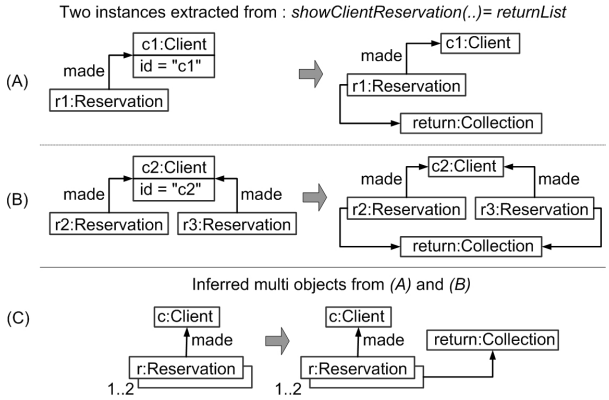


Fig. 3: Inferring MOs from rule instances

$b = op(a_1, \ldots, a_n) : G \Rightarrow H$ 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.

2.4 Introducing Universally Quantified Rule Schemata

The contracts extracted so far may use many rules to describe a single 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. Rule schemata use multi objects (MO) or multi patterns (MP) as a concise way to specify constraints and actions across sets of similar structures.

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*. Rule instances of two corresponding transformations are shown in Figure 3 (A) and (B).

2.4.1 Multi Object Inference

To extract 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 subsection 2.3, 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 $CMOR(op)$.

Consider again the example in Figure 3. The rule in (C) is a combination of the basic rule in (A) with the MO rule derived from (B) whose cardinalities of 1 and 2 for the *Reservation* node are merged to 1..2. In rule (B) we identify Reservation objects $r2$ and $r3$ as equivalent. Merging them leads to a rule isomorphic to rule (A) so both are combined to one MO rule with multi-object r.

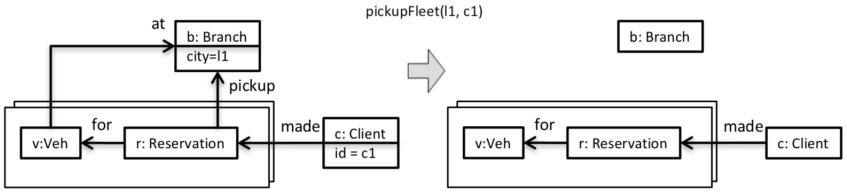


Fig. 4: Rule with multi pattern

2.4.2 Multi Pattern Inference

Occasionally we require universal quantification not just on a single object but on a more general structure. Multi patterns provide this extension. To derive rules with multi patterns we discover *graph fragments* within a rule r that are equivalent, having the same shape, connections and transformation behaviour, and thus can be represented by a multi pattern. A graph fragment consists of nodes and edges that do not necessarily form a graph themselves because

edges in the fragment maybe connected to nodes outside the fragment, called *boundary nodes*.

Figure 4 shows how operation *pickupFleet()* is described as a rule with multi pattern. The operation allows to pick up the set of all vehicles reserved exactly once, by the client with id *c1*, deleting their *at* and *pickup* edges.

We formalise the notion of *rule fragment* and their *equivalence*. Let $r : L \Rightarrow R$ be a rule, e.g., as derived in step (2) as maximal rule. Let further $p = (FL, FR)$ be a pair of graph fragments with $FL \subseteq L$ and $FR \subseteq R$ such that the effect described by p is included in that described by r. That means, the rule fragment deletes and creates a subset of the objects deleted and created by the rule; formally: $FL \setminus FR \subseteq L \setminus R$ and $FR \setminus FL \subseteq R \setminus L$.

In *pickupFleet()* the left- and right graph fragments FL and FR consist of all nodes and edges inside the respective boxes, including the edges crossing the boundaries. The boundary nodes of FL are the *Branch* and *Client* objects, while FR's boundary is just the *Client* object.

Two rule fragments p and p' in r are equivalent if (i) their graph fragments are isomorphic (share the same shape, typing and attributes), and (ii) they have the same external connections within r (share the same boundary nodes).

A rule schema (s, M) consists of the *kernel rule* s and a set of *multi-rules* M. The kernel rule is formed by removing from r all instances of the rule fragments previously identified. That means, it contains only the elements that are to occur exactly once in the transformation. For every rule fragment p, rule $m = s \cup p$ forms a multi rule in M, extending the kernel rule by the rule fragment. Analogous to multi-object rules, if there are isomorphic rule fragments p and p' in r, we select a representative p.

That means, *pickupFleet()* is a rule schema $(s, \{m\})$ consisting of a kernel rule s, given by objects *b: Branch, c: Client* in both left- and right-hand side, and a multi rule m as the complete rule, including the boxed multi pattern.

2.5 Deriving Conditions and Assignments on Attributes and Parameters

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 logical constraints and assignments.

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 [18] 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 (attribute assignment) 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.

2.6 Derivation of Universal Context

Preconditions present in all rules are candidates for invariants. For example, the *Rental* object representing the rental agency is present in all preconditions, and therefore in all observed states, and is considered as an invariant that can be specified once and for all in the class diagram. To extract and cut such invariant context, we employ a similar procedure as for the derivation of maximal rules. That is, we compare the preconditions of all maximal rules to identify structures that are universally present. Universal context presented as global invariant can reduce the size of rules, make them more concise and readable.

Figure 5 shows the end result of the process for the pickup and dropoff rules. Note how attributes and parameter values are replaced by variables, with a condition $\{rid = ref\}$ to map the parameter to the reservation's reference, while *Car* objects are replaced by objects of class *Veh*, and the *Rental* objects linked to the branches are removed as universal context.

3 The Visual Contract Extractor (VCE) Tool

We developed a proof-of-concept tool to evaluate the approach and experiment with its use in different application scenarios. The components of the VCE tool are shown in Figure 6.

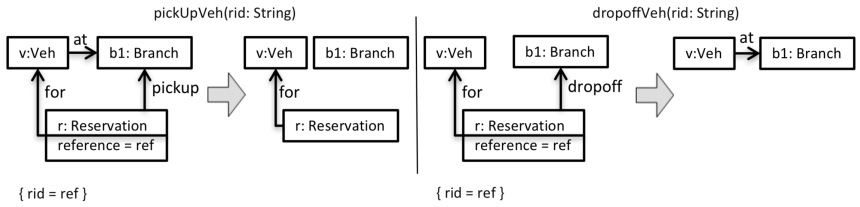


Fig. 5: Final pickup and dropoff rule

– (A) The *Tracer* observing the behaviour of selected classes using AspectJ and constructing contract instances (cf. subsection 2.2);
– (B) the *Generaliser* for generalising contract instances to minimal and maximal rules (cf. subsection 2.3) ;
– (C) the *Inferencer* for learning MO rules and rule schemata (cf. subsection 2.4), attribute conditions and assignments using Daikon (cf. subsection 2.5), as well as universal contexts (cf. subsection 2.6).
– (D) the *Visualiser* for selective display and analysis of contracts; and
– (E) an *Export and Model Control* facility, particularly to the model transformation tool Henshin [5,42] for generating executable contracts and controlling their execution as oracles alongside testing.

VCE is implemented in Java and relies on a MySQL database as back-end to efficiently handle large (numbers of) contracts. Screenshots in this section are taken by applying the tool to two further case studies NanoXML and JHotDraw[1], 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. They are also used for performance evaluation in section 4.

3.1 Tracer

To selectively trace large Java programs, the *Tracer* can be configured *i)* by selecting the relevant classes to define the scope of object types, and *ii)* by identifying methods of interest as each invocation of these methods will produce a single contract instance, covering those objects which are typed over the selected classes.

3.2 Generaliser and Inferencer

Figure 7 (a) shows a maximal rule based on the NanoXML case study. The operation *addChildren()* adds a set of *XMLElement* instances as children to

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

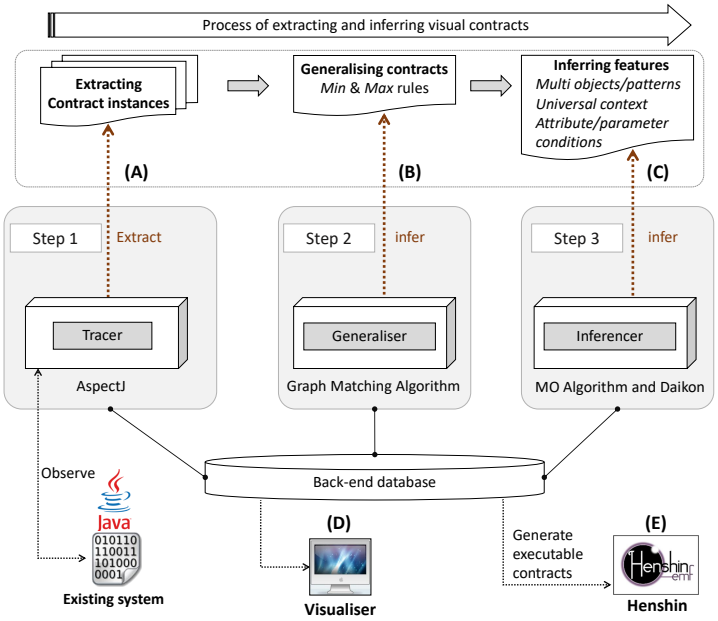


Fig. 6: Overview of the VCE tool

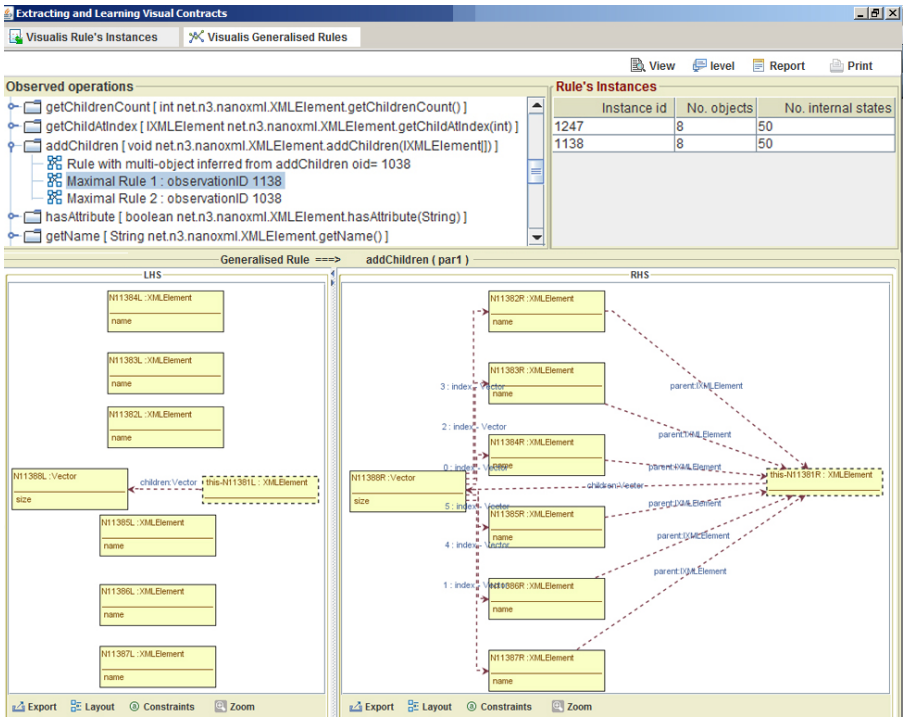
the element it is invoked on. The top left shows a list of rules organised by operation signatures. When selecting, e.g., a maximal rule, all its rule instances will appear in the table at the top right. The lower part of (a) shows the inferred maximal rule, which has 6 equivalent *XMLElement* instances. These are combined in a multi object in Figure 7 (b). The rules also show the *Vector* container used to store child elements, which can be visualised more abstractly by a direct to-$*$ association between the element and its children.

Figure 8 shows a maximal rule with inferred attribute conditions extracted for *addFigure(..)* from the JHotDraw case study. Attribute conditions are shown in a separate dialogue window for selected nodes. For example, in the popup window for the selected LHS node of type *BouncingDrawing* the two top constraint express preconditions (P3) while the 4th constraint is part of the postcondition (P4) stating that the value of its attribute *theQuadTee* must remain unchanged in the post graph.

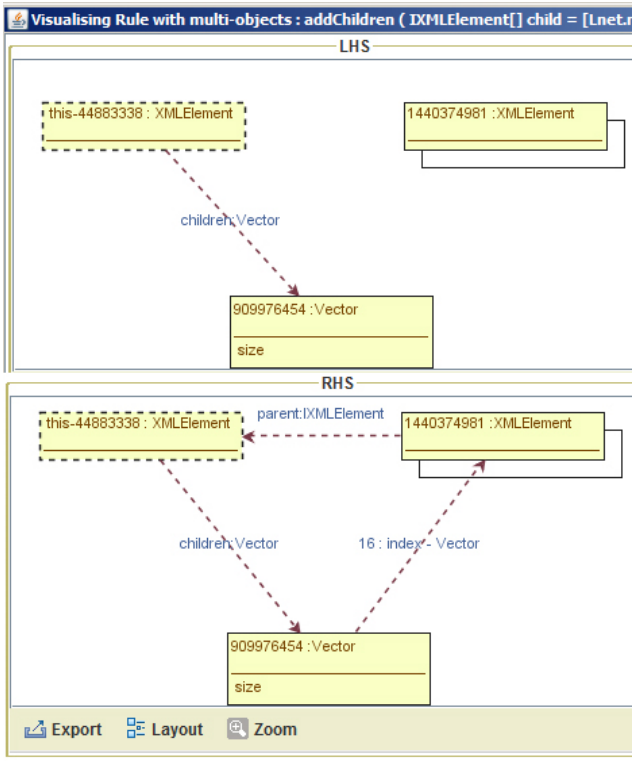
3.3 Visualiser

The role of the visualiser, see Figure 7 and Figure 9, is to organise, browse and display extracted contracts. We support:

– *i)* The distinction in colour and style between elements of the minimal and maximal rule; dotted edges and nodes with coloured background (green for creation, red for deletion and light-golden for nodes with updated attribute values) represent elements of minimal rules, while nodes with white background and solid edges are context elements;



(a) Maximal rule



(b) Rule with multi object

Fig. 7: Generalisation of rules

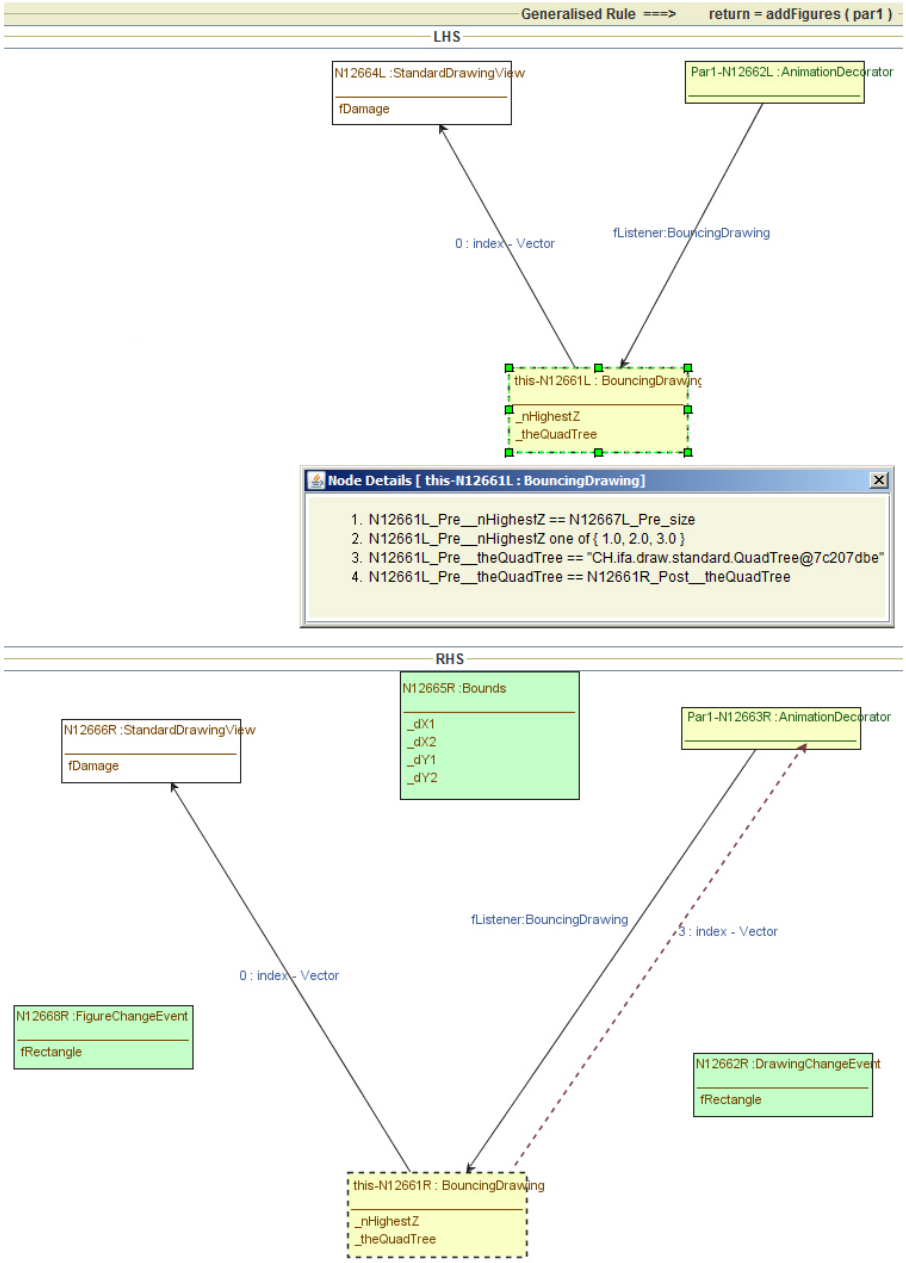


Fig. 8: A rule with attribute conditions.

– *ii)* the alternative display of collections as to-* associations or using explicit collection objects;
– *iii)* the selective visualisation of rules, e.g. the minimal rule or the pre-condition only, with the flexibility to change graph layouts; and
– *iv)* user interaction to confirm if inferred features are correct.

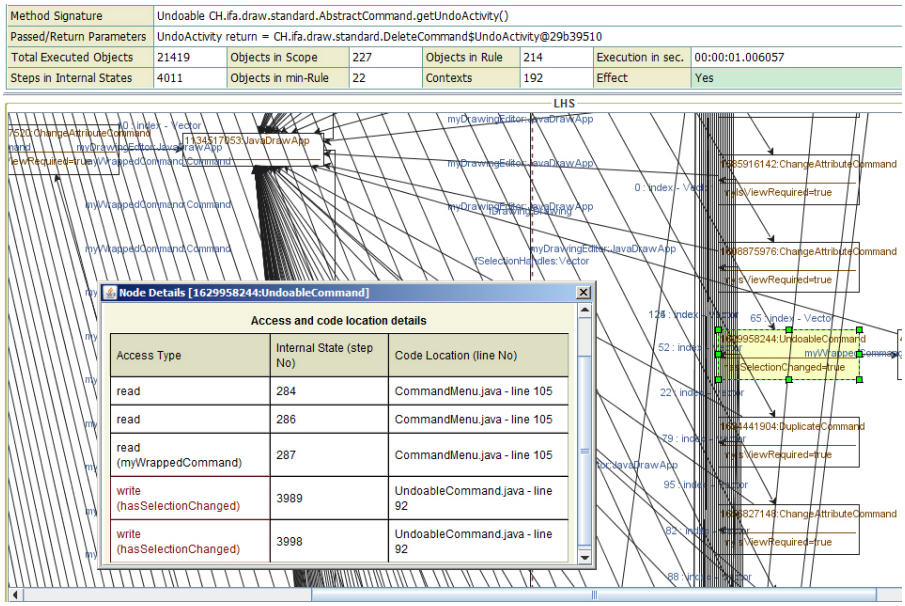
Figure 9 shows partial screenshots of the main interface. In (a), we present an instance extracted from a test as part of the JHotDraw case study. The upper part of (a) gives information on the operation signature, actual parameters and statistics on the extraction process. The lower part shows a contract instance, focussing on a fragment of the left-hand side. As reported, the rule has 214 objects obtained after analysing and filtering 21419 Java objects. By selecting an object such as the instance of *UndoableCommand* a popup menu shows tracing information with corresponding code locations.

The VCE tool provides visualisation options that may be employed to view complex contracts. As an example, consider Figure 9 (b) which shows for the same operation *DeleteFigure(..)* how inferring a rule with multi-objects and displaying its minimal rule, hiding all elements of the precondition that do not contribute to state changes, we obtain a much more readable presentation.

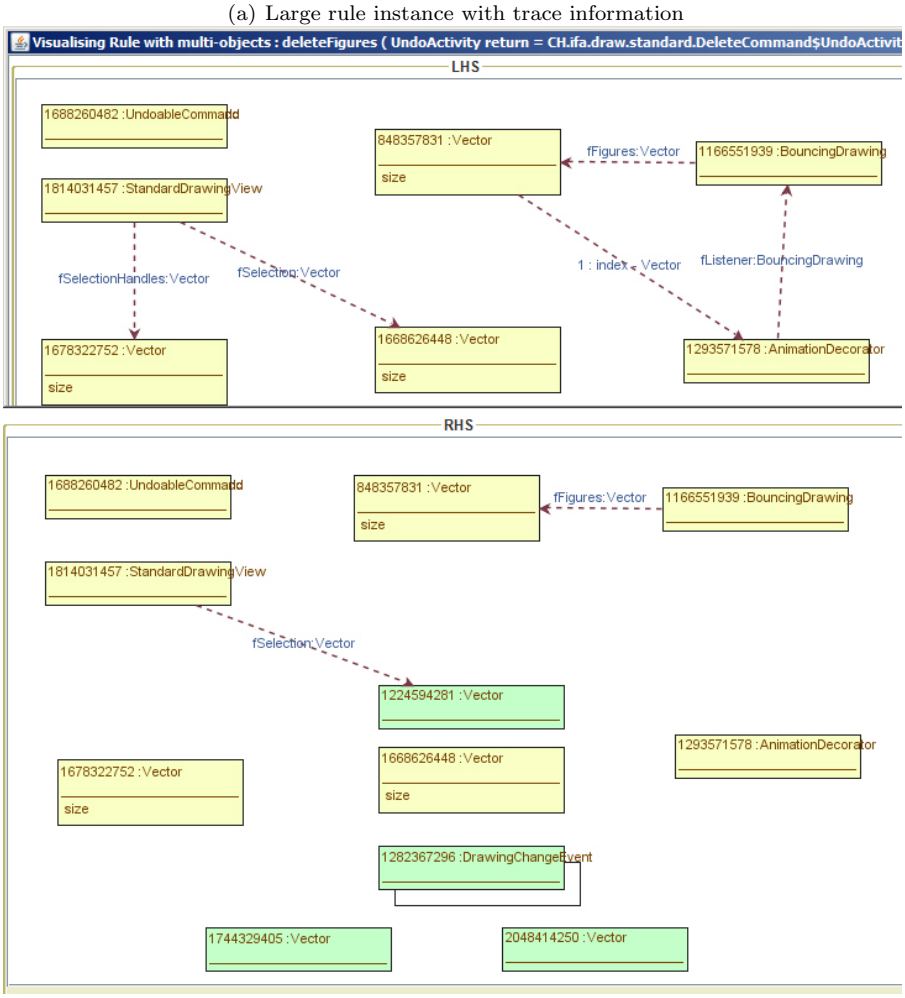
3.4 Export to Henshin

To interface with the tool, contracts can be exported to standard formats such as GXL or DOT graphs. Most notably, however, VCE supports the export of generalised rules to Henshin [5,42], a model transformation language and system which is based on graph transformation concepts [16] and allows to execute rules on (model representations of) object graphs. This allows to use visual contracts as test oracles, which is interesting in itself but also essential for the evaluation of inferred contracts as in the experiment presented in subsection 4.2.

The transformation of the structural parts of a contract is straightforward. Pre- and post graphs of a contract rule are mapped to the left- and right-hand sides of a Henshin rule, equipped with a mapping that signifies the corresponding rule nodes. For example, Figure 10 shows three generalised rules extracted from an experiment with NanoXML exported into Henshin. This notation shows transformation rules in an integrated form, the left- and right-hand sides of a rule merged into a single graph, following the concrete syntax of the Henshin transformation language. The left-hand side (LHS) comprises all model elements stereotyped by *delete* and *preserve*, the right-hand side (RHS) contains all model elements annotated by *preserve* and *create*. The rule in Figure 10 (a) corresponds to the maximal rule shown in Figure 7 (a). Another exported maximal rule for the same system operation *addChildren(..)* is shown Figure 10 (b). In Figure 10 (c), a Henshin rule scheme which generalises over both maximal rules is shown, it is exported from the contract with multi object shown in Figure 7 (b).

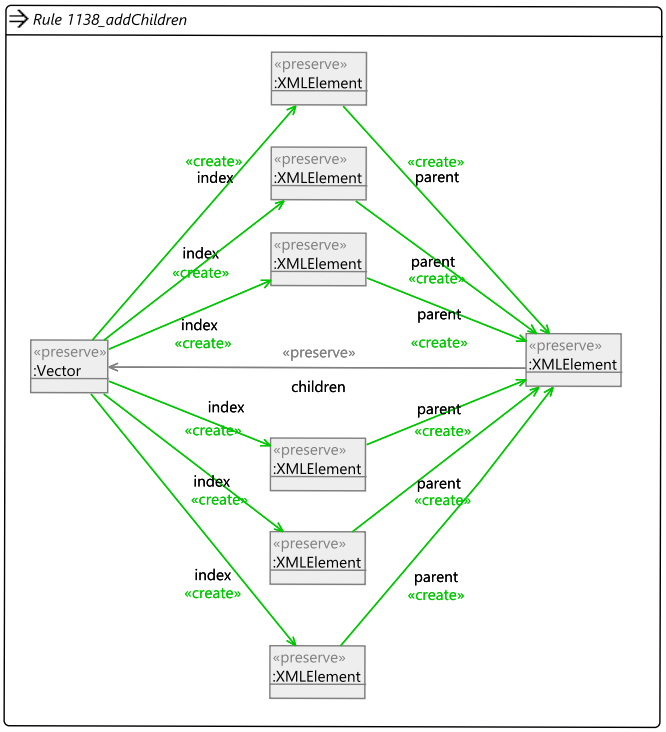


(a) Large rule instance with trace information

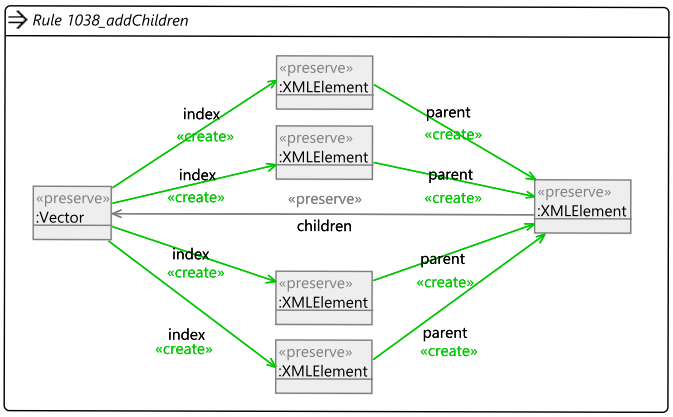


(b) Selective display of a generalised rule

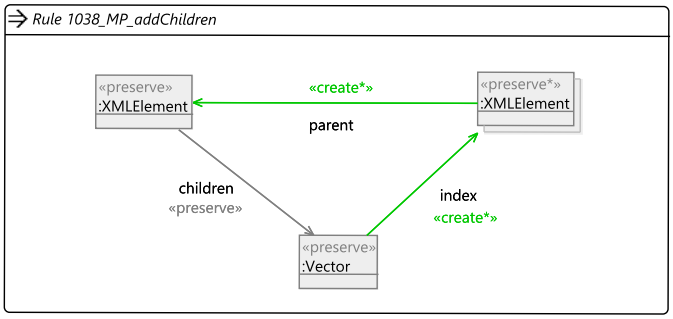
Fig. 9: Large rule instance and generalised rule extracted from *DeleteFigure(..)*.



(a) Henshin rule corresponding to maximal rule in Figure 7.



(b) Another exported maximal rule for the same operation.



(c) Henshin rule scheme generalising over both maximal rules.

Fig. 10: Export of generalised rules to Henshin.

For transforming the constraints learned by Daikon into Henshin, we define five profiles which can be used to configure our Henshin export facility, i.e., the behaviour of the exporter is defined by selecting one of these profiles:

- *P1 (None)*: The constraints learned by Daikon shall be ignored by the exported Henshin rules (as in the examples shown in Figure 10).
- *P2 (Parameters)*: Constraints over contract rule parameters are used to bind the corresponding Henshin rule parameters to rule node attributes. The Henshin rule parameters are derived from the constraints on-the-fly. This profile is useful when concrete parameter values shall be passed to Henshin rule applications.
- *P3 (Pre.Attributes)*: In addition to P2, in this profile we also export constraints which are preconditions over attributes and in which no parameters are involved. The the three top constraint in the popup window shown in Figure 8 are examples of this kind of constraints. In the exported Henshin rules, such constraints are represented as attribute conditions which are checked by the Henshin interpreter before a rule is applied and which thus may restrict the applicability of a rule.
- *P4 (Post.Attributes)*: In addition to the preconditions over parameters and attributes addressed by P2 and P3, respectively, in this profile we also translate postconditions over attributes and parameters. The the 4th constraint in the popup window shown in Figure 8 is an example of this. Such postconditions declaratively specify the effect of a rule on rule node attributes. They are transformed into assignments of attribute values in the exported Henshin rule. This is necessary in all use cases where the model state shall be changed in order to correctly represent the corresponding system state after a rule application.
- *P5 (Returns)*: In addition to the constraints handled by P2, P3 and P4, respectively, in this profile we also export constraints defined over return values. In the exported Henshin rule, such a return constraint is represented by an assignments to a dedicated rule parameter which is declared as *out* parameter (note that all parameters addressed in P2 are *in* parameters).

The integration with Henshin allows *i)* to evaluate and execute extracted contracts, and *ii)* to use the tool more widely in the context of model-based engineering.

3.5 Henshin Control Integration and Deep Oracle

When evaluating the execution of a test case against a model, we have to judge if the behaviour observed is consistent with the model's prescription. Normally this comparison is limited to input-output behaviour observable at the interface of the object or component. Using the Visual Contract Extractor we can implement a *deep oracle* which also detects deviations of the execution's effect on the internal object structure. To this end, we first observe the execution, extract the rule instance and derive the minimal rule as described

in Sect. 2. The minimal rule now provides a concise description of the effect of the execution, which can be compared to the minimal rules of the relevant visual contract. If an equivalent minimal rule is found, we have identified a case with the same effect to the one observed. The corresponding maximal rule of the visual contract is therefore the one to judge the overall correctness of the test execution.

This is done by attempting to apply the selected rule to a model representation of the object state using the same invocation and actual parameters as the test execution. If the application succeeds, i.e., the rule finds an occurrence compatible with the parameter assignment such that all attribute conditions are satisfied, the test case has passed. Otherwise, if either no similar minimal rule can be found matching the effect of the test execution, or the precondition of the corresponding contract rule is not applicable, the test case has failed. Either outcome provides evidence of behaviour implemented, but not specified by the contract.

The application of a contract rule on a model state requires the export of extracted contracts to Henshin and the creation of an initial model state representing the initial object state of the system under test. Once the model state is initialised, it will be updated in synchronisation with the SUT's object state. The application of the model rules to the model states is controlled through a test adapter presenting the API of Henshin as a clone of the SUT's interface.

4 Evaluation

In this section, we discuss the correctness and completeness of extracted contracts, cross-validate them against additional test cases and report on experiments to assess the utility of visual contracts and the scalability of the extraction as implemented by the prototype.

4.1 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 instances of the classes selected for tracing, cf. start of section 2) 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. However, we can expect that whenever both implementation and model preconditions are satisfied, the observable effect of the implementation-level step matches the effect of the model-level step. 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 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 notions of correctness and completeness. The limited amount of code and our familiarity with the application allow 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 [6], 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 11 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_12* 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.

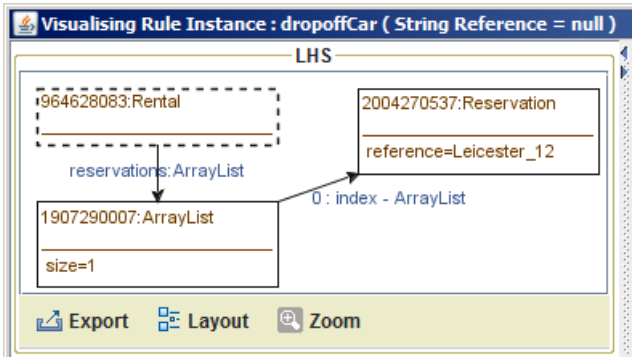
```
1  public void dropoffCar(String Reference){
2
3      int iIndex = getReservationIndex(Reference);
4      if (iIndex==−1){
5          return;
6      }
7
8      Reservation getReservation = this.reservations.get(iIndex);
9      // check if reserved car has been picked up already
10     if (getReservation.pickup!=null){
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

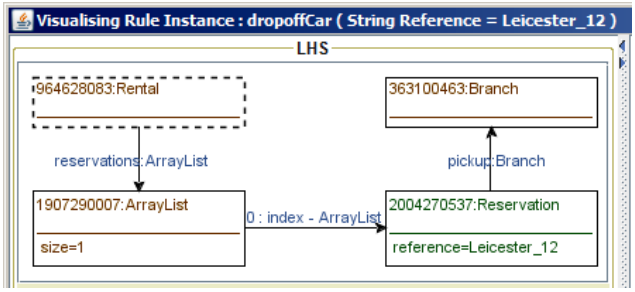
More generally, due to the method of model extraction (and assuming it was correctly implemented in our prototype tool) we can assert that model and implementation should show the same behaviour at least for the test cases used. In particular

- rule 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 rule instances they are extracted from;
- maximal rules subsume all rule instances they derive from, i.e., every rule instance can be replicated as an application of the maximal rule;
- rules with multi objects and multi patterns 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 rule instances their maximal rule originates from.

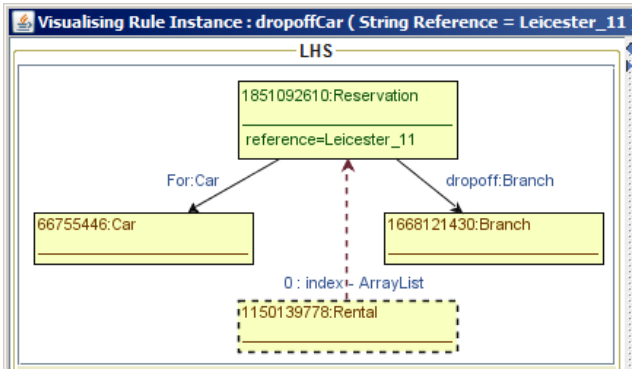
The fact that, in general, models are only representative of the behaviour they were extracted from is an obstacle to some applications, such as their use in automated verification, where contract extraction has to be followed by a manual review and completion. In subsection 4.3 we demonstrate an application to program understanding in the context of testing and debugging that does not rely on completeness or correctness beyond the set of tests executed.



(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. 11: Rule instances for *dropOffCar()*

4.2 Correlation between Model Accuracy and Code Coverage

In addition to the manual validation of correctness and completeness as presented in the previous section, we discuss an experiment to assess quantitatively the correlation between code coverage and accuracy of extracted contracts. Given a test suite and system under test acting as *system oracle*, we

investigate to which extent the *model oracle* created by the extracted contracts (cf. subsection 3.5) imitates the system oracle. Our hypothesis is that *higher code coverage during contract inference implies greater similarity between the behaviours of the system and model oracles*, i.e., higher accuracy of our reverse engineered models. We use our Car Rental case study as the subject for this experiment and present our setup and results in the remainder of this section. Finally, threats to validity are discussed.

Experimental Setup and Results

To assess the accuracy of inferred models is a validation problem familiar in machine learning, where models obtained from training data sets are applied to independent validation data sets. Following a common strategy, we perform a cross-validation using a set of system test cases and using the system itself as a readily available test oracle. When a test case calls a system operation, we have two kinds of expected behaviours: (i) the operation succeeds and updates the system state or (ii) the operation is not possible in the given system state and an exception is thrown. For the cross-validation, we split the set of test cases into training and validation cases. Training cases are used to learn the system model. After executing all training test cases, we extract contracts and export the generalised rules to Henshin. Then, we set up the system state and the corresponding model state and use the validation cases to investigate the model. The idea is, for each test case, to classify the behaviour of the model according to the four options of how the model oracle's behaviour relates to the system oracle's behaviour:

- *True Positive (TP)*: The system operation is possible in the given system state (updating the system state) and the corresponding Henshin rule is applicable on the corresponding model state.
- *False Positive (FP)*: The system operation is not possible in the given system state (an exception is thrown) but the corresponding Henshin rule is applicable on the corresponding model state.
- *True Negative (TN)*: The system operation is not possible in the given system state (an exception is thrown) and the corresponding Henshin rule is not applicable on the corresponding model state.
- *False Negative (FN)*: The system operation is possible in the given system state (updating the system state) but the corresponding Henshin rule is not applicable on the corresponding model state.

Figure 12 illustrates the decision tree for the classification of validation test cases. For each test case, if the test throws an exception, we check if there is a Henshin rule modelling the system operation called by the test which is applicable with the same actual parameters as in the test execution. If so, the test case is classified a *false positive*, and as *true negative* otherwise. If no exception is thrown by the test case, i.e., the system state is successfully updated, we proceed as follows: We generate a rule instance by observing the operation invoked by the test case and check if an equivalent minimal rule

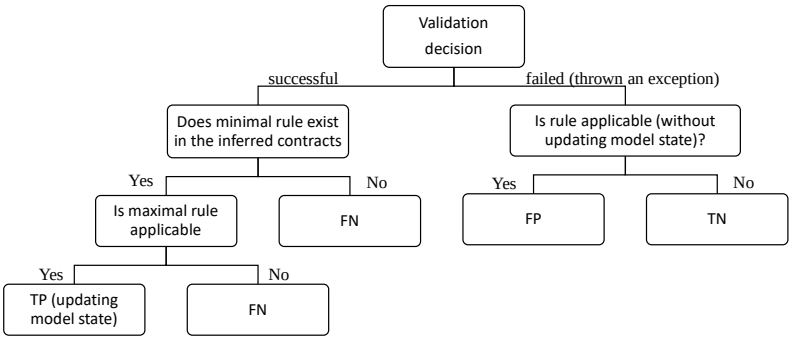


Fig. 12: Decision tree for the classification of validation test cases

exists in the inferred contract. If not, there is no corresponding model behavior and the test case is classified as *false negative*. If there is an equivalent minimal rule, we first identify the maximal rule corresponding to that minimal rule and then use the Henshin rule corresponding to the identified maximal rule to apply it with the same actual parameters as in the test execution. If the Henshin rule is not applicable, the test case is classified as *false negative*. If the rule is applicable, we update the model state by applying the rule and the test case is classified as *true positive*.

To summarize the classification of all validation test cases, we calculate precision (see Formula 1) and recall (see Formula 2) values as usual, thus obtaining a quantitative measure of how well the model oracle imitates the system oracle or, more generally, how accurate the generalized contracts model the behavior of the system. A lack of precision (FPs) could be caused by non-observed object types, missing or inadequate attribute conditions, etc., while low recall values (FNs) could be caused by a lack of coverage of the training test cases.

$$\text{Precision} := \frac{\#TP}{\#TP + \#FP} \tag{1}$$

$$\text{Recall} := \frac{\#TP}{\#TP + \#FN} \tag{2}$$

We performed the described experiment using our car rental case study. In total we defined 120 test cases partitioned into five groups, referred to as A-E in the sequel, where each group represents a concrete scenario of how the system could be used. In contrast to classical k-fold cross-validation, our groups of test cases are not of equal size because of dependencies between test cases in the respective scenarios. We consider four splits: 1 out of 5, 2 out of 5, 3 out of 5, and 4 out of 5. For example, in the case “1 out of 5”, one group of test cases is used as training test cases while those of the four remaining groups are used as validation test cases. Since our test case groups are not equally sized, each of the splits “k out of 5” is run in five rounds in which we

select k training test groups and use the remaining 5-k groups for validation. To perform the experiments as described above, constraints learned by Daikon for the generalised rules have been exported to Henshin using profile P2, i.e., constraints over rule parameters are used to bind the corresponding Henshin rule parameters to rule node attributes such that concrete parameter values can be passed to Henshin rule applications (cf. subsection 3.4).

Our results are summarised by the four tables presented in Figure 13. Each of the tables refers to one of the "k out of 5" splits, starting with the "1 out 5 split" on top, down to the "4 out of 5" at the bottom. For each split, columns 2 to 6 show, for each of the five rounds of a split, which groups of test cases have been selected for training and validation, respectively. In the first round of the "1 out of 5" split, for instance, the test cases of group A have been used for training, while those of the remaining groups have been used for validation. The number of test cases comprised by each of the test case groups is indicated in the respective column headers, e.g., test group A comprises 42 test cases. Precision and recall values are shown for each round in columns 7 and 8, average values aggregated for all five rounds of a split are shown in the bottom line of the respective table. Columns 9 and 10 show the code coverage achieved by the training cases[2]. Line coverage is presented in column 9, while column 10 refers to branch coverage. Again, coverage measures are shown for each of the five rounds of each split, summarised by an average value at the bottom line of each table.

A first observation is that we have a constant precision of 1.0, i.e., no false positives have been observed in our experiment which in turn means that the generalised contracts behave correctly. Since we only exported Daikon constraints over parameters and attributes, we conclude that the context in which a generalised contract rule may be correctly applied according to our test suite is sufficiently determined by its structural precondition, restricted by actual parameter values bound to node attributes. Indeed, reviewing the operations of our Car Rental case study, application conditions are of structural nature only. Taking this into account, the high precision score is not surprising, and may not transfer to operations with complex conditions on attribute values.

A more interesting picture arises for the recall values. Average recall values for each of our four tables in Figure 13 show a strong correlation of recall and branch coverage. We expected branch coverage to be a better indicator for completeness of inferred contracts than line coverage. However, the fact that both metrics show very similar measures was a positive surprise. Since we know that a contract instance represents a path through the program, we would have expected such a correlation for path coverage, which is generally hard to achieve and measure. A qualitative explanation for the strong correlation between completeness and branch coverage can be found by reviewing the source code of our case study. Firstly, most operations are relatively simple, so paths are short. Secondly, there is a small number of loops in the con-

[2] We have used Cobertura, a code coverage utility for Java, to calculate the percentage of the covered code by our tests. It is available at http://cobertura.github.io/cobertura/

	Test group (no. of invocations)							Training-test coverage	
1/5 Spilt	A (42)	B (22)	C (22)	D (18)	E (16)	Precision	Recall	Line coverage	Branch coverage
Round 1	T	V	V	V	V	1	0.32	47%	35%
Round 2	V	T	V	V	V	1	0.47	60%	53%
Round 3	V	V	T	V	V	1	0.71	82%	64%
Round 4	V	V	V	T	V	1	0.71	75%	55%
Round 5	V	V	V	V	T	1	0.55	74%	53%
						1	0.55	67%	52%

	Test group (no. of invocations)							Training-test coverage	
2/5 Spilt	A (42)	B (22)	C (22)	D (18)	E (16)	Precision	Recall	Line coverage	Branch coverage
Round 1	T	T	V	V	V	1	0.4	68%	59%
Round 2	V	T	T	V	V	1	0.7	87%	74%
Round 3	V	V	T	T	V	1	0.9	87%	68%
Round 4	V	V	V	T	T	1	0.75	76%	57%
Round 5	T	V	V	V	T	1	0.53	87%	68%
						1	0.65	81%	65%

	Test group (no. of invocations)							Training-test coverage	
3/5 Spilt	A (42)	B (22)	C (22)	D (18)	E (16)	Precision	Recall	Line coverage	Branch coverage
Round 1	T	T	T	V	V	1	0.62	88%	75%
Round 2	V	T	T	T	V	1	0.83	93%	77%
Round 3	V	V	T	T	T	1	0.78	87%	68%
Round 4	T	V	V	T	T	1	0.82	88%	70%
Round 5	T	T	V	V	T	1	0.55	88%	72%
						1	0.72	88%	72%

	Test group (no. of invocations)							Training-test coverage	
4/5 Spilt	A (42)	B (22)	C (22)	D (18)	E (16)	Precision	Recall	Line coverage	Branch coverage
Round 1	T	T	T	T	V	1	0.75	94%	79%
Round 2	T	T	T	V	T	1	0.5	94%	79%
Round 3	T	T	V	T	T	1	0.71	88%	72%
Round 4	T	V	T	T	T	1	1	94%	79%
Round 5	V	T	T	T	T	1	0.8	93%	77%
						1	0.75	93%	77%

T: Training
V: Validation

Fig. 13: Results of validation test cases, described in four tables

trol structure, i.e., a relaxed notion of path coverage based on a small upper bound of loop executions is not very different from branch coverage. Thirdly, for those loops that have been executed by our tests, the data operations performed within are very schematic, such that extracted contract instances can be generalised to rules with multi-objects or -patterns.

Threats to Validity

Concerning external validity, it is unclear how the results of our experiment will translate to other applications. In particular, the source code of our case study

does not contain complex control structures, as they may occur in more algorithmic software. In the latter case, branch coverage might be an insufficient indicator for measuring the completeness of the (set of) extracted contracts. However, one can argue that this problem pertains to the design of appropriate test suites which in turn are used for learning visual contracts. In any case, we believe that comparable results will be observed for similar applications based on complex and dynamic object structures. Our ability to validate this claim more widely, e.g., on the JHotDraw and NanoXML case studies, is currently limited by the lack of test suites providing a similar level of coverage. This is discussed further in the Conclusion.

Our results could be biased by our test setup which may threaten internal validity. In order to execute both the training and the validation test cases, small system states are initialised, only containing several dozens of runtime objects which are artificially created to be appropriate for all of the tests that will operate on them. Thus, our sample states could be too small to assess precision. However, we confirmed our precision values by inspecting the nature of our system operations whose preconditions are mostly of structural nature. Thus, we are confident that similar precision values will be obtained for larger system states.

Finally, construct validity may be affected by the way we measure the correctness of contracts, i.e., rules modelling successful operation invocations are classified to be correct as long as they are applicable to the corresponding model state when passing the same concrete parameters as the invocation. A stronger notion of correctness would be that the resulting model state corresponds to the altered system state obtained after executing the system operation. However, we achieve the same guarantee indirectly, by selecting a rule from the contract that has the same effect (minimal rule) as observed during the invocation of the operation. Assuming that minimal rules are computed and matched correctly, this is sufficient to show that the resulting model and system states are in correspondence.

4.3 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. The documentation for two of the operations is shown in Figure 14. Tests were executed and results recorded in two different formats: (A) as sequences of invocations and returns of opera-

Specification of operations

`String` **`registerClient`** (`String` `city`, `String` `client`)

Creates new *client object* for client and registers it with the branch at *city*. The attribute *branch.cMax* will be increased for each new client added.

Parameters:
`city` – non-null string value used to get branch object by city name.
`client` – non-null string value used to set client name
Returns:
`String` – if the client is registered successfully with the branch, client id of the form city + "_" + Branch.cMax, *`null`* otherwise.

`String` **`makeReservation`** (`String` `client`, `String` `pickup`, `String` `dropoff`)

Creates new reservation object for a client that must be registered with *pickup* branch. The *pickup* branch must have at least one Car available to be booked. The attribute *branch.rMax* will be increased by 1 for each new reservation.

Parameters:
`client` – non-null string value used to get client object by name.
`pickup` – non-null string value used to get branch object by city name
`dropoff` – non-null string value used to get branch object by city name.
Returns:
`String` – if the reservation object is created successfully, reservation reference of the form city + "_" + Branch.rMax, *`null`* otherwise.

Fig. 14: Rental Car Service operations documented in natural language.

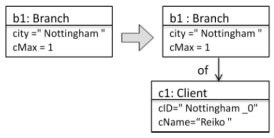
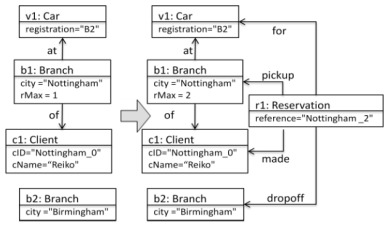
tions from the interface, with queries added to display details of the internal state after each step (see Figure 15 (a)) and (B) as sequences of visual contract instances extracted from the same invocations (see Figure 15 (b)). Both representations are at the same level of abstraction and provide the same information, except for the last column of Figure 15 (b) referring to the lines of code whose execution led to the presence of the objects of the contract. This traceability data is produced and displayed also by the VCE tool.

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 up to 5 invocations each, containing a total of 20 failures to be traced down to the 8 seeded faults. For example, the test case traced in Figure 15 allows to detect 3 failures. In the first step, the client id returned is incorrect (it should be *Nottingham_1*) and the *cMax* attribute is not increased as required in the documentation of the operation in Figure 14. In the second step, the Reservation reference recorded and returned should be *Nottingham_3*.

The 66 participating students were volunteers from an MSc module on (UML-based design, implementation and testing of) Service-oriented Architectures running February-May 2015 at the University of Leicester. We used 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 stu-

		* to show the state after
Step	Operation invocation	Output
1	registerClient("Nottingham", "Reiko")	"Nottingham_0"
*	showClients("Nottingham")	[0]={cName="Reiko", cID="Nottingham_0"}
	showBranch("Nottingham")	{city="Nottingham", rMax=1, cMax=1}
	showCars("Nottingham")	[0] = {Registration="B2"} [1] = {Registration="C3"}
2	makeReservation("Nottingham_0", "Nottingham", "Birmingham")	"Nottingham_2"
*	showClientReservations("Nottingham_0")	[0]={ reference="Nottingham_2" , made=" Nottingham_0" , pickup="Nottingham" , dropoff= "Birmingham", for="B2"}
	showBranch("Nottingham")	{city=" Nottingham", rMax=2, cMax=1}

(a) Textual representation of test report

Step	Operation invocation	Extracted visual contracts	Access in the code *line number*
1	registerClient("Nottingham", "Reiko")	Return : " Nottingham_0" 	b1:Branch - 6, 11, 13 c1:Client - 9, 10, 11, 14
2	makeReservation("Nottingham_0", "Nottingham", "Birmingham")	Return : " Nottingham_2" 	b1: Branch - 21, 24, 39, 40 b2:Branch - 22, 40 c1:Client - 24 v1:Car - 30, 40 r1:Reservation - 40, 47, 48

(b) Visual representation of test report

Fig. 15: An example of a test report used in the study, representing two similar test invocations in different forms: textual and visual.

dents in group B with an average coursework mark of 68.1%. From the module, and specifically the coursework on implementation and testing, students were familiar with the concept of model-based testing of services as well as with visual contracts as a means to specify service interfaces. The participation in the experiment was voluntary. The Car Rental Service interface, its documentation and the two types of assignments were introduced to all students in a joint 50 min session prior to the experiment. This included running through an example of the task in each representation. 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.

As summarised in Table 1, 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 identi-

fying 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[3] 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 effect sizes of 0.452 for recall and 0.561 for precision have been calculated using Cohen's d measure using the same online tool. This measure is commonly used for independent samples and similar standard deviations and populations. Values around 0.5 are considered to indicate medium effects.[3]

Table 1: 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
effect size	0.452	0.561

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;

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

– *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.

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.

4.4 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[4] , 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. However, we are not interested in how NanoXML is designed and implemented in general, but instead focus on two classes, *XMLElement* and *XMLAttribute*, which provide the functionality to manipulate XML documents. In particular, we monitor all *XMLElement* methods, executing 5605 test cases in order to evaluate the handling of large numbers of invocations.

While substantial tests are provided with the software, the majority is not for the two classes of interest. We therefore choose to generate our own test suite using the CodePro test case generator[5]. This creates JUnit templates that we adapted and completed manually to improve coverage. These tests cover 2099 out of 5836 instructions. In Figure 16 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, rule 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.

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 **.Com-*

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

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

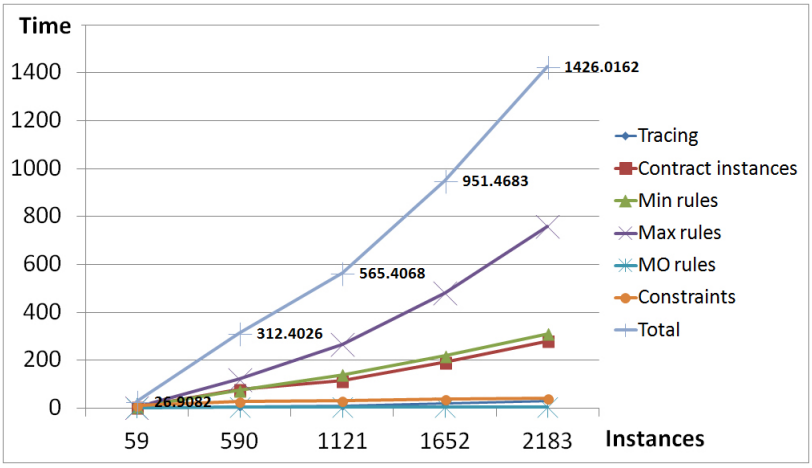


Fig. 16: Scalability for extracting contracts from NanoXML

mandMenu.actionPerformed(comExe). We use GUI testing using WindowTester[6] 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 17, but the quadratic component of maximal rule extraction is less significant due to the smaller overall number of rule instances.

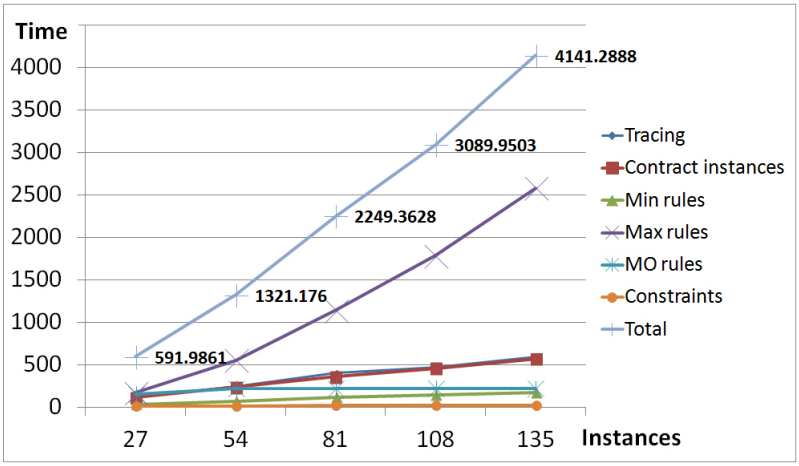


Fig. 17: Scalability for extracting contracts from JHotDraw

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

Unlike NanoXML where the number of invocations / rule instances is large but the size of each rule instance small, JHotDraw produces rule instances up to several hundreds of objects. Table 2 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 2: 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.

A potential threat to the external validity is that we have only considered two systems, both stimulated using test suites focussing on dynamic object structures which might not be representative for testing the full functionality of these systems. While we may get different results for different test suites (and different systems), we do not see serious theats to our key finding on how the extraction of visual contracts scales with (a) an increasing number and (b) an increasing size of rule instances.

4.5 Summary

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 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.

5 Further Applications

So far we have illustrated how our approach and tool can be used to aid program understanding and visualisation of behaviour. Moreover, the integration

with Henshin, turning visual contracts into executable transformation rules, allows us to evaluate and apply contracts more widely, e.g., in the context of model-based testing [24,31].

In this section, we sketch two additional application scenarios. In Sect. 5.1, we discuss the usage of dynamically extracted visual contracts for a new debugging paradigm which can be considered as visual or model-based debugging. Sect. 5.2 proposes an approach to generate complex model editing operations automatically from examples.

5.1 Visual Debugging

Traditional debugging, as supported by an IDE such as Eclipse, allows programmers to interactively inspect and trace the dynamic behaviour of a program. It requires to define, in advance, breakpoints to hold execution at a certain point, allowing to observe and investigate accessed objects, variables and their actual values. To apply this technique for tasks such as localising faults, it needs sufficient precision in identifying breakpoints, which may be an intricate task. Defining many breakpoints or a breakpoint inside a loop is usually not practical, as it may lead to either stopping the execution many times or observing similar details with minor differences at each stop. In the worst case, programmers single-step through instructions and observe changes to the program state.

Extracted visual contract instances can serve as an alternative presentation to support *visual debugging*. The idea is to exploit the debugging interface provided by the Java Virtual Machine to generate program snapshots which can be visually inspected. This raises the level of abstraction from implementation-based debugging to model-based debugging. Accompanying trace information finally helps to localise faults in the source code.

The advantages of debugging at the model-level have been discussed in the literature, e.g. in [37]. Existing approaches mainly focus on debugger frameworks for dedicated domain-specific languages and are not applicable to mainstream Java programs. The monitoring approach presented in [22] share the approach of using models to detect faults in the behaviour of the software, but does not address fault localisation.

The idea of using graphical representations of dynamic object structures for program understanding and debugging has been realised in the eDOBS Eclipse plug-in [21], a Java debugging add-on which allows developers to interactively inspect and visualize (parts of) the current heap of a Java program at run-time as a UML object diagram. However, while this approach raises the level of abstraction from plain programming structures to the level of UML object diagrams, extracting the object-level changes caused, e.g., by a method invocation, remains a tedious task which involves the manual inspection of a potentially large number of object graph snapshots. In contrast, using rule instances we can present the effect of a method invocation in a single visual representation.

5.2 Learning Model Editing Operations

Complex editing operations such as model refactorings are a valuable configuration parameter for many tools in Model-driven Engineering (MDE), e.g. to continuously improve model quality using refactoring tools [7], or to describe the changes between two versions of a model in a meaningful way [28,14]. However, MDE platforms such as the Eclipse Modeling Framework offer only a generic low-level API for model modification. Likewise, editing operations generated from meta-models, e.g. as proposed in [30,29], are still primitive. Complex operations can be implemented manually or by specifying their effect as a model transformation. Both approaches require a deep understanding of the meta-model and its relation to the concrete syntax, thus being only accessible to tool developers and language designers.

Our approach can be used to learn complex editing operations automatically from examples specified by domain experts. An example of a complex editing operation, i.e. the model states before and after a model refactoring, can be specified using standard model editors. Example models are transformed into the graph representation of the VCE tool, generalised to transformation rules and finally exported to Henshin. The exported transformation rules can be integrated as complex editing operations in model editors.

In contrast to previous "model transformation by example" proposals requiring manual processing or augmentation of generated operations at the abstract syntax level [26], our aim is to stick entirely to the concrete syntax notation domain experts are familiar with.

While it shares some of the fundamental technology, this application diverges from the one in this paper in important ways. It does not rely on extraction of contract instances from Java executions, but works on a small, manually produced set of model transformation examples, including negative ones where no transformation should take place. As a consequence, in [27] we have explored alternative solutions to inferring simple attribute conditions, but also addressed the inference of negative application conditions.

6 Related Work

To the best of our knowledge, the work presented in this paper is the first that presents an integrated tool for inferring visual contracts with advanced features such as multi-objects and patterns, attribute conditions, etc. from a test executions. From a broader perspective, related tools can be found in the wide field of reverse engineering. Among them, we particularly have to mention tools for extracting models from implementation artefacts, e.g., for the reverse engineering of UML sequence diagrams [48], activity diagrams [35], entity data models from databases [33], or graph grammars representing sets of nested call graphs [46].

Reverse engineering visual contracts is a process of inferring rules from transformations. This has been suggested in a number of areas, including the

modelling of real-word business processes [13], biochemical reactions [45] and model transformations [15]. 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: the extraction of models from implementations in general and the inference of transformation models in particular.

6.1 Reverse Engineering of Models

Automated reverse engineering is based on static or dynamic analysis. The static approach, exemplified by [40,38,43], 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 [38], 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 [12,49,47], is that the extracted model represents only those behaviours that are actually executed. In particular [47] 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.

Approaches for inferring invariants have been considered in component-based verification [34] and software testing, e.g., to address the test oracle problem [10] or to generate logical test inputs [8]. We share with them the technique of using dynamic invariant detection by Daikon [19].

6.2 Inference of Transformation Models

The generalisation of contract instances is related to the task of semi-automatically learning rules from model transformations [26]. We go beyond this by supporting inference of advanced rule features such as multi-objects and multi-patterns as well as attribute conditions. Moreover, we support a fully automated inference of transformation rules, while the process presented in [26] relies on manual interventions, e.g., to adapt inferred pre- and post-conditions.

[13] 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 rule 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 and -patterns for visual contracts.

In [45] 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 [25], we distinguish *in-place* where source and target have the same metamodel and *out-place* transformations where the metamodels are different [36]. For learning *out-place* transformations, [15] use input-output pairs representing the result of a transformation process rather than a single step. [20, 44, 9] also address the learning of *out-place* transformations, while our approach focusses on *in-place* transformations.

[32] 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.

7 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 -patterns as well as 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.

We also reported on the integration of our tool with the Henshin model transformation tool to simulate extracted contracts. It remains to be seen if visual contracts, that go beyond externally observable behaviour, can improve fault localisation. Our hypothesis is that faults leading to incorrect internal object states could be detected immediately as deviations between the specified and the observed object transformation, rather than later when the incorrect data is used by another operation, thus indicating the faulty invocation more accurately. Another line of enquiry is the use of extracted models for validation and verification using Henshin's suite of analysis tools, which include state space exploration, validation of invariants and detection of conflicts and dependencies between rules.

A current limitation of our approach is its reliance on tests that provide good coverage of the behaviours to be represented as rules. We have found that off-the-shelf test generation approaches are not suitable to create enough of the deep test cases we need to exercise all behaviour (see a related discussion in [41]) and are looking at bespoke integrations supporting a cycle of model-based test generation, test execution, and model extraction in an adaptive testing approach.

Finally, we plan to explore in more detail the applications discussed in section 5, using contract extraction to support testing and debugging as well as the inference of model transformations from examples.

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

References

1. Alshanqiti, A., Heckel, R.: Extracting visual contracts from java programs. In: Intl. Conf. on Automated Software Engineering, pp. 104–114. ACM (2015)
2. Alshanqiti, A.M., Heckel, R.: Towards dynamic reverse engineering visual contracts from java. Electronic Communications of the EASST **67** (2014). URL http://journal.ub.tu-berlin.de/eceasst/article/view/940
3. Alshanqiti, A.M., Heckel, R., Kehrer, T.: Visual contract extractor: a tool for reverse engineering visual contracts using dynamic analysis. In: D. Lo, S. Apel, S. Khurshid (eds.) Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, pp. 816–821. ACM (2016). DOI 10.1145/2970276.2970287. URL http://doi.acm.org/10.1145/2970276.2970287
4. Alshanqiti, A.M., Heckel, R., Khan, T.: Learning minimal and maximal rules from observations of graph transformations. Electronic Communications of the EASST **58** (2013)
5. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: Proc. 13th Intl. Conf. on Model Driven Engineering Languages and Systems, pp. 121–135 (2010). DOI 10.1007/978-3-642-16145-2_9. URL http://dx.doi.org/10.1007/978-3-642-16145-2_9
6. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced concepts and tools for in-place EMF model transformations. In: D.C. Petriu, N. Rouquette, Ø. Haugen (eds.) Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I, *Lecture Notes in Computer Science*, vol. 6394, pp. 121–135. Springer (2010). DOI 10.1007/978-3-642-16145-2_9. URL http://dx.doi.org/10.1007/978-3-642-16145-2_9
7. Arendt, T., Taentzer, G.: A tool environment for quality assurance based on the eclipse modeling framework. Automated Software Engineering **20**(2), 141–184 (2013)
8. Artzi, S., Ernst, M.D., Kieżun, A., Pacheco, C., Perkins, J.H.: Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In: Proc. 1st Workshop on Model-Based Testing and Object-Oriented Systems (2006)
9. Balogh, Z., Varr, D.: Model transformation by example using inductive logic programming. International Journal - Software and Systems Modeling **8**(3), 347–364 (2009)
10. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Trans. Software Eng. **41**(5), 507–525 (2015). DOI 10.1109/TSE.2014.2372785
11. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings: Rule extraction and tool support. ECEASST **16** (2009)
12. Brito, H., Marques-Neto, H., Terra, R., Rocha, H., Valente, M.: On-the-fly extraction of hierarchical object graphs. Journal of the Brazilian Computer Society pp. 1–13 (2012)
13. Bruggink, H.: Towards process mining with graph transformation systems. In: H. Giese, B. Knig (eds.) Graph Transformation, *Lecture Notes in Computer Science*, vol. 8571, pp. 253–268. Springer International Publishing (2014). DOI 10.1007/978-3-319-09108-2_17. URL http://dx.doi.org/10.1007/978-3-319-09108-2_17
14. Bürdek, J., Kehrer, T., Lochau, M., Reuling, D., Kelter, U., Schürr, A.: Reasoning about product-line evolution using complex feature model differences. Automated Software Engineering pp. 1–47 (2015)

15. Dolques, X., Dogui, A., Falleri, J.R., Huchard, M., Nebut, C., Pfister, F.: Easing model transformation learning with automatically aligned examples. In: Proceedings of the 7th European conference on Modelling foundations and applications, ECMFA'11, pp. 189–204. Springer-Verlag, Berlin, Heidelberg (2011)
16. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
17. Engels, G., Lohmann, M., Sauer, S., Heckel, R.: Model-driven monitoring: An application of graph transformation for design by contract. In: A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.) Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings, *Lecture Notes in Computer Science*, vol. 4178, pp. 336–350. Springer (2006). DOI 10.1007/11841883_24. URL http://dx.doi.org/10.1007/11841883_24
18. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(13), 35 – 45 (2007). DOI http://dx.doi.org/10.1016/j.scico.2007.01.015. URL http://www.sciencedirect.com/science/article/pii/S016764230700161X. Special issue on Experimental Software and Toolkits
19. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1-3), 35 – 45 (2007). DOI http://dx.doi.org/10.1016/j.scico.2007.01.015. URL http://www.sciencedirect.com/science/article/pii/S016764230700161X
20. Faunes, M., Sahraoui, H., Boukadoum, M.: Generating model transformation rules from examples using an evolutionary algorithm. In: Proceedings of the 27th IEEE/ACM Intl. Conf. on Automated Software Engineering, pp. 250–253. ACM (2012)
21. Geiger, L., Zündorf, A.: edobs-graphical debugging for eclipse. Electronic Communications of the EASST **1** (2007)
22. Hamann, L., Hofrichter, O., Gogolla, M.: Ocl-based runtime monitoring of applications with protocol state machines. In: Proc. 8th European conference on Modelling foundations and applications, pp. 384–399 (2012). DOI 10.1007/978-3-642-31491-9_29. URL http://dx.doi.org/10.1007/978-3-642-31491-9_29
23. Hausmann, J.H., Heckel, R., Lohmann, M.: Model-based development of web services descriptions enabling a precise matching concept. Int. J. Web Service Res. **2**(2), 67–84 (2005). DOI 10.4018/jwsr.2005040104. URL http://dx.doi.org/10.4018/jwsr.2005040104
24. Heckel, R., Lohmann, M.: Towards contract-based testing of web services. Electronic Notes in Theoretical Computer Science **116**, 145 – 156 (2005). DOI http://dx.doi.org/10.1016/j.entcs.2004.02.073. URL http://www.sciencedirect.com/science/article/pii/S1571066104052831
25. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Conceptual modelling and its theoretical foundations. In: A. Düsterhöft, M. Klettke, K.D. Schewe (eds.) Conceptual Modelling and Its Theoretical Foundations, chap. Model transformation by-example: a survey of the first wave, pp. 197–215. Springer-Verlag, Berlin, Heidelberg (2012)
26. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model transformation by-example: a survey of the first wave. In: Conceptual Modelling and its Theoretical Foundations, pp. 197–215. Springer (2012)
27. Kehrer, T., Alshanqiti, A.M., Heckel, R.: Automatic inference of rule-based specifications of complex in-place model transformations. In: Theory and Practice of Model Transformation - 10th International Conference, ICMT 2017, Held as Part of STAF 2017, Marburg, Germany, July 17-18, 2017, Proceedings, pp. 92–107 (2017). DOI 10.1007/978-3-319-61473-1_7. URL https://doi.org/10.1007/978-3-319-61473-1_7
28. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: 26th Intl. Conf. on Automated Software Engineering, pp. 163–172 (2011)
29. Kehrer, T., Rindt, M., Pietsch, P., Kelter, U.: Generating edit operations for profiled UML models. In: Proc. Intl. Workshop on Models and Evolution, *CEUR Workshop Proceedings*, vol. 1090, pp. 30–39 (2013)

30. Kehrer, T., Taentzer, G., Rindt, M., Kelter, U.: Automatically deriving the specification of model editing operations from meta-models. In: Proc. 9th Intl. Conf. on Model Transformations, pp. 173–188. Springer (2016)
31. Khan, T.A., Runge, O., Heckel, R.: Testing against visual contracts: Model-based coverage. In: Proc. Intl. Conf. on Graph Transformation, pp. 279–293 (2012)
32. Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: L. Tratt, M. Gogolla (eds.) Proceedings of the Third international conference on Theory and practice of model transformations, *Lecture Notes in Computer Science*, vol. 6142, pp. 153–167. Springer Berlin Heidelberg (2010)
33. Malpani, A., Bernstein, P., Melnik, S., Terwilliger, J.: Reverse engineering models from databases to bootstrap application development. In: Proc. 26th Intl. Conf. on Data Engineering, pp. 1177–1180 (2010). DOI 10.1109/ICDE.2010.5447776
34. Mariani, L., Pezzè, M.: A technique for verifying component-based software. Electr. Notes Theor. Comput. Sci. **116**, 17–30 (2005). DOI 10.1016/j.entcs.2004.02.089. URL http://dx.doi.org/10.1016/j.entcs.2004.02.089
35. Martinez, L., Pereira, C., Favre, L.: Recovering activity diagrams from object oriented code: an mda-based approach. In: Proc. Intl. Conf. on Software Engineering Research and Practice, vol. 1, pp. 58–64 (2011)
36. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electron. Notes Theor. Comput. Sci. **152**, 125–142 (2006)
37. Pavletic, D., Voelter, M., Raza, S.A., Kolb, B., Kehrer, T.: Extensible debugger framework for extensible languages. In: Reliable Software Technologies–Ada-Europe 2015, pp. 33–49. Springer (2015)
38. Rountev, A., Volgin, O., Reddoch, M.: Static control-flow analysis for reverse engineering of uml sequence diagrams. SIGSOFT Softw. Eng. Notes **31**(1), 96–102 (2005)
39. Runge, O., Khan, T.A., Heckel, R.: Test case generation using visual contracts. ECEASST **58** (2013)
40. Sarkar, M.K., Chatterjee, T., Mukherjee, D.: 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)
41. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (T). In: M.B. Cohen, L. Grunske, M. Whalen (eds.) 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015, pp. 201–211. IEEE Computer Society (2015). DOI 10.1109/ASE.2015.86. URL https://doi.org/10.1109/ASE.2015.86
42. Strüber, D., Born, K., Gill, K.D., Groner, R., Kehrer, T., Ohrndorf, M., Tichy, M.: Henshin: A usability-focused framework for emf model transformation development. ICGT (2017)
43. Tonella, P., Potrich, A.: Reverse engineering of the interaction diagrams from c++ code. In: Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on, pp. 159–168 (2003)
44. Varró, D.: Model transformation by example. In: Intl. Conf. on Model Driven Engineering Languages and Systems, pp. 410–424. Springer (2006)
45. You, C.h., Holder, L.B., Cook, D.J.: Learning patterns in the dynamics of biological networks. In: Intl. Conf. on Knowledge Discovery and Data Mining, pp. 977–986. ACM (2009)
46. Zhao, C., Kong, J., Zhang, K.: Program behavior discovery and verification: A graph grammar approach. IEEE Trans. Software Eng. **36**(3), 431–448 (2010)
47. Zhao, C., Kong, J., Zhang, K.: Program behavior discovery and verification: A graph grammar approach. IEEE Transactions on Software Engineering **36**(3), 431–448 (2010)
48. Ziadi, T., Da Silva, M.A.A., Hillah, L.M., Ziane, M.: A fully dynamic approach to the reverse engineering of UML sequence diagrams. In: Proc. Intl. Conf. on Engineering of Complex Computer Systems, pp. 107–116 (2011)
49. Ziadi, T., Da Silva, M.A.A., Hillah, L.M., Ziane, M.: 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. IEEE (2011)