Inferring Visual Contracts from Java Applications

Thesis submitted for the degree of
Doctor of Philosophy
at the University of Leicester

By

Abdullah Mahfodh Alshanqiti

Department of Informatics
University of Leicester

April 2017

Inferring Visual Contracts from Java Applications

Abdullah Mahfodh Alshanqiti

Abstract

Visual contracts model the operations of components or services by pre- and post-conditions formalised as graph transformation rules. They provide a precise intuitive notation to support testing, understanding and analysis of software. However, creating a detailed model of a system in any language is error-prone.

Visual contracts are no exception, and their specification of object states and transformations requires a deeper understanding of a system than models of externally visible behaviour. This limits their applicability in testing, verification and program understanding, thus inventing an effective technique for extracting visual contracts automatically would enable their wider use in general.

In this thesis we study a reverse engineering approach to address such problems by extracting visual contracts dynamically from existing systems. We propose an inference solution and implement a prototype tool in Java with empirical evaluations of the performance, completeness, correctness and utilisations.

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 allow generalisation by universally quantified (multi) objects. They support program understanding in general, and the analysis of tests based on a concise, visual and comprehensive representation of operations' behaviour in particular.

Acknowledgements

In the name of Allah, the most merciful, the most beneficent

- *Above all, I am so grateful to Allah Almighty for giving me the patience and the capability to accomplish this thesis.*

- *I give heartfelt thanks to my supervisor Prof. Reiko Heckel. Without his incredible generosity and continuous support, this thesis would not have even been completed. I still remember the first few emails I received from him regarding my MSc project in 2010; since then, 7 years have passed, and I have learned so many things from him both academically and personally. His criticism and encouragement have always been important and meant a lot to me, because, he always converts my mistakes and pressure into lessons and strengths. Advisedly, he was involved in every aspect of this thesis and he was always sharp to spot inconsistencies in my writing that I overlooked. No words I could use to express my feeling, but I wish him a wonderful and awesome life ahead with no tensions and worries.*

- *I would particularly like to thank my second supervisor Dr. Neil Walkinshaw for his great suggestions and so kind help in several occasions.*

- *I am grateful to my thesis examiners, Dr. Emilio Tuosto and Dr. Jens Krinke, for their insightful reviews and a thoroughly inspirational viva.*

- *Special recognition should be given to Prof. Arend Rensink and Prof. Daniel Varro whom I met in GTVMT 2014 for their help and nice suggestions. Also, special thanks should go to Dr. Tamim Khan for his help and cooperation during the preliminary stages of my research journey.*

- *I sincerely acknowledge and offer my profound gratitude to the Islamic University in Madinah Almunawarah and to the Ministry of Education in Saudi Arabia for offering me fully funded scholarship and also for their continued generosity. I would like to extend my sincerest thanks and appreciation to the Department of Informatics of the University of Leicester for offering me the chance to work as a teaching assistant, and also for supporting me to attend several international conferences.*

- *I have had the honorable opportunity to know many people in the UK whom I am glad to call my true friends. My life as a PhD researcher would not have been as enjoyable without these close friends: Dr. Abdullah Alhejaili, Dr. Mohammed Alabdullatif, Dr. Ayman Bajnaid, Dr. Samir Alrahili, Eng. Ahmad Alaskah, Eng. Abdusamad Alzubali, Osama Alahmadi, Samir Alharbi, Musaab Alharbi and many others. Their friendships are really memorable as everyone has been extremely sympathetic and supportive to bring out the best in me. I will certainly miss all of them.*

- *My great appreciation goes to my beloved parents for their wholehearted fondness, deep affection, and endurance. Their dedicated self-sacrifice and being always selfless in giving me the best of everything make me ever speechless and teary eyed when just thinking of them. I would like to take this blessed moment to also express my never-ending love and deep gratitude towards my siblings for their prayers for my success. Sincerest thanks to all of you.*

- *I owe my deepest gratitude to my wife Aeshah for her encouragement, emotional and continuous support while the thesis was being written. Sometimes, when things get complicated, it makes a person irritable, even when it comes to simple things in life. She has been most helpful in supporting me with this aspect. A big thank-you especially to my beautiful daughter Abrar that just watching her smile makes me realise how beautiful my life is. I have to admit this thesis would not have been possible without the love of my family.*

Abdullah Alshanqiti

Table of Contents

List of Tables

List of Figures

Abbreviations

ADM	**A**rchitecture **D**riven **M**odernization
AOL	**A**spect **O**riented **L**anguage
API	**A**pplication **P**rogram **I**nterface
DbC	**D**esign **b**y **C**ontract
DPO	**D**ouble **P**ush **O**ut
DSL	**D**omain **S**pecific **L**anguage
EMF	**E**clipse **M**odeling **F**ramework
GT	**G**raph **T**ransformation
GTS	**G**raph **T**ransformation **S**ystems
JML	**J**ava **M**odelling **L**anguage
MBE	**M**odel **B**ased **E**ngineering
MDE	**M**odel **D**riven **E**ngineering
MDD	**M**odel **D**riven **D**evelopment
MDA	**M**odel **D**riven **A**rchitecture
MDRE	**M**odel **D**riven **R**everse **E**ngineering
MDM	**M**odel **D**riven **M**onitoring
MO	**M**ulti **O**bjects
MTBE	**M**odel **T**ransformation **b**y **E**xample
NAC	**N**egative **A**pplication **C**onditions
OMG	**O**bject **M**anagement **G**roup
OOP	**O**bject **O**riented **p**rogramming
PIM	**P**latform **I**ndependent **M**odel
PSM	**P**latform **S**pecific **M**odel

RE	**R**everse **E**ngineering
SUT	**S**ystem **U**nder **T**est
UML	**U**nified **M**odelling **L**anguage
VC	**V**isual **C**ontract

Part I

Introduction and Background

Chapter 1

Introduction

Much work on software modelling aims at facilitating software lifecycle through the use of models as main artifacts (Ramos, Ferreira, and Barcelo 2012; Sommerville 2011). This helps to make development, analysis, and testing requirements robust and more reliable. Modelling abstracts representation of systems from details, allowing to focus on fundamental parts to precisely understand and analyse functionality of software systems.

In the field of software engineering, models can be used to develop both existing systems or proposed systems to be constructed. In the case of developing an existing system, models can describe how the system works based on extracting knowledge or design information using a reverse engineering approach (Siegel 2014). For proposed systems, models are used to specify the requirements following a forward engineering approach to design and implement software. Here, implementations can be generated from models using so-called model-driven development (MDD) (Whittle, Hutchinson, and Rouncefield 2014).

The prevalence of many different types of software systems, ranging from small embedded systems to large, complex and integrated systems, results in the need to use different types of models. To this end, many types and levels of

models have been suggested in which a system can be represented from various perspectives. The main perspectives followed to model software systems are listed below (Sommerville 2011):

- an external perspective for modelling the environment of a system.
- an interaction perspective for modelling communications between components of a system, or between systems and their environments.
- a structural perspective for modelling the organisation or the structure of a system. A UML class diagram, for example, is widely used to model the structure of classes.
- a behavioural perspective for modelling dynamic activities and interaction in the system. For example, UML state machine, sequence diagrams and object diagrams are popular kinds of models used for this purpose. The latter represents a complete or partial behaviour of the classes in the class diagram.
- an architecture perspective for modelling the overall framework of a system. It describes both structural and behavioural models at a high level using, e.g., UML package diagram.

The approach considered in this thesis focuses on design information extracted from existing systems, handled by reverse engineering structural (classes) and behavioural (objects) models. We define what type of model we want to extract in Section 1.1 and the main motivations behind it in Section 1.2.

1.1 Visual Contracts

Visual contracts 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 are based on graph transformation rules, as the specification of a system can be given by a type graph and a set of rules, where the system's behaviour is represented by a transformation relation labelled by rule names (cf. Section 1.3.2).

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 Object Constraint Language (OCL). They are "visual", using UML notation to model complex patterns and transformations intuitively and concisely, see the example in Figure 1.1. Their executable semantics, based on graph transformation, supports model-based oracle and test case generation (Khan, Runge, and Heckel 2012a; Runge, Khan, and Heckel 2013), run-time monitoring (Engels et al. 2006a), service specification and matching (Hausmann, Heckel, and Lohmann 2005), state space analysis and verification. In Section 2.2, we give more details for defining what is a contract in software modelling, and discusses what are the visual contracts we consider in this thesis.

```
1  public class Rental implements IRental{
2      ...
3      public String registerClient(
4                      String city,
5                      String clientName) {
6          Branch b = getBranch(city);
7          if (b !=null){
8              Client c = new Client();
9              c.name =clientName;
10             c.iD = b.city + (b.cMax++);
11             b.of.add(c);
12             return c.iD;
13         } return null;
14     } ...
15 }
```

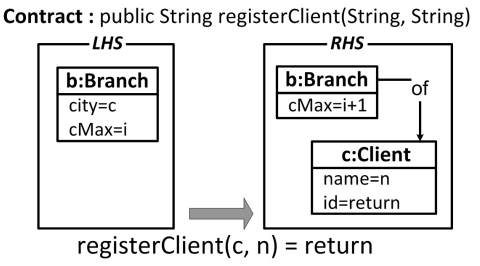


FIGURE 1.1: An example of a visual contract on the right that models a Java method *registerClient(..)*

1.2 Motivation

Reverse Engineering is "a process of discovering the technological principles of a device, object, or system through analysis of its structure, function, and operation" (Chikofsky and Cross 1990). It is required with undocumented systems, or systems that have insufficient or out of date documentation. It provides an abstract version of how a system works, which can then be used in, e.g., testing or debugging activities. In Section 2.3, we expand the discussion about reverse engineering techniques, focusing on models and covering topics such as existing analysis approaches and their challenges.

1.2.1 Reverse Engineering Visual Contracts

Reverse engineering visual contract is the process of extracting a formal model of a system, based on graph transformation concepts. Our ultimate motivation behind this came from the following common challenges:

- The lack of up-to-date models for existing software (Kipyegen, Korir, and Njoro 2013). This could lead to erroneous understanding of required specifications, affecting maintenance and/or testing activities. For example, when a system is changed, all the relevant models need to be updated to reflect such changes. The challenge is that updating any detailed (behavioural) models can be a tedious and error-prone process. Developers might not have sufficient support to correctly modify models nor have enough time due to tight deadlines.
- The difficulty of creating a detailed model of the behaviour of a complex application. Designing system specifications manually, particularly for describing behaviours, is a time-consuming, error-prone and evidently not an easy task.

Visual contracts are no exception, and their specification of object states and transformations require a deeper understanding of a system than models of externally visible behaviour such as state machines or sequence diagrams. This limits their applicability in testing, verification and program understanding in general.

The mentioned challenges can be addressed by reverse engineering models from existing systems. Consequently, our objectives are:

1 to study these challenges by proposing a dynamic approach for extracting contracts automatically from implementations. This also involves addressing the problem of having an out-of-date contract model or allowing to generate new contracts for unmodelled systems.

2 to investigate model generalisation (e.g., extracting shared behaviours and combining them into higher level versions) for supporting program comprehension from two aspects: (a) object behaviours and their relations based on graph pattern matching and (b) object attributes using an invariant detector technique.

3 to systematically evaluate the requirements of such proposal, including the utility, scalability, correctness and completeness.

In addition to the common advantages of reverse engineering models such as increasing understandability, discussed in Chapter 2, the extracted visual contracts can be exploited to bridge an important gap in model driven engineering (MDE). In Section 2.4, we explain that our approach can support the transformation from implementation layer to Platform Specific Model (PSM) layer.

The generated contracts by our approach can automatically give an accurate description of behaviour of operations at a high level of abstraction. They convey information more effectively than code level analysis. They also support program

understanding in general, and the analysis of tests based on a concise, visual and comprehensive representation of operations' behaviour in particular.

1.2.2 Problem Statement

Regardless of the strength of visual contracts in specifying object states and transformations, their creation is complex, error-prone and time-consuming. This limits their applicability in general. A recent supporting tool proposed by (Amálio and Glodt 2015) for visual modelling including contract diagrams (i.e., model operations by pre- and post- conditions) based on Z (Spivey and Abrial 1992) has been evaluated. The major limitation found in this tool is the usability in diagram editing tasks.

The reverse engineering of visual contracts can enable their wider use in testing and verification, and provide a valuable tool for program understanding. Thus, we seek to study the extraction of candidates of visual contracts from implementations as a bottom-up approach. The two main requirements for this extraction are:

- The extracted contracts should faithfully describe the behaviours of the system.
- The extraction process should consume less time and effort than designing contract manually.

In Section 1.3.4, we give more deep explanations about the challenges of extraction of visual contracts. We propose a dynamic solution, based on tracing and model inference, implemented in a prototype tool. The empirical evaluations presented in Chapter 6 assess the performance, completeness, and correctness of visual contracts, as well as their utilisations in testing.

1.3 Overview of the Approach

The approach considered in this thesis aims at extracting candidates of visual contracts from implementations of sequential Java applications. These contracts represent the pre- and postconditions of operations of classes or components by means of graph transformation rules.

1.3.1 Requirements Analysis

Model extraction from an existing system can be performed by different reverse engineering approaches, discussed in Section 2.3.3. These approaches have different pros and cons and no single analytical approach can support all desired features. For example, the static approach can support analysing incomplete systems, e.g., components that cannot be executed independently (Rountev, Volgin, and Reddoch 2005), while the dynamic approach allows to detect dynamic object-oriented behaviours (Canfora, Di Penta, and Cerulo 2011).

Moreover, in the static approach, it is possible to generate a complete model by analysing source code without having to execute the system, but it is uncertain to be correct as many *false-positive* behaviour can be produced in the model (Ashish and Aghav 2013; Pistoia and Tripp 2014). In contrast, the dynamic approach generates an incomplete model but is potentially more correct as its analysis is based on the actual execution of the system (Cornelissen et al. 2009).

In order to extract visual contracts from implementations that describe operation behaviours, we need to:

- determine classes and methods using static approach and
- the changes of object/data states at runtime.

These two requirements show that both static and dynamic approaches are required in order to extract visual contracts. Our approach introduced in Part II is semi-automatic. It initially starts with extracting class diagram from code, allowing to manually select classes and methods of interest to be executed by test cases. The rest is automatic based on dynamic analysis. The focus in this thesis is on the latter step as extracting classes statically is achievable by many available tools.

One critical difference between static and dynamic approaches that led us to adopt the dynamic is its capability in observing method binding (i.e., overloaded and overridden methods), which occurs at compiling/runtime only. In Section 3.2, we explain by example that it is impossible to determine some behaviours using static analysis only.

1.3.2 Visual Contracts as GT Rules

In this section, we introduce basic concepts of graph transformation, including rules and transformations with attributes and multi objects. These concepts serve as the formal basis for our proposal in Part II. We follow the spirit of the double-pushout (DPO) approach (Ehrig et al. 2006) while adopting a set-based presentation.

1.3.2.1 Typed Attributed Graph and Graph Morphism

- A *graph* $G = (V, E, s, t)$ consists of a set V of nodes (or vertices), a set E of edges, and source, target functions $s, t : E \to V$.
- An *attributed graph* $G = (G_V, G_E, G_D, src^G, tar^G)$ consists of a set G_V of nodes (or vertices) with a distinguished set of *data nodes* $G_D \subseteq G_V$, a set G_E of edges, and source, target functions $src^G, tar^G : G_E \to G_V$. We

assume that data nodes represent values and do not have outgoing edges. When clear from the context we will drop indices and superscripts.

- A *graph morphism* $f : G \to H$ is a pair of mappings $f_V : G_V \to H_V, f_E : G_E \to H_E$ preserving sources and targets, and respecting the distinction between object and data: for all $v \in G_V$, $v \in G_D$ iff $f_V(v) \in H_D$.
- A *type graph* TG is a distinguished graph introducing vertex, edge and data types.
- An *instance graph* over a type graph TG is a graph G with a graph morphism $type_G : G \to TG$. *Typed graph morphisms* are morphisms preserving the typing. Instance graphs and morphisms over TG form the category $Graph_{TG}$.

A UML class diagram is formally represented as an *attributed type graph* TG, i.e., a distinguished graph defining vertex, edge, attribute and data types from which object graphs can be constructed. An *instance graph* over TG is a graph G (i.e., represents UML Object Diagram) equipped with a graph morphism $G \to TG$ assigning every element in G its type in TG.

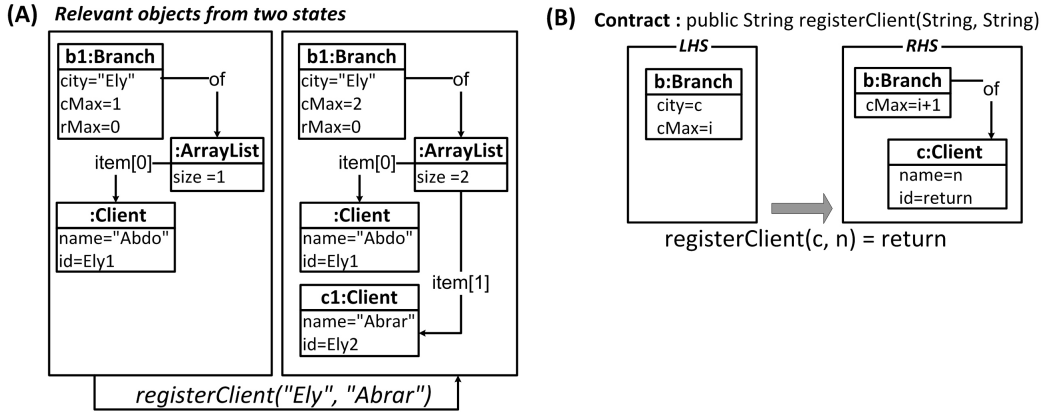


FIGURE 1.2: An example of graph transformation rule

1.3.2.2 Rules and Graph Transformation System

Figure 1.2 (B) shows a transformation rule which, when applied to the object graph in the left of (A) at the match mapping b to $b1$, produces a graph similar to the one in the right of (A), taking into account that of represents an object container (ArrayList). In Part II we give more examples of transformation rules. Formally:

- A rule is a pair $L \xleftarrow{l} K \xrightarrow{r} R$ of injective graph morphisms. A *rule* $p = (L \Rightarrow R)$ defines *left-* and *right-hand side* graphs such that their union $L \cup R$ (and hence their intersection $L \cap R$) is well-defined. In this thesis we assume that $K = L \cap R$ and l, r are inclusions. The set of rules over TG is $Rule_{TG}$.

  Given an instance graph G, rule p can be applied if there is an injective morphism $m : L \to G$ satisfying the dangling condition. That means L is isomorphic to $m(L)$ and, removing $m(L \setminus R)$ from G, the resulting structure is still a graph, i.e., there are no dangling edges. The derived graph H is obtained from G by deleting $m(L \setminus R)$ and adding a copy of $R \setminus L$, denoted $G \overset{p,m}{\Longrightarrow} H$. The construction of this is a double-pushout diagram like in Figure 1.3.

- A graph transformation system (GTS) (P, π) consists of a set of rule names P and a function π assigning each name p a rule $\pi(p) = L \xleftarrow{l} K \xrightarrow{r} R$. The resulting transformation relation is denoted by $G \overset{p}{\Longrightarrow} H \subseteq Graph_{TG} \times Graph_{TG}$.

Extracting visual contract is concerned with deriving specifications from existing systems. As explained, this needs dynamic analysis technique that allow to observe changes of object states. In the case of typed graph transformation, the specification is given by a type graph and a set of rules and the system's

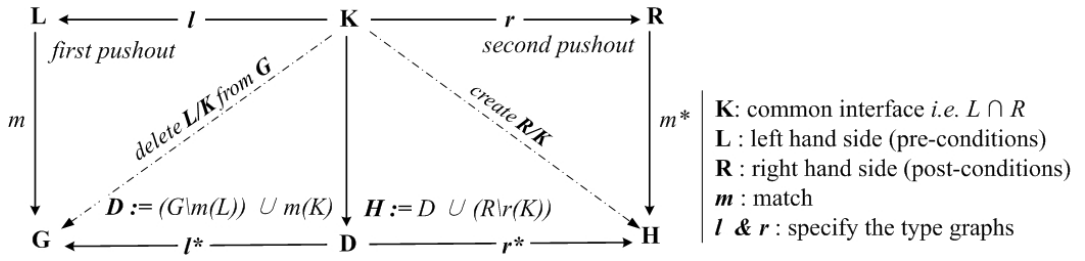


FIGURE 1.3: DPO diagram representing a rule application

behaviour can be represented abstractly by a transformation relation labelled by rule names. In this setting, dynamic reverse engineering visual contracts means to infer rules from observed changes of object states.

In Chapter 3, we present our solution for constructing rules by tracing object states dynamically, while in Chapter 4 we generalise such rules by inference of higher level features.

1.3.2.3 Rules with Multi Objects

Multi objects (MOs) support universal quantification over unknown contexts in a rule. An example of a rule with multi object can be seen in Figure 1.4 where node *Reservation* is an MO node (shown with a 3D shadow) with cardinality (one-to-many) applicable to object graphs with at least one *Reservation* node connected to the *Client*. Formally: A *multi-object (MO) rule* $mr = (L \Rightarrow R, M, card)$ is a rule with a set $M \subseteq (L_V \setminus L_D)$ of MO nodes and cardinality constraints $card : M \to (\mathbb{N} \cup \{*\}) \times (\mathbb{N} \cup \{*\})$. It states how many concrete objects each MO can be instantiated by. The set of MO rules over TG is $MRule_{TG}$.

To derive MO rules from regular rules we have to discover sets of nodes that have the same structure and behaviour, then represent them by a single multi-object node. Chapter 5 discusses our proposed algorithm to do so and then

inferring MO rules. This adds more concise constraints to specify actions across sets of objects of different cardinalities.

```
1   public void cancelClientReservation(
2                               String clientID) {
3
4       for (int i=this.reservations.size()−1;
5           i>=0; i−−) {
6
7           if (!this.reservations.get(i)
8               .made.id==clientID) {
9               continue;
10          }
11          this.reservations.remove(i);
12      }
13  ...
```

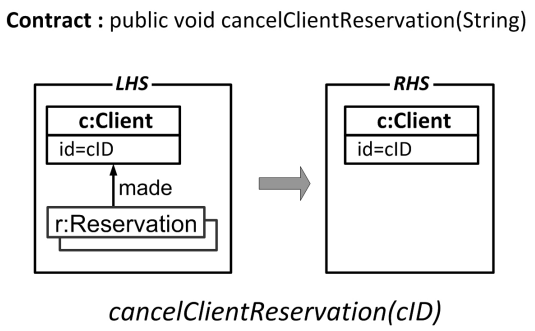


FIGURE 1.4: Representing multi-object for *cancelClientReservation(..)*

1.3.2.4 Graph Transformation Specification

A *graph transformation specification (GTSpec)* $\mathcal{G} = (TG, OP, P)$ consists of a type graph TG, a set of operation names OP, and a set of *parametrised rules* $op(x_1, \ldots, x_n) = y : L \Rightarrow R$. Here MO rule $(L \Rightarrow R, M, card) \in MRule_{TG}$ is labelled by operation $op \in OP$ and augmented by formal parameters $x_1, \ldots, x_n \in L_V \setminus M$ and return $y \in R_V \setminus M$. The transformation obtained by applying rule $op(x_1, \ldots, x_n) = y : L \Rightarrow R$ at match m to an instance graph G is denoted $op(a_1, \ldots, a_n) = b : G \Rightarrow H$ if $m(x_i) = a_i$ and $m^*(y) = b$, briefly $G \overset{op(\overline{a})=b}{\Longrightarrow} H$ where $\overline{a} = a_1, \ldots, a_n$.

1.3.3 Proposed Methodology

This section provides an overview of the proposed approach for reverse engineering VCs from existing systems. The approach is divided into three main steps, illustrated in Figure 1.5 and presented in detail in the mentioned chapters. The first step (A) focuses on extracting actual behaviour by tracing, yielding instance-level versions of visual contracts (rule instances). We adopt a dynamic

analysis approach to construct contract instances by observing system's execution at runtime, introduced in Chapter 3.

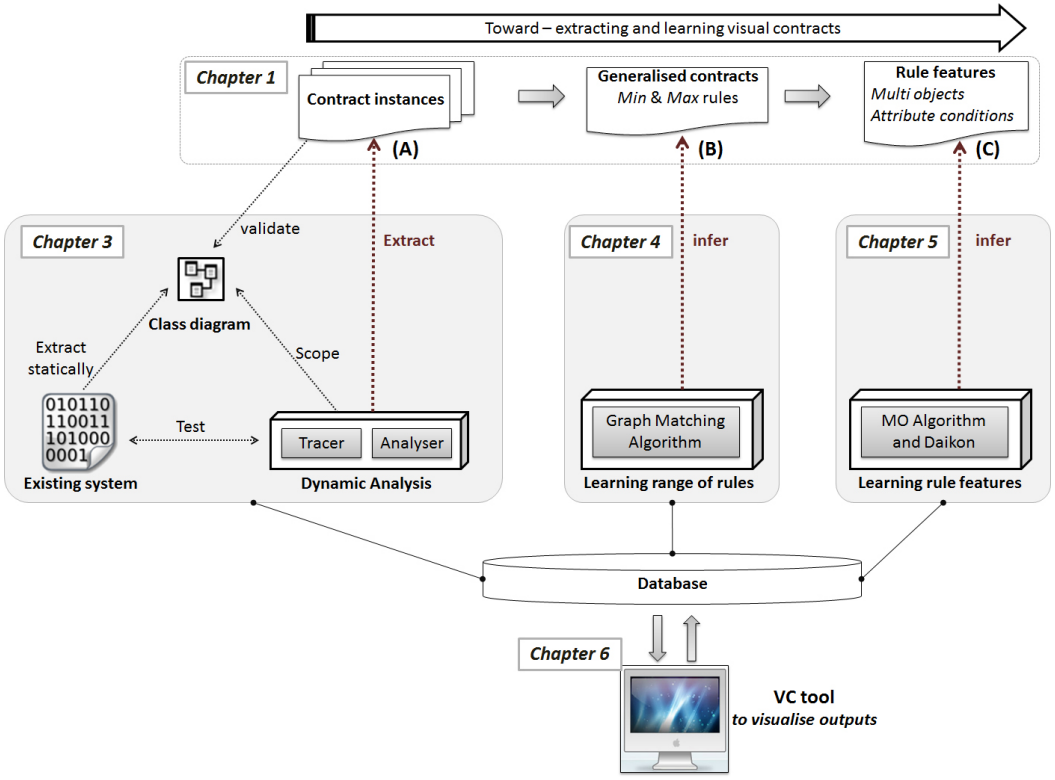


FIGURE 1.5: Overall picture of the thesis

The extracted contract instances describe partial behaviour of the system or component. Generalising them to a model comprehensive version is the second step, see Figure 1.5 (B). At this step, our solution in Chapter 4 depends on typed attributed graph transformation rules and uses directed sub-graph matching algorithms. In the third step, shown in (C) of Figure 1.5 and introduced in Chapter 5, we seek to increase generality of contracts by inferring multi objects and attribute conditions based on data. In the following subsections, we give more details about these three steps.

1.3.3.1 Extraction of Contract Instances

The model illustrated in Figure 1.1 represents a contract instance, reflecting a specific behaviour of operation *registerClient()*. To extract such a contract instance from the implementation, we propose a novel approach based on dynamic reverse engineering, see (Alshanqiti and Heckel 2014). We use instrumentation based on Aspect Oriented Programing (AOP) to observe executions of existing test cases, record these observations in traces and analyse them to filter out irrelevant objects based on their classes and aggregate their basic steps into rule instances covering the overall precondition and effect for that execution.

Along with each constructed instance we collect traceability data for each element of the contract, such as the access type (read/write) with line numbers in the code. These traceability data can be used for graph matching in the next stage and also for program understanding, e.g., as part of testing or debugging.

1.3.3.2 Generalisation of Contract Instances

The idea of *generalisation* as far as this thesis is concerned is to infer concise and comprehensive contracts (rules) from instances to increase understandability. This is because each contract instance only represents one possible outcome from any executed operation. The aim is to combine them to a general behaviour model.

We propose a graph matching algorithm to extract *minimal* and *maximal* rules from pairs of graphs representing transformations, see (Alshanqiti, Heckel, and Khan 2013). Using extracted contract instances as an input set to this algorithm, our approach allows to generate for each contract instance a *minimal rule*, i.e., the smallest rule able to perform the given transformation (Bisztray, Heckel, and Ehrig 2009). This results in a further partitioning of the input set into instances with isomorphic minimal rules. In the *maximal rule*, we match

the rule instances within each partition and identify their common context elements. Therefore, any constructed maximal rules would contain all the context that is present in all instances.

1.3.3.3 Inference of Advanced Rule Features

The aim of this inference is to discover advanced graph transformation features on generalised rules, which allow to raise the level of specification. For example, a multi-object node represents a set of nodes in an instance graph the rule is applied to and carries a cardinality constraint for that set. We propose a novel approach (Alshanqiti and Heckel 2015) to infer rules with multi objects from regular rules. It allows to represent regular rules by equivalent rules with smaller number of elements, i.e., by representing any set of identical nodes by a single multi-object node.

Furthermore, we allow to enhance generalised rules by including conditions on attributes and parameters. Each extracted contract instance includes actual data types and values for node attributes, passed parameters and return. By considering a list of such data, extracted from all rule instances that belong to a specific generalised rule, we can then use them as an input or training-set to a machine learning tool to discover more useful constraints to be added on generalised rules. For this purpose, we adopt an invariant detector tool *Daikon* (Ernst et al. 2007) that is perfectly adequate and supports the main needed features.

1.3.4 Technical Challenges

Reverse engineering visual contracts can be identified by: (1) tracing changes to the data state of the component or service under investigation and their representation by contract instances and (2) the generalisation of these contract

instances. The latter is a major challenge, which requires solving the problem of finding and matching relevant independent contract instances that are extracted from different traces. For example, object identifiers cannot be used as anchors for matching object graphs extracted from different traces.

The construction of contract instances requires a strategy to observe and record efficiently the changes of object structures. Additionally, there are potentiality large numbers of traces to be processed.

Furthermore, each constructed contract represents only a partial behaviour of the system, i.e., the model obtained is valid for the scenario that was executed. Therefore, it is important to find out how to extract comprehensive models that describe general system behaviours. This is considered the most difficult part in the thesis.

For a deeper discussion we address the challenges in the order of the process, as shown in Figure 1.5. The technical problems mostly concentrate on graph pattern matching, as sub-graph isomorphism is known to be NP-complete (Conte et al. 2004):

Scalability. Given an interface and a set of test cases, when analysing the execution of one of its operations, there could be a large number of object instances, including low level objects (e.g., from Java.util.*). Hence, how to define the scope of tracing this operation? How to distinguish between relevant and irrelevant objects in a scalable way to model their behaviours as a rule?

Subgraph Isomorphism. If we are able to identify the relevant objects and represent them as visual contracts (pre and post-conditions), how to generalise them into higher level rules to obtain a complete behavioural model? As mentioned, we rely on graph matching to generalise rule instances. It is known that finding common subgraph isomorphisms has an exponential time complexity

and could produce a large number of matches (Dahm et al. 2015). How is our approach affected by these problems? Are we losing behaviours, affecting the quality of the results?

Correctness. With respect to the implementation, correctness means that models must behave exactly as the implementation. This implies that models should not include any invalid behaviours, i.e., not allowing behaviour that is not implemented. How to evaluate correctness of the extracted VCs? assuming that the implementation represents the truth.

Completeness. Completeness of models means that the implementation of all the system's behaviour is captured by the extracted contracts. How to evaluate the completeness of extracted VCs to ensure that there are no missing behaviours?

Generally, the extraction of a complete and correct model is a major challenge (Canfora, Di Penta, and Cerulo 2011; Cornelissen et al. 2009) and still an open problem. In our approach, such extraction is not feasible and cannot be proved conclusively. However, we propose a scalable solution as discussed in Part II, allowing to address completeness. We use manual inspection to evaluate the correctness of extracted contracts as discussed in Chapter 6.

1.4 Contributions and Related Publications

The contributions of this thesis focus on the process of extracting and generalising visual contracts from implementations of sequential Java applications, represented by graph transformation rules. We study techniques related to dynamic reverse engineering for extraction and graph pattern matching for generalisation. The papers (Alshanqiti and Heckel 2014; Alshanqiti and Heckel

2015; Alshanqiti, Heckel, and Kehrer 2016; Alshanqiti, Heckel, and Khan 2013) contributed to this thesis address different steps of the process, summarised in this section as follows:

- Instance of visual contracts: we propose a novel approach based on dynamic reverse engineering to extract visual contract instances from existing Java applications.
- Minimal and maximal graph transformation rules: we propose a novel approach to generalise extracted contracts by inferring a range of graph transformation rules as interface specifications for operations.
- Rules with multi objects: we propose a novel approach to raise the level of abstraction of generalised contracts by inferring rules with universally quantified multi objects.
- Rule conditions on attributes and parameters: we implement a learning approach to enhance contract descriptions by inferring rule conditions on parameters and attributes.
- Evaluation: we assess completeness and correctness of extracted visual contracts using a case study discussed in Section 3.2. We use two other case studies (*NanoXML - a small non-validating XML parser for Java* 2016) and (*JHotDraw as Open-Source Project by Java* 2016) to evaluate scalability of each part of the approach. We also evaluate the usefulness of the approach based on 66 participating MSc students for analysing test reports and identifying faults.
- Prototype tool: we implement all proposed solutions in Java to prove their concepts. The source code of the tool is available at GitHub repository[1]

[1]The source code of our prototype is available at `https://github.com/AMahfodh/IGTRRep`

Related Publications. The main parts of this thesis have been peer-reviewed and published in four papers, shown below. My contribution in each paper has been to develop the methodology and models, implement the tools, conduct the experiments, perform the analysis, and draft the paper. All my contributions were under the supervision of Prof. Reiko Heckel who also contributed to the formalisation not included in this thesis.

- Abdullah M Alshanqiti, Reiko Heckel, and Tamim Khan (2013). "Learning Minimal and Maximal Rules from Observations of Graph Transformations". In: *Electronic Communications of the EASST* 58

- Abdullah M. Alshanqiti and Reiko Heckel (2014). "Towards Dynamic Reverse Engineering Visual Contracts from Java". In: *Electronic Communications of the EASST* 67. URL: `http://journal.ub.tu-berlin.de/eceasst/article/view/940`

- Abdullah M. Alshanqiti and Reiko Heckel (2015). "Extracting Visual Contracts from Java Programs (T)". in: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 104–114. DOI: `10.1109/ASE.2015.63`. URL: `http://dx.doi.org/10.1109/ASE.2015.63`

- Abdullah M. Alshanqiti, Reiko Heckel, and Timo Kehrer (2016). "Visual contract extractor: a tool for reverse engineering visual contracts using dynamic analysis". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 816–821. URL: `http://doi.acm.org/10.1145/2970276.2970287`

1.5 Thesis Structure

The rest of the thesis is organised into seven chapters, categorised into three main parts. Chapter 2 gives fundamental background on model driven reverse engineering and visual contracts. Part II introduces our technical contributions relying on graph transformation concepts, discussed in three chapters:

- Chapter 3 explains our approach for tracing and constructing contract instances from implementations, supported by a running example. The approach begins with a static analysis step to extract structural models (class diagram). Then, it relies on dynamic analysis to trace and observe executions at runtime.
- Chapter 4 presents a detailed discussion on learning and generalising visual contracts by graph pattern matching. The resulting contracts are represented as minimal and maximal rules.
- Chapter 5 discusses the approaches for increasing the level of generality by inferring multi objects and attribute/parameter conditions.

In Part III, we evaluate and compare our proposed approaches with the state of the art in Chapter 6 and Chapter 7. We conclude the thesis, discuss the limitations and possible future directions in Chapter 8.

Chapter 2

Model Based Engineering

In this chapter, we go through the modelling concepts and terminology in the contexts of software engineering that are useful to understand the methods described in further chapters. First we introduce relevant definitions, acronyms and languages in Section 2.1 to explain what model we are interested in. The topics covered by the remainder of the chapter are design-by-contract and visual contracts in Section 2.2, and reverse engineering models in Section 2.3. These topics give more depth explanations about the problem statement, presented in Section 1.2.2, and identify the main challenges in the thesis. We explore, in Section 2.4, two different uses of contracts based on specification of component interfaces and dynamic monitoring and debugging.

2.1 Basic Terminology

Before delving into a discussion of model based engineering, it is useful to define what is a model. The term *model* devotes a "universal technique to understand and simplify the reality through abstraction" (Ramos, Ferreira, and Barcelo 2012). In software engineering, 'model' has been defined in many different ways

in the literature, comprehensively in (Ramos, Ferreira, and Barcelo 2012; Seidewitz 2003; Siegel 2014). Some definitions focus on representing the design artifact of a complex structure, such as a relational database schema, an interface definition (API), a semantic network or a complex XML document (Bernstein, Halevy, and Pottinger 2000). Other definitions lean more toward abstract formal specifications of the function, structure and/or behaviour of a system (Pender 2003; Siegel 2014). The latter involve viewing, manipulating, reasoning about systems under analysis and expressing how to understand the inherent complexity of software systems.

In general, a model is "a set of statements about a System Under Study (SUS), where a statement means expression about the SUS that can be considered true or false" (Seidewitz 2003). Another general definition by (Kühne 2006) states that a model is "an abstraction of a (real or language-based) system allowing predictions or inferences to be made".

In the context of this thesis, a model is a formal inferred description of the system's operation. It gives a high level of abstraction to convey information more effectively than code. It is represented by metamodel instances (see next section). In particular, we consider visual contracts used to model the operations of components or services by pre- and post-conditions.

2.1.1 System and Meta-Models

Figure 2.1 describes the semantic relationships between a system and different modelling levels (Brambilla, Cabot, and Wimmer 2012). For example, the relation *represents* (Favre 2004; Pender 2003) between *model* and *system* explains that the system can be described as a graph, composed of elements (nodes and edges). Each element in such a graph maps to an element in the metamodel by the *instancesOf* relation, such that their model conforms to the metamodel. A

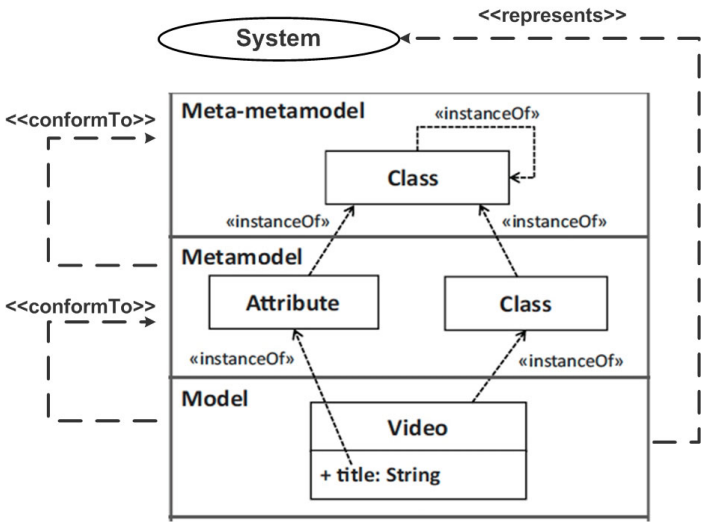


FIGURE 2.1: Relationships between system, models and metamodels (main part of this figure is originally taken from) (Brambilla, Cabot, and Wimmer 2012)

similar relation holds between metamodel and meta-metamodel. The purpose of designing metamodels is to define a set of concepts and relations between these concepts to validate their instance models (Seidewitz 2003).

Metamodels can be used for defining modelling languages or other classes of structures (Pender 2003). In our approach, we are concerned with extracting metamodel instances representing visual contracts from model instances representing object graphs.

2.1.2 Acronyms of Model Driven (MD*)

Modelling is an activity in software engineering that abstracts and analyses software systems for many different purposes. This explains the emergence of many modelling terms, used to describe different approaches and activities. For example, some approaches use modelling as a fundamental part of understanding and analysing software, while other approaches intent to support code generation. In the following list, we briefly define the main modelling terms and the relationship among them, see Figure 2.2:

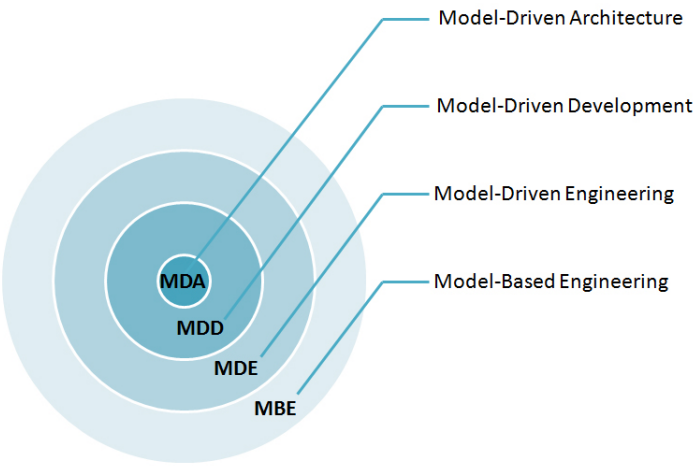


FIGURE 2.2: The main layers of model driven MD* (Brambilla, Cabot, and Wimmer 2012)

- Model-Based Engineering (MBE): Under the umbrella of Software Engineering (SE), MBE is a specific approach that aims to facilitate the software lifecycle through the use of models as the main artifacts (Ramos, Ferreira, and Barcelo 2012; Sommerville 2011). The standard utilisation of MBE and its future visions are discussed in (Ramos, Ferreira, and Barcelo 2012), which summarises proposed formalisms, modelling tools and applications. The expectations mentioned indicate that MBE will be a fundamental paradigm for development environments, but it would probably face the challenge of addressing: how to guarantee that the developed system meets the specification?

- Model-Driven Engineering (MDE) is an approach for software development that puts emphasis on domain models rather than programming (Schmidt 2006; Sommerville 2011; Stahl, Voelter, and Czarnecki 2006; Whittle, Hutchinson, and Rouncefield 2014). It aims at increasing the abstraction of a particular application domain, allowing to focus on knowledge and activities of that domain. Hence, it combines two key technologies: (1) Domain-Specific modelling languages (DSL) designed for a certain domain or context and (2) transformation engines and generators

to increase automation and analysis in program development (Schmidt 2006). The latter includes model-driven reverse engineering, which is crucial to this thesis. It is noteworthy that the main advantage of adopting MDE is not for code generation but to provide a well-documented software architecture (Whittle, Hutchinson, and Rouncefield 2014).

- Model-Driven Development (MDD) supports a more effective modelling approach (Mellor, Clark, and Futagami 2003; Sommerville 2011; Stahl, Voelter, and Czarnecki 2006). It is a subset of MDE but all its models are simultaneously abstract and formal, expressed in the UML. One of the main objectives of MDD is to automate the generation of implementations from models (Whittle, Hutchinson, and Rouncefield 2014). This implies that MDD models must support semantic functionalities, like coding in any programming languages. In this setting, models are not just used to design or document software system but also to implement software without a separate coding phase.

  The MDD approach offers advantages in the field of software development, especially in reducing the time consumed by manual coding and also in increasing the quality of generated software. However, there are still challenges in this approach, such as the complexity of generating all the required code from models, where manual change is still needed. This is due to the fact that models are more abstract than code, especially when code-patterns or external libraries are require to be used (Heckel and Lohmann 2007). Additionally, preserving the consistency between models and any manual changes in code is a challenge (Stahl, Voelter, and Czarnecki 2006).

- Model-Driven Architecture (MDA) is a standardisation of MDD by the Object Management Group (OMG) that uses a set of UML models (Siegel

2014). Unlike MDE, it does not support all aspects of the software engineering process. MDA is limited to design and implementation activities (Sommerville 2011). For instance, model based testing or debugging is not part of MDA. Figure 2.3 describes briefly the architecture of applying MDA in software development. The process relies essentially on UML to model a system at different level of abstractions and use transformation technique/rules between produced models and also between MDA's layers (Chen et al. 2006; Siegel 2014; Sommerville 2011). The general aims of MDA are to (1) represent systems at any level of abstraction, (2) enable transformations between models, (3) support execution of models and (4) support model exchange in a variety of modelling languages.

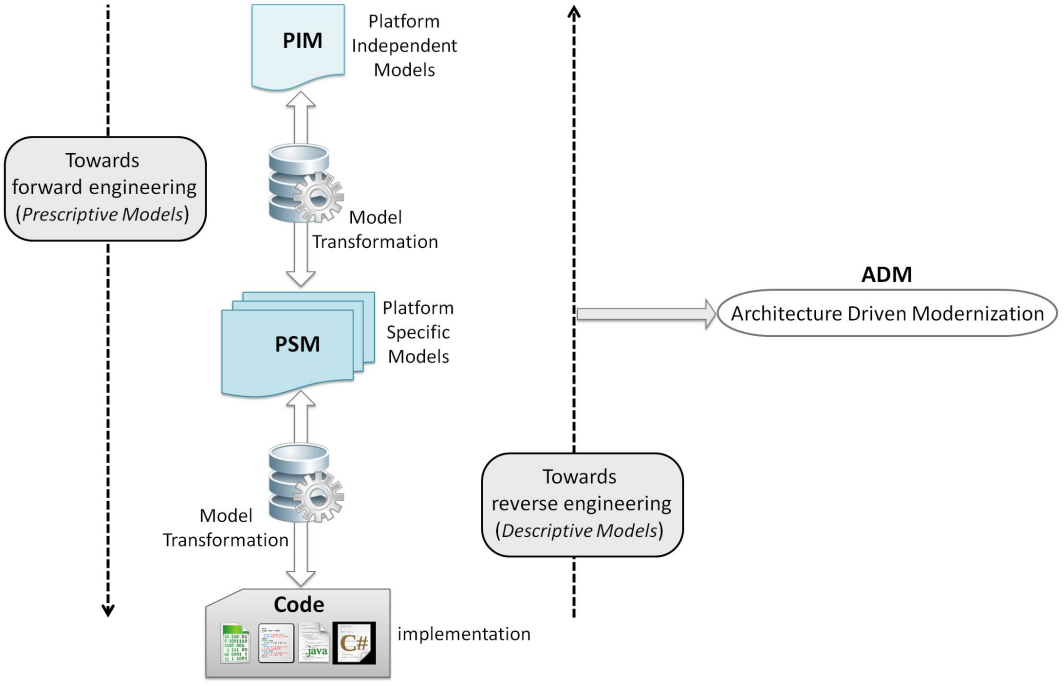


FIGURE 2.3: Architectural layers in MDA

The modelling approach followed in this thesis is a subset of MDE but does not adopt MDD. We focus on the generation of models from implementations, which is supported by MDE as model-driven reverse engineering (Favre 2004;

Whittle, Hutchinson, and Rouncefield 2014). This also can be applied on existing implementations of unmodelled systems. While MDD is also defined as a subset of MDE (Whittle, Hutchinson, and Rouncefield 2014), our approach is opposite to the main purpose of MDD, in the sense that MDD focuses on the generation of implementations from models. In the next sections of this chapter and also in Section 2.4, we expand the discussion about MDA and model-driven reverse engineering to clarify the relevant concepts in some more detail.

2.1.3 Prescriptive and Descriptive Models

Considering the two arrows in the left and right of Figure 2.3, software engineering can include *forward engineering* to specify a system to be constructed, or *reverse engineering* to extract knowledge or design information (Siegel 2014). In this setting, the model produced by forward engineering is *prescriptive*. It is called *descriptive* when applying the reverse engineering method as discussed in (Favre 2004) and citied from (Seidewitz 2003).

With respect to the correctness of models produced during the engineering process against the implementation, it is worth mentioning the point of view presented in (Favre 2004). Producing prescriptive or specification models usually means that the model carries the truth when testing a system under study. However, descriptive models suppose that the system represents the truth, i.e., "a descriptive model is said to be valid if everything said by the model about the system is actually true".

In general, the method we follow in this thesis aims to produce *descriptive* models from existing systems and one of the challenges here is to evaluate correctness of our models. This is not common practice in MDA as most of existing MDA tools focus on the forward engineering to produce prescriptive models (Chen et al. 2006).

2.1.4 Modelling Languages

A modelling language is a set of models defined by a metamodel, e.g., the set of all class diagrams making up the language of class diagram. This metamodel describes structure, syntax, semantics, and rules which are then used to define a model (Siegel 2014).

A number of languages for software modelling have emerged, some of which are for general use, i.e., they can be applied to any sector or domain, such as the well-known Unified Modelling Language (UML). Other languages are designed specifically for a certain domain or context, known as Domain-Specific Languages (DSLs) such as JML. As mentioned earlier, our modelling language is (visual contract), closely related to the following modelling languages:

- The Unified Modelling Language (UML) is "a family of modelling notations unified by a common meta-model covering multiple aspects of business and system modelling" (Siegel 2014). It is an instance of the MOF (Meta Object Facility) meta-metamodel defined by OMG (*The Unified Modeling Language (UML)* 2016). The approach proposed in this thesis relies on two kinds of UML diagrams: class and object diagrams. These diagrams are used to design structural and behavioural models, see example in Figure 3.4.

- The Java Modelling Language (JML) is a behavioural interface specification language for Java programs that follows the Design by Contract paradigm (Leavens, Baker, and Ruby 2006; *The Java Modeling Language (JML)* 2016). It exploits annotation comments in the source files to specify the pre/postconditions and invariant of Java methods. We compare JML to the visual contracts we seek to extract from existing systems in Section 2.2.

2.2 Contracts MDE

This section focuses on defining what is a contract in MDE, and discusses what are the visual contracts we consider in this thesis. Related contract based approaches and tools are given in Section 2.4.

2.2.1 Design-by-Contract

The term *contract*, in software modelling, refers to a formal document/agreement between two parties, in order to protect their responsibilities and rewards, e.g. between class and client (Lohmann, Sauer, and Engels 2005; Meyer 1992). This agreement is normally used to improve software reliability, including correctness and robustness of design models against their specifications. Correctness, in this setting, is usually checked by assertions, which evaluates to true or false, expressed in the agreement as preconditions, postconditions and invariants (Heckel and Lohmann 2005; Lohmann, Sauer, and Engels 2005). Robustness aims to handle exceptions in the case of assertion violations, e.g, when indicating a fault at one of contract's parties.

For Java methods, specifications between caller and callee can contain many redundant conditions that are harder to maintain, and could make software more complex to understand. This has been addressed by the Design by Contract (DbC) (Meyer 1992), which aims at increasing understandability of software specifications. DbC is supported by languages such as Eiffel[1], JML, iContract (Kramer 1998), etc.

Listing 2.1 illustrates an example, in Eiffel syntax. The contract of *registerCli ent(..)* discussed in the running Example 3.2. Here, the main clauses *require*

[1]Eiffel is a commercial DbC tool developed by Bertrand Meyer (Meyer 1992), `https://www.eiffel.com/values/design-by-contract/`

```
//
// Eiffel syntax
//
procedure_name(argument declarations) is
                -- Header comment
        require
                Precondition
        do
                Procedure body
        ensure
                Postcondition
end
//
// example from the case study, discussed in Chapter 3.
//
registerClient(String city, String clientName) is
        -- method to register a client at the branch name (city)
        require
                city /= null
                clientName /= null
        do
                --// algorithm statements ..
                Client c = new Client();
                Branch b = getBranch(city);
                ...
        ensure
                b /= null
                b.cMax = old b.cMax + 1
                b.of.contain(c)=true
end
```

LISTING 2.1: Contract representation based on Eiffel syntax

and *ensure* are used to specify input pre- and post-conditions on which logical contract checks can be defined.

2.2.2 Visual Contracts

In this thesis, we use UML notation, formally supported by graph transformation rules to represent contracts as a behavioural model. Visual contracts support high-level behaviour modelling of software operations (Meyer 1992). They consist of a pair of conditions that describe a system's state, relevant to a specific operation, before (pre-condition) and after (post-condition) execution. The pre-condition specifies all requirements to execute the operation, while the post-condition describes its effects (Heckel and Lohmann 2005; Lohmann, Sauer, and Engels 2005).

The UML includes the textual Object Constraint Language (OCL)[2] to describe rules by pre- and post- conditions and invariants. Despite OCL's capabilities, it does not provide a graphical notation, which makes it harder to use or maintain software specifications (Lohmann, Sauer, and Engels 2005).

Software modelling languages built on mathematical Formal Methods (FMs) strive precise and unambiguous models. However, FMs are often complicated and require high cost to be used (Amálio and Glodt 2015; Woodcock et al. 2009). Therefore, the usability of FM tools by developers or designers is problematic. The use of graph transformation rules as a formal concept with visual UML notation will support, in a graphical and potentially understandable way, what the textual OCL fails to provide.

Visual contracts are typed over design classes from UML class diagrams and use UML object diagrams to represent their pre- and post-conditions. The combination between graph transformation rules and UML contributes for two key advantages (Baresi and Heckel 2002; Lohmann, Mariani, and Heckel 2007):

- It introduces rigorous behaviour modelling of software operations with an option for formal analysis (Baresi and Heckel 2002). A similar idea, with respect to the use of Z (Spivey and Abrial 1992) instead of GT, is discussed in (Amálio and Glodt 2015), see Contract Diagrams (CDs).
- It uses a UML-like notation as a front-end, which is easier to understand by software designers (Lohmann, Mariani, and Heckel 2007).

[2]The Object Constraint Language (OCL) is a declarative language, i.e., complement of the UML notation with the aim of addressing some of its limitations. It is set by the (OMG) as a standard for object-oriented analysis and design, see for more details `http://www.omg.org/spec/OCL/`

2.3 Model Driven Reverse Engineering

This section focuses on model driven reverse engineering activities and approaches, under the umbrella of MBE. In 1990, (Chikofsky and Cross 1990) gave effective definitions of many concepts in the context of Reverse Engineering (RE), which are still in use nowadays. They defined RE as an examination process (1)to identify the technological principles of the system's components and their inter-relationships, and (2) to create a new representation of the system at a higher level of abstraction. RE, in general, can be applied at any level of abstraction and covers only analysis activities, i.e., does not modify or change the software system. It addresses several problems, such as re-documentation, re-design, and knowledge recovery (Chikofsky and Cross 1990).

RE has been successfully applied in many domains, reviewed and highlighted in (CanforaHarman and Di Penta 2007). Examples of these areas include relational databases, identifying reusable assets, recovering architectures, recovering design patterns, building traceability between software artifacts, code smells, identifying clones, computing change impacts, re-modularising existing systems, testing and audit security, migrating toward new architectures and platforms, etc. (Canfora, Di Penta, and Cerulo 2011). In the next sections, we firstly give some details about general goals and difficulties in RE. Then we narrow down to model driven analysis approaches in Section 2.3.3 and model construction in Section 2.3.4.

2.3.1 General RE Motivations and Goals

There are six comprehensive goals in RE, proposed by Chikofsky (Chikofsky and Cross 1990), i.e., coping with complexity, generating alternate views, recovering

lost information, detecting side effects, synthesizing higher abstractions, facilitating reuse. Nevertheless, they are all targeted to support comprehension and evolution (Canfora, Di Penta, and Cerulo 2011), mostly, for either software maintenance[3] and/or development. In addition to this, (Eilam 2011) presents and examines many RE tools, related to understanding different issues of computer security. For instance, reverse engineering existing systems to evaluate the level of security or defeat copy protection. Other security tools are used to audit malicious software on both sides; malware developers or hackers to discover vulnerabilities and developers of antivirus to trace and fix infected systems (Eilam 2011).

2.3.2 Scientific Challenges

The essence of reverse engineering involves (1) analysing software artifacts to extract information and (2) translating these information into abstract representations. Performing the first step (software analysis) is subject to circumstances such as the executability of the system, the availability of source codes and expected view by abstracting facts of the system. Consequently, different approaches (e.g., static, dynamic, hybrid and historical) can be applied, targeting different purposes or uses. We go deeper into these approaches and discuss their pros and cons in Section 2.3.3.

With respect to the second step, translating the derived information into more abstract and understandable representations, this step contains challenges such as incomplete and/or inaccurate representations of the system (Binkley 2007; Canfora, Di Penta, and Cerulo 2011; Ernst 2003). This is because the derived information could either neglect some important behaviour or include false positive/negative. For example, tracing implementations by specifying the scope

[3]The IEEE Standard for Software Maintenance "recommends reverse engineering as a key supporting technology to deal with systems that have the source code as the only reliable representation" (Canfora, Di Penta, and Cerulo 2011).

of element types, such as classes or methods, could exclude other important elements, resulting in (false negative) aspects not permitted by the system. False positives, however, can result from analysing the whole implementation, including dead code behaviours. More discussion about the second step is given in Section 2.3.4.

In addition, there is often a trade-off between having a *fully automatic* process that suffers from the inability to include experts' knowledge and a *semi-automatic* one that is subject to human interactions and subjective decisions. The latter, for example, involves specifying particular inputs to analyse the subject system (Canfora, Di Penta, and Cerulo 2011). This makes the derived behaviours dependent on user-inputs.

Another challenge is traceability and the ability to define relations between different levels of abstractions (Chen et al. 2006). This is due to the fact that the lower level may describe more detailed information (e.g. aspects from rich programming languages), than the higher level that usually abstracts from specific information. For instance, the transformation from PSM to PIM implies that there should be a loss of information. To address this and achieving complete bidirectional transformations without losing information, it requires semantic knowledge to fill the gap between different levels of abstractions (Chen et al. 2006).

The approach proposed in this thesis is semi-automatic, since identifying test cases and the scope to limit observing execution are manual activities. The rest is automatic, starting from tracing to the inference of contract features. Although it is a challenge to generate a complete representation, our solution allows to improve completeness by incrementally including new behaviours when observing additional contracts as required (cf. Section 4.3). For the traceability problem, it is beyond the scope of this thesis.

2.3.3 Analysis Approaches

Reverse engineering begins with an analysis to derive information from the subject system. Here, we give a brief survey of the various analysis mechanisms and discuss their advantages and disadvantages. The relevant analysis mechanisms to this thesis are static and dynamic, explained in Chapter 3. We discus hybrid and historical analysis as informative background.

Static Analysis

Static analysis of source code, exemplified by (Rountev, Volgin, and Reddoch 2005; Sarkar, Chatterjee, and Mukherjee 2013; Tonella and Potrich 2003), aims to extract information from implementations without the need to execute any part of the system (CanforaHarman and Di Penta 2007; Ernst 2003; Venkatasubramanyam and G. R. 2014). It requires parsing source code into internal representations, e.g., Abstract Syntax Trees (AST), for further analysis, which is useful with incomplete systems, e.g., components that cannot be executed independently (Rountev, Volgin, and Reddoch 2005)

The main feature of static analysis lies in the possibility of describing all possible behaviours of the system (Ernst 2003; Venkatasubramanyam and G. R. 2014). Generating CFG by parsing code, for example, provides a complete view of all possible paths that the execution can run through. This could lead to generate precise results in reasonable time. However, static analysis suffers from difficulties to observe important object-oriented behaviours, including the inability to recognise: (1) false-positive behaviours (e.g., a valid path in a CFG that cannot be executed in the code), (2) class polymorphism and (3) static and dynamic class binding (i.e., occurs at compile or run time) (Canfora, Di Penta, and Cerulo 2011).

For example, (Tonella and Potrich 2003) propose a static approach for generating sequence and collaboration diagrams from C++ code, thereby potentially over-approximating the actual behaviour. In (Binkley 2007; Canfora, Di Penta, and Cerulo 2011; CanforaHarman and Di Penta 2007), the state of the art on static analysis approaches is described. Their discussions emphasised the need to address three major challenges. Being able to (1) parse different language variants and non-compilable code, (2) extract facts and (3) extract a program's semantics.

Dynamic Analysis

The information extracted using dynamic analysis describes actual behaviours of a system in terms of traces of execution (Cornelissen et al. 2009; Ernst 2003). This kind of analysis allows to identify run-time information, such as actions of method calls, changes of object states and values of variables etc. Moreover, the trace outputs are generated in response to a particular set of inputs, usually by testing or real time monitoring (Venkatasubramanyam and G. R. 2014).

The main part of our solution is dynamic (discussed in detail in Chapter 6), based on running the system, as are others (Brito et al. 2012; Zhao, Kong, and Zhang 2010; Ziadi et al. 2011). The potential drawback here is that the extracted model represents only those behaviours that are actually executed. In general, many dynamic reverse engineering approaches take advantage of aspect oriented concepts for instrumentation at different levels. We focus on instrumenting low-level Java code to generate visual contract instances at runtime. A similar strategy lies behind (Brito et al. 2012) generating object graphs or (Ziadi et al. 2011) extracting sequence diagrams.

In order to permit dynamic tracing for a system it requires instrumenting[4] the source/complied code or profiling data collected by debugging methods (Cipolla-Ficarra 2010; Venkatasubramanyam and G. R. 2014). This is unlike static information in many ways. For instance, dynamic information is more precise in identifying feasible behaviours, as it gives evidence of each valid path by means of recording actual access at run-time. It is able to detect dynamic object-oriented behaviours, such as dynamic binding (Ernst 2003).

(Cornelissen et al. 2009) provide a comprehensive survey on dynamic analysis approaches, focusing on program understanding, while more general discussion and challenges are covered in (Canfora, Di Penta, and Cerulo 2011). They point out that many techniques and tools still provide unsatisfactory results. Since dynamic analysis relies on the inputs used to execute the target system, this leads to three main drawbacks. Firstly, it is limited to observing partial behaviours only, which in turn impede the construction of a complete view. Secondly, it suffers from the inability to determine particular inputs to trigger the system elements of interest. Thirdly, as the trace output is a sequence of actions, deriving the semantic meaning of these actions is complicated, e.g. understanding precisely the knowledge behind the order and dependency of all actions is currently impossible (Ernst 2003).

In addition, while each trace output provides precise information, gathering multiple traces can cause a scalability issue. On top of this, the generalised information from multiple traces may become imprecise (Cipolla-Ficarra 2010; Cornelissen et al. 2009; Ernst 2003). For example, the generalisation could produce an over-approximation model from observed behaviours that may include aspects not allowed by the system, caused by inaccurate inferences. The under-approximation model can also be produced, resulted by accidentally omitting some necessary information.

[4]The instrumentation aims to modify code, by inserting annotations at particular parts, allowing to record/collect data accessed at run-time. It is usually done by Aspect-Oriented Languages (AOL) tools

Hybrid Analysis

Combining static analysis and dynamic analysis techniques is always worth considering; there are some situations where reverse engineers must weigh the pros and cons of each individual technique applied (Ernst 2003). In hybrid analysis, it is possible to avail the strengths of both static and dynamic techniques to overcome their limitations (Canfora, Di Penta, and Cerulo 2011; Elberzhager, Münch, and Nha 2012; Ernst 2003). For example, the imprecise results (e.g., false positive behaviours) obtained by static analysis can be tested dynamically to improve their precision via feasible behaviours. Furthermore, inferring semantic information by static analysis (e.g. mining recurring code patterns) could help to filter out large execution traces and improve their generalisation techniques, as proposed in (Hassan et al. 2008).

The main issue with the hybrid approach is the need to combine and hop between different analysis tools. This may affect usability by developer, including required set-up and complex configurations. A report on many existing hybrid approaches, focusing on quality assurance techniques, can be found in (Elberzhager, Münch, and Nha 2012).

Historical Analysis

This type of analysis is well-known in mining software repositories to help in understanding software system. A software repository is an archiving storage, normally on online servers, on which software packages, source code and other valuable information can be retrieved and installed during software evolution (Kagdi, Collard, and Maletic 2007). It supports reporting tool, such as bug-tracking (e.g., Bugzilla). As described in (Canfora, Di Penta, and Cerulo 2011), and also in an extensive survey on historical analysis approaches (Kagdi,

Collard, and Maletic 2007), the repositories contain valuable details about software changes over times and their metadata[5]. They can be used to feed some machine learning algorithms, as training sets, in order to infer useful information that supports software analysis.

Compared to static or dynamic analysis for the purpose of program comprehension, this approach gives a different angle of understanding software system. It helps to understand: (1) "how an artifact was modified over time?" and (2) "what artifacts changed together?" (Canfora, Di Penta, and Cerulo 2011).

2.3.4 Model Construction

The second step of the RE process is to create a new representation of the system at a higher level of abstraction (Chikofsky and Cross 1990). In practice, this is difficult for three reasons discussed in (Canfora, Di Penta, and Cerulo 2011; CanforaHarman and Di Penta 2007). Generally, analysing derived information (i.e., from the first step of the RE process) requires (1) a powerful query language to construct views and (2) the possibility to visualise constructed views. It also requires (3) enough understanding from reliable sources to abstract low level artifacts in a way that enables constructing them for higher levels.

There are many existing approaches to build views at different levels of abstraction, one of which is the design level (Canfora, Di Penta, and Cerulo 2011). Model-driven approaches belong to this level. The techniques used for such derivation are commonly based on process mining (Liesaputra, Yongchareon, and Chaisiri 2015), grammar inferences[6] (Cook and Wolf 1998; Zhao, Kong, and Zhang 2010) and statistical analysis by machine learning algorithms (Qiu

[5]Examples of software metadata: "who made the change, why the change was made, when the change was done etc" (Kagdi, Collard, and Maletic 2007)

[6]The grammar inference problem can be informally stated as follows. "Given some sample sentences in a language, and perhaps some sentences specifically not in the language, infer a grammar that represents the language" (Cook and Wolf 1998)

et al. 2010a). Process mining technique, for example, have been considered in many algorithmic approaches, including heuristic mining and matching approaches using, e.g., pattern recognition (Conte et al. 2004; Dahm et al. 2015; Jouili, Mili, and Tabbone 2009; Liesaputra, Yongchareon, and Chaisiri 2015).

2.4 The Use of Visual Contracts in MDE

Contract-based specification plays an important role in MDE as it supports formal definitions of transformation requirements between layers and/or models. It takes concrete shape in the context of MDA, particularly in the transformation between PSM and implementation layers. Here code generation from models, for example, can be formally specified and automated. A typical attempt for dynamic code generation is proposed in (*The Fujaba Tool Suite: From UML to Java and Back Again* 2016), see Figure 2.4 (C). This is targeted to generate concrete behaviour code for standalone applications. Other approaches, as shown in (A) and (B), allow to generate structural code, including (contracts) specification code (Engels et al. 2006b; Lohmann, Engels, and Sauer 2006). Our approach aims to generate a model at PSM from the implementation as described in (E).

2.4.1 Specification of Component Interfaces

Contract based specification, at code level (e.g., for an API of a library), can add precise descriptions about software behaviours within the code (Karaorman and Abercrombie 2005; Leavens, Baker, and Ruby 2006; *The Java Modeling Language (JML)* 2016). At model level, however, contract in graphical representation can support formal development activities such as (1) visual behavioural specification (Heckel and Lohmann 2007; Lohmann, Engels, and Sauer 2006) and (2) generating behavioural code (*The Fujaba Tool Suite: From UML*

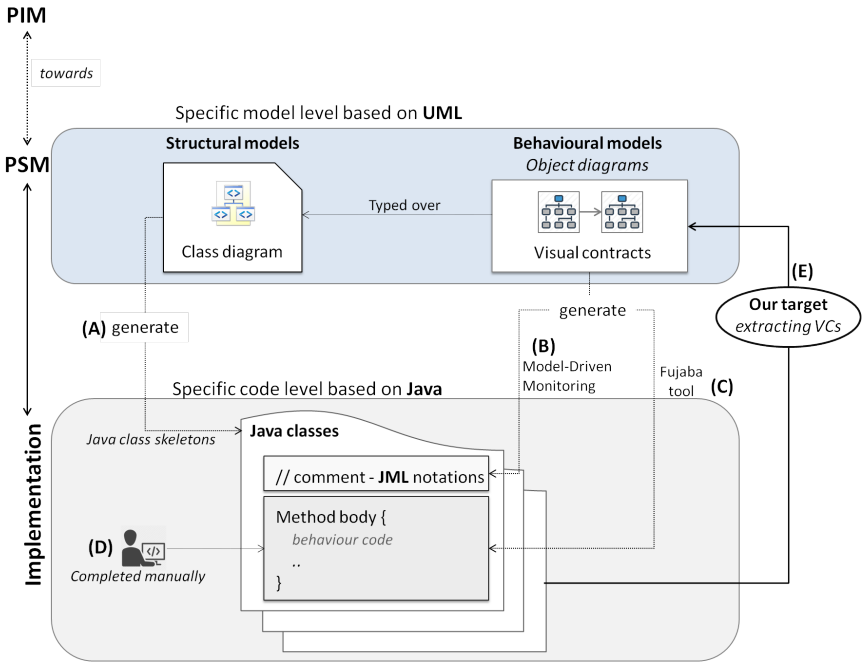


FIGURE 2.4: The involvement of visual contracts in MDA approach

to Java and Back Again 2016). Apart from implementation and model levels, contract-based specification is also used in web service interfaces (Heckel and Lohmann 2005) and feature models (Thüm et al. 2012). In our approach, we are particularly interested in discussing contracts at UML (Siegel 2014) level but extracted from implementations in Java, as shown in Figure 2.4 (E).

With respect to the tools used for contract specifications, we have come across two main kinds:

- Contract tools based on Java code, e.g., jContractor[7] (Karaorman and Abercrombie 2005) and tools that use annotation languages, such as JML (Leavens, Baker, and Ruby 2006) ignored by the Java compiler. JML is the most wider used and integrated with modelling tools.

- Tools for modelling specifications visually including contract diagrams (i.e., modelling behaviour of operations by pre- and post-conditions) such

[7]jContractor is a Java implementation of Design By Contract for the Java language. Homepage can be found in `http://jcontractor.sourceforge.net/`

as VCB (Amálio and Glodt 2015) but based on Z (Spivey and Abrial 1992).

2.4.2 Dynamic Monitoring and Debugging

In order to make visual contracts executable, they have to be translated to executable code that can be compiled by, e.g., jContractor (Karaorman and Abercrombie 2005) or JML (Leavens, Baker, and Ruby 2006), etc. Contract compliers, in general, provide dynamic monitoring and debugging mechanisms by instrumenting source code that define textually precondition, postcondition and class invariant operations. At runtime, executing contracts allow to trace assertions and report with any violations.

Considering Figure 2.4 (B), translating VCs from PSM to JML assertions (at code level) has been considered in several approaches (Engels, Güldali, and Lohmann 2007; Engels et al. 2006b; Heckel and Lohmann 2007; Lohmann, Engels, and Sauer 2006), discussing so-called Model-Driven Monitoring (MDM). Apart from generating Java classes from class diagram, these approaches propose practical mapping solutions, with tools support. Given a class diagram and visual contract, these approaches generate:

- Java class files describing skeleton structures, shown in Figure 2.4 (A). Such code structures include operation signatures from interfaces, attributes and method types from classes, etc. The implementation of method bodies will not be generated but their declarations would tell programmers which parts of the code needs to be implemented and completed. For example, consider classes *Branch* and *Client* from the class diagram under Figure 3.4 (A). The generated Java classes will be similar

to the code, lines 22-50, shown in the left of Figure 3.4. The implementation of method bodies such as *registerClient(..)*, lines 4-20, is expecting to be completed/modified manually.

- From each VC, they generate contract assertion code for each method, similar to the code described in Listing 2.1 but in JML syntax. These assertions are written as annotations and used to monitor implementation of behavioural code.

The key objective of MDM is to monitor and test manual implementations against their specifications that are hard to generate by following MDA concepts. For example, MDM allows to formally check correctness of Figure 2.4 (D) that is completed by a programmer (Heckel and Lohmann 2007), by referring to the VC's specification. This kind of checking could address consistency problems between PSM and implementation, which is one of the MDA's challenges.

The contracts (at instance level) extracted by our approach can support a comparable monitoring mechanism with MDM approaches, without depending on contract compilers. Since the contract specifications are already determined and the target is to check implementations against specifications, it is possible to use model-based testing (MBT) based on visual contracts. To be more precise, visual contracts have been substantially used for: (1) deriving test cases (Dai et al. 2007; Guldali et al. 2009; Heckel and Lohmann 2005; Runge, Khan, and Heckel 2013), (2) acting as a test driver or oracle (Dai et al. 2007; Khan, Runge, and Heckel 2012b), (3) measuring coverage (Khan, Runge, and Heckel 2012a) and (4) verifying models (Dotti et al. 2006). Therefore, by taking these contributions into consideration, our approach could fill the missing part to automate MBT.

2.5 Summary

In this chapter, we discussed the underlying terminologies and concepts of models — in the contexts of software engineering — that are useful to understand many parts of the thesis, particularly visual contracts. We presented the necessary background, related to reverse engineering technique, that covers general objectives, challenges, advantages and disadvantages of existing analysis approaches. We have explained that visual contracts combine graph transformation rules that support formal analysis, and UML class/object diagrams that are easy to use by software designers.

After giving the general overview of contract's uses (as specification of component interfaces and dynamic debugging) and highlighting the major technical challenges, we go into more detail to explain the main contributions in the three chapters in Part II. We will introduce our proposal, focusing on how visual contracts can be extracted from existing software.

Part II

Inference of Visual Contracts

Chapter 3

Extraction of Contract Instances

This chapter presents the first step of our methodology, published in (Alshanqiti and Heckel 2014), for constructing contract instances from deterministic Java applications. This step is based on observing runtime changes in object structures when an operation is executed. Any contract instance represents the behaviour of a specific invocation of the operation. Its precondition captures the objects that were read, while the postcondition describes how the object structure changed during the invocation. This also means that a contract instance can represent only one possible outcome of the executed operation.

As discussed in Chapter 2, dynamic reverse engineering requires to trace a system's execution, e.g., by instrumentation of the code, as well as the analysis of the traces to extract behavioural models. To explain how our approach has met these requirements, we divide this chapter into four main sections. In Section 3.1, we define, at a high level, our strategy for constructing contract instances from object observations. Section 3.2 presents a simple case study to illustrate the problem in more detail and our contributions. This case study is also used by the forthcoming chapters as a running example. Section 3.3 and 3.4 present the implementation of our approach that focuses on instrumenting Java bytecode using (*AspectJ* 2016) to observe the internal state of the system

and its changes during execution. The trace output is a recorded sequence of elementary read and write operations that is analysed to construct the contract instance.

3.1 Extraction Approach

Building behaviour models, in the form of visual contract instances, from Java application begins with observing system executions, which results in the generation of a set of accessed objects. This observation requires dynamic analysis, which can be performed by tools that support code injection (i.e., instrumentation for white box analysis) such as (*AspectJ* 2016; *BTrace* 2017; *Byteman Trace* 2017; *InTrace* 2017; *Method Tracing* 2017). Regardless of the different features provided by these tools, they all support the basic functionality of tracing read/write access objects at runtime, making an insignificant difference between them from the viewpoint of this thesis. Thus, the well-known AspectJ has been chosen as tracing tool for our approach, discussed in Section 3.3.

Given a set of objects O, obtained by tracing an operation invocation *op*, ordered by access time, where $O = \{o_1, \cdots, o_k\}$ and $k \in \mathbb{N}$. Each object o_k is a result of access by an elementary r read or w write operations, and the same o_k can be accessed more than once during *op*'s execution. We assume w access to an object o_k if:

- at least one data-type attribute of o_k's local member has a change,
- it has been connected or disconnected to itself and/or any other object o_j such that $o_j \in O$, and
- it has been deleted or created.

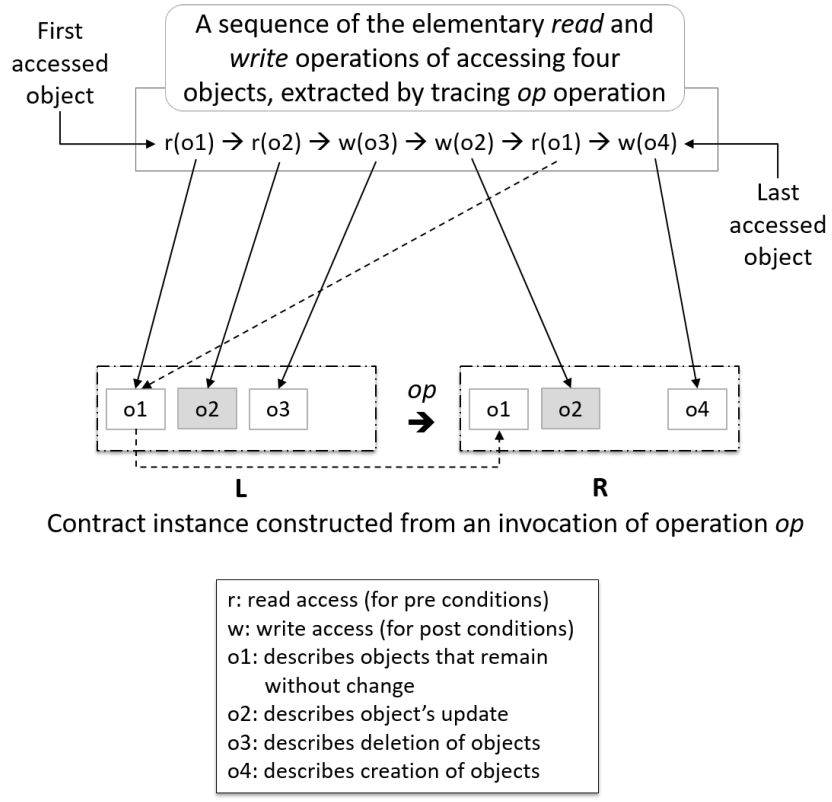


FIGURE 3.1: Description of how an extracted sequence of read and write operations of accessing objects are used for constructing contract (or rule) instances

while, we assume r access to o_k if it has no w access (i.e., has no change). Considering Figure 3.1, we analyse a sequence of the elementary r and w operations of accessing a set of objects $O = \{o1, o2, o3, o4\}$ to construct a contract instance. These objects are generated by the observation of *op*'s execution such that the first ($o1$) and the last ($o4$) accessed object in the set O represent the start and end of *op*'s execution.

Figure 3.2 gives the overview of our approach for extracting contract instances in four steps. The first two steps focus on tracing, discussed in Section 3.3. Step 3 and 4 present the implementation of our strategy for contract construction, introduced in Section 3.4. Starting with empty pair of graphs LHS and RHS, our strategy aims to decide at which graph LHS and/or RHS an accessed

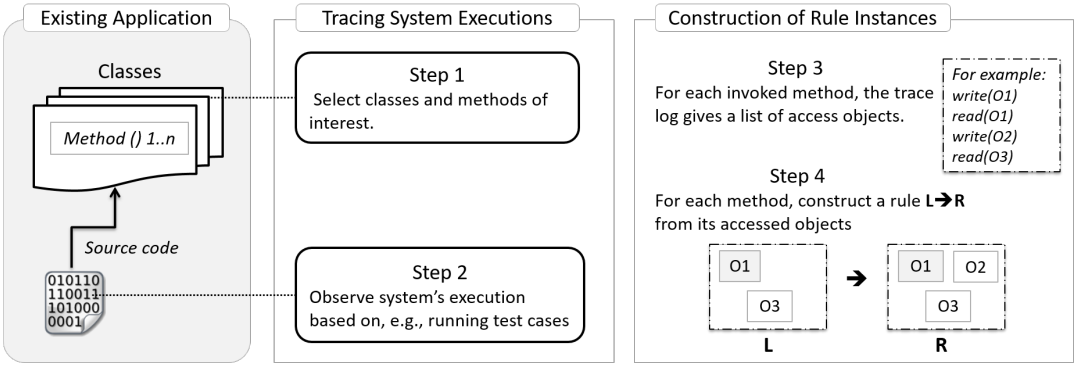


FIGURE 3.2: Overview of extracting contracts instances

object o_k needs to be added or neglected. We explain briefly the process of our strategy, illustrated in Figure 3.1, as follows.

1 Add $LHS \leftarrow w(o_k)$ for object deletion, while $LHR \leftarrow w(o_k)$ for object creation cases, see $w(o3)$ and $w(o4)$ in Figure 3.1 as examples.

2 Add $\forall r(o_k) \in O$ into LHS without duplication (i.e., has not been already added in step 1), see $r(o1)$ and $r(o2)$ as examples.

3 $\forall w(o_k) \in O$ that are caused by updating value of date type members, or by creating/destroying relation with other objects, add their last write-access that appear in the set O into RHS. This means if an objects being changed several time, we only consider the first $r(o_k)$ and the last $w(o_k)$, while we ignore all the other $w(o_k)$ that occur in between, see $w(o2)$.

4 Add a copy of $\forall r(o_k) \in LHS$ into RHS, e.g. see $r(o1)$.

3.2 Running Example

We consider a simple case study of a Car Rental Service[1] as a running example to illustrate the problems treated in this thesis. We will use it frequently

[1]Examples of Java code fragments from implementation can be found in Appendix A.

```
public interface IRental extends Serializable{

  public String registerClient(String city, String clientName);
  public String makeReservation(String ClientID,String pickUp,String dropOff)
  public void cancelReservation(String Reference);
  public void cancelClientReservation(String clientID);
  public void pickupCar(String Reference);
  public void dropOffCar(String Reference);
  public Reservation[] showClientReservations(String clientID);
  public Client[] showClients (String city);
  public Car[] showCars (String city);

}
```

LISTING 3.1: Interface of a Car Rental Service

in Part II to illustrate the contributions we have made, and also to evaluate our approach in Part III. This case study is designed to represent a range of different preconditions and effects of operations over a complex object-structure, including the creation of objects, the creation and deletion of links as well as attribute updates and constraints.

An interface with the operations of application is given in Listing 3.1. The class diagram in the top right of Figure 3.4 shows the classes implementing these operations. The specification of the operations, given in Figure 3.3, describe that operation *registerClient(..)* creates a new client object, registers it with the branch at *city*, and updates attribute *branch.cMax*. Operation *makeReservation()* allows to book a car for a client by creating a new reservation object that must be registered with pickup branch. Operations *pickupCar()* and *dropoffCar()* control the movement of a car from the *pickup* to the *dropoff* branch.

In the following subsections we explain informally how Java classes, object structures and methods are represented by class, object diagrams and visual contract, respectively.

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.

`Void` **`pickupCar`** (`String reference`)

Removes linkes *pickup* and *for* between reservation object and *pickedup* branch. The reserved *car* can only be picked up once. If there is no suitable reservation, the operation does not have an effect on the state.

Parameters:
`reference` – non-null string value used to get *reservation* object by reference.
Returns:
`no return`

`Void` **`dropoffCar`** (`String reference`)

Creates new link *at* by returning reserved car to the *dropoff* branch, and removes reservation object with all its links, namely: *made*, *pickup*, *dropoff* and *for*. If there is no suitable reservation, the operation does not have an effect on the state.

Parameters:
`reference` – non-null string value used to get *reservation* object by reference.
Returns:
`no return`

`Void` **`cancelReservation`** (`String reference`)

Removes reservation object that matches *`reference`* (if it exists) with all its links, namely: *made*, *pickup*, *dropoff* and *for*. If there is no suitable reservation, the operation does not have an effect on the state.

Parameters:
`reference` – non-null string value used to get *reservation* object by reference.
Returns:
`no return`

FIGURE 3.3: Specification of Car Rental Service

3.2.1 Structural Features

The left of Figure 3.4 shows code fragments of Java classes *Rental, Branch,* and *Client* and their corresponding classes in the class diagram under (A). Fields of Java classes are represented as UML attributes or associations depending on their type, i.e., fields of primitive types lead to UML attributes, while fields of object types turn into associations. For example, the fields declared in lines 26-28 are represented as attributes, while the (at) and (of) fields in lines 24-25 become (1-to-many) association.

At the implementation level, we translate a ($*$) association to an explicit collection object. This is because collection objects provide information about the organisation of their elements. For example, elements of an *ArrayList* in (B) are known to be ordered. The choices made in representing class structures determine the representation of object structures. For example, (B) shows fragments of two states representing objects referred to as ($b1$) and ($c1$) and their changes due to the invocation *registerClient(...)*.

3.2.2 Behavioural Code and Contracts

Each visual contract describes the pre- and postcondition of a possible way of executing a method. Consider (B), which shows relevant objects of two states resulting from an execution of the *registerClient()* operation in lines 4-20. The rule describing this execution is shown in (C). It specifies how the objects and their attributes and links change from one state to the other. In the code this could be achieved by assignments, which represent the creation and deletion of objects and links or the update of attributes. For instance, the assignment in lines 12 and 15 represent the creation of an object and an edge respectively.

```
1  package RentalService;
2  public class Rental implements IRental{
3      ...
4      public String registerClient(
5                      String city,
6                      String clientName) {
7
8          Branch b = getBranch(city);
9
10         if (b !=null){
11
12             Client c = new Client();
13             c.name =clientName;
14             c.iD = b.city + (b.cMax++);
15             b.of.add(c);
16
17             return c.iD;
18         }
19         return null;
20     }
21 }
22 public class Branch implements
       Serializable {
23
24     private ArrayList<Car> at;
25     private ArrayList<Client> of;
26     private String city=null;
27     private int cMax;
28     private int rMax;
29     ...
30     public Branch (String City,
31                    int CMax,
32                    int RMax) {
33
34         this.city = City;
35         this.cMax=CMax;
36         this.rMax=RMax;
37     }
38 }
39 public class Client implements
       Serializable {
40
41     private String name;
42     private String iD;
43     ...
44     public Client ( String CName,
45                     String CID) {
46
47         this.name = CName;
48         this.iD = CID;
49     }
50 }
```

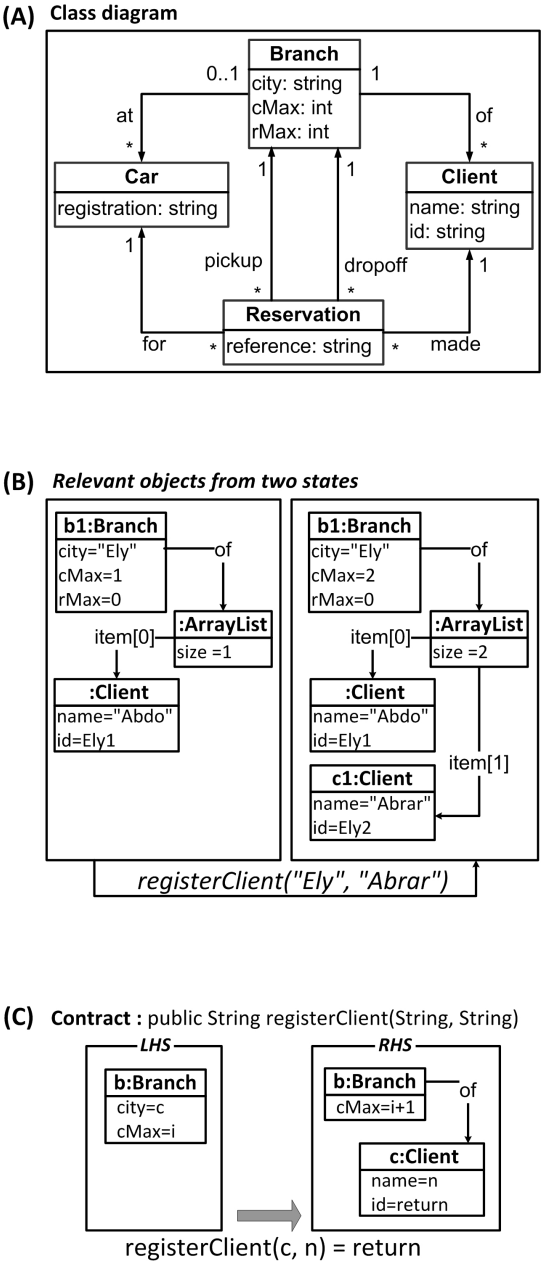


FIGURE 3.4: Mapping classes and objects and representing operations

It should be noted that information about object references and attribute values between states is necessary when extracting visual contracts. Without this information it is impossible to distinguish between some behaviours. For example, consider code fragments of operation *makeReservation()* in Figure 3.5. It is impossible to determine statically whether the assignment in line 9 should

```
1   ...
2   public Reservation(
3           String Reference,
4           Client Made,
5           Branch Pickup,
6           Branch Dropoff,
7           Car CarFor) {
8       ...
9       this.pickup=Pickup;
10      ...
11  }
12  ...
```

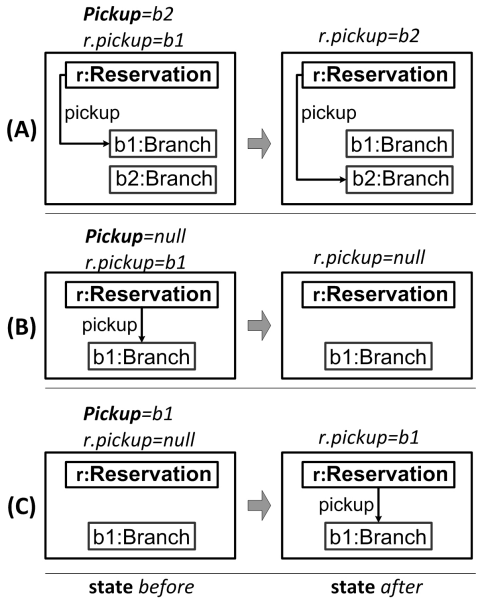


FIGURE 3.5: Uncertain cases to recognise code behaviours by static analysis

be represented as creating or deleting a link between those two objects. More precisely, consider the states before and after, as shown in (A), (B) and (C) of Figure 3.5. Such assignment could lead to three different behaviours, depending on the value of *Reservation.pickup* in the state before (e.g. *r.pickup=b1*) and the reference of passed object parameter *Pickup* (e.g. *Pickup=b2* from third parameter of the *constructor*) that cause change to the state of the *reservation* object (r). Consequently, this raises the need to use dynamic analysis to extract visual contracts.

3.3 Tracing System Executions

The ability of executing existing implementations is required. However, we assume that the required test cases are given along with implementations. To organise this section, we first introduce the basic concepts of AOP and AspectJ that we used to generate logs from the existing system. We then discuss their

main features that can be exploited in the rule generalisation step presented in detail in Chapter 4.

3.3.1 Aspect Oriented Programming

Aspect Oriented Programming (AOP) is a programming paradigm that complements Object Oriented Programming (OOP) in modularising crosscutting concerns such as observation and logging (Laddad 2009). It can be used as a tracing approach to extract behaviours of object structures at runtime.

(*AspectJ* 2016) is a well-known AOP language that facilitates the creation of a special class called *aspect* to encapsulate crosscutting concerns. It is Java-based and supports AO concepts such as *pointcut*, *join points*, and *advices* (Laddad 2009), explained as follows:

- A join point is a place where the crosscutting actions take place in the program execution flow. This place can be a method call, a method execution, a field access or an initialisation of an object, etc. It can be controlled with an *aspect* class by specific variables such as *thisJoinPoint*.
- A pointcut is a program declaration that expresses selecting join points and collects context at those points. For example, when selecting a join point that specifies an execution of a method, it collects contexts such as the `this` object and the arguments to the method.
- An advice allows to implement crosscutting behaviours to be executed at each join point reached as well as *before*, *after* or *around* join points.

In this setting, each join point allows to observe an internal transformation step that occurs on a specific object, representing a basic action. Here, we are interested in composing sequences of basic actions, observed by *before* and *after*

advices for each join point reached. The purpose is to construct a single rule that represents the behaviour of a certain operation.

3.3.2 Generating Logs

The initial requirement to generate logs are to select Java classes (e.g., from the class diagram) and methods to be invoked by test cases. We assume these are to be defined before the tracing process takes place, to identify the objects of interest at runtime.

In order to generate logs by controlling what actions on which objects to react to, we declare a *pointcut* expression that matches all kinds of join points without any restriction, see Listing 3.2. This means we are able to record all actions that test or change existing objects and fields or create new objects.

As the *advice* implements the observation mechanism to be executed before and/or after each join point, it enables us to trace state changes by comparing the values obtained from the execution of the actions.

```
// Pointcut declaration without restrictions, allowing to observe the execution of the all
  kinds of internal states
pointcut stateTriggers(): !within(Tracing.*);
```

LISTING 3.2: The declaration of our Pointcut

Observing all actions that involve read or write access to any part of an object, including invocations and executions, we produce a large number of join points. These are filtered by the classes defining the scope of our observation, so that we only record join points relating to instances of these classes. Such classes can be defined directly from the source code using the ObjectAid tool[2].

The result is a sequence of nested join points as shown in Listing 3.3, tracing the *registerClient(..)* operation of the Car Rental case study (cf. Section 3.2) and

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

```
1  // Tracing registerClient(..) operation from Car Rental Service API
2   { public String registerClient(String city, String clientName); }

3  // The following is a sequence of nested joinpoint outputs from tracing the above
     operation
 ...
4       1 before execution(String RentalService.Rental.registerClient(String, String))
5          2 before call(Branch RentalService.Rental.getBranch(String))
6             3 before execution(Branch RentalService.Rental.getBranch(String))
7                4 before & after get(Branch[] RentalService.Rental.branches)
8                5 before & after get(String RentalService.Branch.city)
9             3 after execution(Branch RentalService.Rental.getBranch(String))
10         2 after call(Branch RentalService.Rental.getBranch(String))
11         6 before initialization(RentalService.Client())
12            7 before & after initialization(java.io.Serializable())
13            8 before & after execution(RentalService.Client())
14         6 after initialization(RentalService.Client())
15         9 before & after set(String RentalService.Client.cName)
16         10 before & after get(String RentalService.Branch.city)
17         11 before & after call(String java.lang.String.valueOf(Object))
18         12 before & after set(String RentalService.Client.cID)
19         13 before & after get(String RentalService.Branch.city)
20         14 before & after get(ArrayList RentalService.Branch.ofClients)
21         15 before & after set(String RentalService.Client.cID)
22         16 before & after get(ArrayList RentalService.Branch.ofClients)
23         17 before & after call(boolean java.util.ArrayList.add(Object))
24         18 before & after get(String RentalService.Client.cID)
25      1 after execution(String RentalService.Rental.registerClient(String, String))
...
```

LISTING 3.3: Sequence of nested join points obtained by tracing

selecting all the classes in Figure 3.4 (A). Depending on the action performed, join points translate into basic inner rule instances describing the relevant state transformations. The idea is to aggregate these rule instances into a single rule as presented in Figure 3.4 (C).

3.4 Constructing Rule Instances

We analyse traces at runtime once the invocation of a method of interest has completed. In order to aggregate the rule instances for a sequence we identify elements across the sequence that refer to the same objects. Then the basic

actions are composed along these shared objects to form the overall contract instance. This is done using object identifiers, which provide a unique and stable handle on all objects.

We exploit four types of AspectJ advices to observe the accessed objects and their changes: *before*, *after*, *after returning* and *after throwing*. Figure 3.6 describes the overall process of analysing a sequence of join points, see lines 4 to 25 in Listing 3.3. We restrict the relevant join points based on the selected classes, shown under (1) of Figure 3.6. This is to define the scope of the operation as the set of objects potentially affected and match them to the objects that have actually been accessed based on the join points. These are the elements of the rules constructed as a result. We discuss these steps in more details in the following subsections.

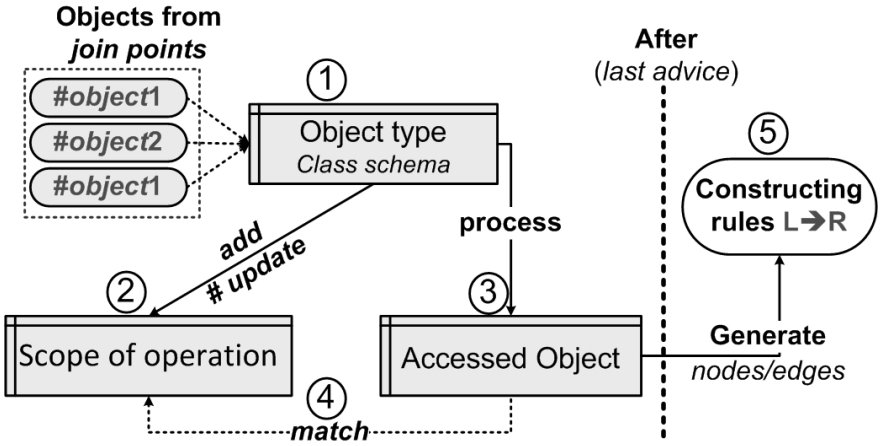


FIGURE 3.6: Overview of our trace analysis

3.4.1 Scope of Operation

The scope of an execution contains all objects potentially needed for the construction of the rule, see Figure 3.6 (2). It is defined by navigating *this()* and *target()* references from each relevant join point, recording also the values of attributes that may change during execution.

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

LISTING 3.4: Operation *enumerateAttributeNames()* from (*NanoXML - a small non-validating XML parser for Java* 2016)

In addition, the scope prevents us from considering unnecessary objects, which are of the right class but unrelated to the trace. For example, Listing 3.4 shows the query method *enumerateAttributeNames()* which does not affect any member of its object but writes to a local *Vector* object created in line 2. The same class exists in the class diagram, but write access to the local *Vector* object is not relevant to the rule to be extracted, i.e., rule from *enumerateAttributeNames()* .

3.4.2 Accessed Objects

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

There are two explicit join point types for handling read and write access at the field level: the *get-field* and *set-field* join points. In lines 7 and 15 of Listing 3.3 we show examples of read and write access, respectively. In contrast, access at the object level is implicit in the *method call* join point, where it needs specific restrictions to deal with collection object operations. For example,

adding elements to a collection represents write access, see line 23. Analogously, the calls in the following example require the execution to read the elements of the collection.

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

We rely on Java object identifiers to find the accessed objects in the scope as illustrated in Figure 3.6 (3).

3.4.3 Cases to Construct Rules

Once the execution of the last advice has completed, see line 25 in Listing 3.3, we construct a rule $r : L \rightarrow R$ by building up graphs L and R by navigating the accessed objects. This is described in Figure 3.6 (5).

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

TABLE 3.1: Main cases for rule construction

Table 3.1 shows the cases used for constructing rules. For instance in the first row, the *initialization* join point indicates the creation of a new object. Accordingly, we add a new vertex to R. The second row indicates the creation of an edge, which means that the necessary source and target vertices must be added to both L and R. We translate any object attribute *variable-in-class* that points to another object and has a valid type in the class diagram into an edge. The initial value of an *object attribute* will be added to L and the last value of a relevant write access to R.

In the last column we state for each element if it is minimal (required for the specification of the effect) or context (shared between pre- and postcondition) based on the access type. Note that due to the semantics of Java, there is no join point type for destroying an object, as the garbage collector automatically destroys objects that are not reachable by any reference. This is one of the technical limitation of the approach, but as long as the main application of our technique is in program understanding, this aspect of Java's semantics is reflected correctly in the extracted contracts.

3.4.4 Information on Accessed Objects

Property	**Description**
node type	a defined object type, valid in the class diagram
isThis	current object that includes main observed operation, i.e, the execution of operation of interest begins from it
isReturn	true if the object is the returned parameter from observed operation, false otherwise
isParameter	specify if the object (node) is one of the passed object parameters
isMinimal	true if the object is part of minimal rule elements, context element otherwise
isInitialised	true only with created objects, false otherwise
isCollection	specify if the object is a container type
no. step	steps numbers in the path of accessed objects
access info	list of accesses with type (read/write), values and locations details, including class names and line numbers in the code (in the case of tracing source code, not the byte or compiled code)

TABLE 3.2: Extracted properties for each object (node element) in the rule

Table 3.2 describes the main properties extracted from each accessed object, represented as a node in the rule. These properties are useful for program understanding, e.g., as part of testing or debugging. We use some of them in defining node signature needed for learning and generalising rule instances, discussed next in Chapters 4 and 5.

3.5 Summary

This chapter presented a dynamic approach to extract, for each operation invocation, a version of *a visual contract* capturing the observed behaviour. We explored the proposed strategy and the notion of a visual contract that we want to reverse engineer from Java implementations. We base our discussion on behavioural Java code and UML class/object models illustrated by a case study of a Car Rental Services, see Section 3.2. We explained that representing a contract for a given operation requires:

- determining the type of classes used for implementing operations and
- determining accurately the changes of object and data states.

These two requirements made clear for us that both static and dynamic analysis techniques are important in order to infer visual contracts. The first requirement is practically achievable by static analysis tools (e.g., ObjectAid[3]). Observing object and data states need dynamic analysis.

Our observation technique is made by weaving instrumentation code using AspectJ. They result in a trace recording object creation, read and write access to objects and attributes, etc., caused by the invocation. We filter out irrelevant objects and aggregate the basic observations into a contract instance capturing the overall precondition and effect of the execution.

Along with each constructed instance we collect traceability data for each element of the contract, such as the line numbers in the code responsible for each access. These traceability data can be generally used for program understanding, e.g., as part of testing or debugging.

[3]ObjectAid UML Explorer based on the OMG's UML 2.0 specification (*ObjectAid UML Explorer for Eclipse* 2014) that aims to extract Class Diagram: `http://www.objectaid.com/class-diagram`

Moreover, each extracted rule (contract instance) describes one possible behaviour of the operation, covering only the part executed by one test case. Therefore, we usually obtain more than one rule for a specific operation, which then need to be combined or generalised. In the next Chapter 4, we introduce our proposal for generalising rule instances in order to obtain a general specification of the operation's behaviour.

Chapter 4

Generalisation of Contract Instances

In this chapter, we introduce the second step of our methodology, published in (Alshanqiti, Heckel, and Khan 2013). It aims at inferring visual contracts as interface specifications for the operation from sets of contract instances, i.e., pairs of graphs representing transformations. These contract instances are represented by typed graph transformation rules (Ehrig et al. 2006), extracted by tracing the execution of test cases as discussed previously in Chapter 3.

Constructed contract instances can be classified by their effects as described by a minimal rule (cf. Section 4.1.2). Then maximal rules (cf. Section 4.1.3) are derived that generalise the preconditions of all instances sharing the same effects, i.e., keeping only context that is always present when the relevant minimal rule is applied. These two rules can assist developers to understand the range of object behaviours, with maximal rules representing the best candidate to describe the functionality of the system.

To tackle the possibility of generating dynamically incomplete behaviour models with respect to the actual implementations (Cornelissen et al. 2009), our solution allows to improve completeness by incrementally including new behaviours when observing additional contracts.

The following sections are organised as follow: In Section 4.1, brief definitions of contract instances, minimal and maximal rules are given. In Section 4.2, the inference approach is introduced with graph matching algorithms and an example demonstrating the construction of maximal rules. In Section 4.3, an incremental inference technique is explained, which aims to improve completeness of the extracted models.

4.1 Basic Definitions

Abstractly speaking, we are interested in extracting rules from contract instances representing transformations (cf. Chapter 3). Following the running Example 3.2 and the basic definitions of graph transformation rules introduced in Section 1.3.2, we assume a type graph TG and set of rule names P, and for each $p \in P$ a set $\xrightarrow{p}\subseteq \mathbf{Graph}_{TG} \times \mathbf{Graph}_{TG}$ of successful transformations. The original rules are not known but the aim is to define, for each rule name $p \in P$, a rule $\pi(p)$ such that $\xrightarrow{p}\subseteq\overset{p}{\Longrightarrow}$, i.e., the given examples are part of the rewrite relation.

4.1.1 Contract Instances

Considering Figure 3.4 (C), a contract instance consists of a pair of UML object graphs representing the situation before and after the operation. We write $b = op(a_1, \ldots, a_n) : G \Rightarrow H$ to indicate the invocation $op(a_1, \ldots, a_n)$ of an operation with signature $op(x_1 : T_1 \ldots, x_n : T_n) : T$ leading to a transformation of G into H. We assume that G, H live in a common name space given by unique object identities, so the elements deleted, preserved and created by the transformation are $G \setminus H, G \cap H$ and $H \setminus G$, respectively.

4.1.2 Minimal Rules

A minimal rule $r = min_{op}(i)$ is a subrule of the rule instance i it is constructed from. It is able to perform a given transformation without considering context elements, i.e., the same elements are created and deleted, but r is stripped of unnecessary context. In Figure 4.1 (A), we give an example to differentiate between minimal rule elements ($r2$ and $c1$) and context elements ($this : Rental$ and $r1$) that remain without change in both graphs.

4.1.3 Maximal Rules

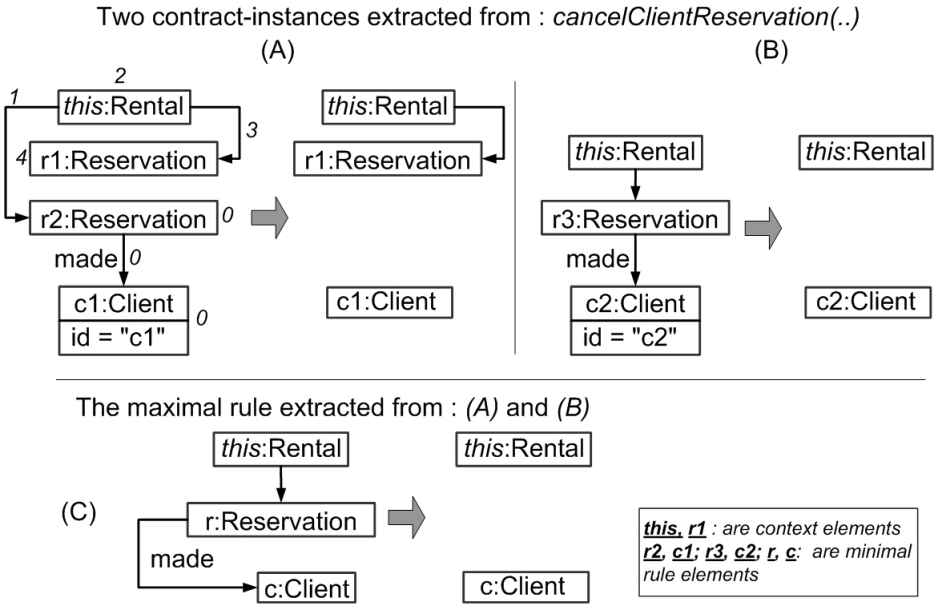


FIGURE 4.1: Inferring maximal rules from contract instances, see Section 3.2

While the minimal rule captures the shared effect of all instances, it does not provide a common precondition. The latter is obtained in the maximal rule by including all context that is present with all instances.

Consider the two pairs of graphs given in Figure 4.1 (A) and (B). The matches for their minimal rules are given by nodes $r2, c1$ and $r3, c2$ respectively. The intersection of their contexts excludes the Reservation node $r1$, while preserving

the Rental node *this*. The maximal rule shown in (C), as inferred from these two examples, is therefore isomorphic to the second example.

4.2 Inference Approach

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

The example in Figure 4.1 illustrates our solution. Under (A) and (B) we show two generated contract instances, representing two pairs of graphs. The inference algorithm analyses these graphs with the purpose of discovering a rule approximating the one in (C).

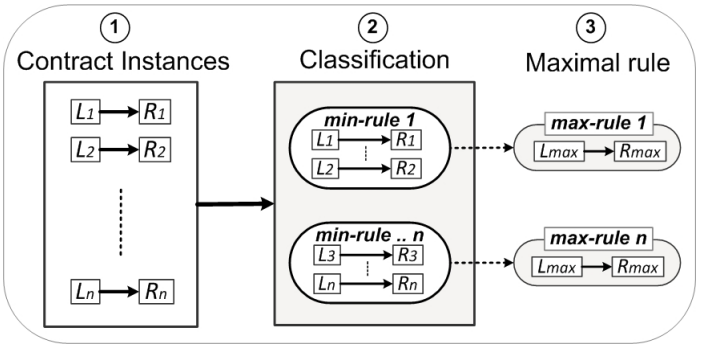


FIGURE 4.2: Maximal rules inferred from contract instances, extracted from a particular operation

The process of generalising a given set of contract instances is performed in two steps as described in Figure 4.2. The first step, from (1) to (2), classifies all contract instances into distinct sets based on their minimal rules. In the second

step, from (2) to (3), we infer a single maximal rule from each classified set. We explain these two steps in detail in the next Section 4.2.1 and Section 4.2.2.

4.2.1 Classification of Contract Instances

We consider all contract instances representing executions of the same operation, e.g., the set of instances under Figure 4.2 (1). Then, we extract a *minimal rule* from each instance, i.e., the smallest rule containing all objects referred to by the operation's parameters and able to perform the observed object transformation. The construction has been formalised in (Bisztray, Heckel, and Ehrig 2009) and implemented without considering parameters in autocite2013AlshanqitiLearning.

Formally, given a contract instance $b = op(a_1, \ldots, a_n) : G \Rightarrow H$ its minimal rule is the smallest rule $L \Rightarrow R$ such that $L \subseteq G, R \subseteq H$ with $a_1, \ldots, a_n \in L$ and $b \in R$ as well as $G \setminus H = L \setminus R$ and $H \setminus G = R \setminus L$. That means, the rule is obtained from the instance by cutting all context not needed to achieve the observed changes nor required as input or return.

The proposed algorithm for classifying contract instances is presented in Algorithm 1. Here, the algorithm takes a set of pairs of UML object graphs as an input and then produce a classification based on shared effects as shown under Figure 4.2 (2).

In order to handle efficiently large (numbers of) UML object graphs, we use a relational database (MySQL), see *line 1*. This provides the means to formulate complex operations on graphs as declarative queries. Similar motivations have led to the use of relational databases in graph transformation before, e.g., in (Varró, Friedl, and Varró 2006). *lines 2-9* sketch the process of extracting the minimal rules using unique object identities to match graphs and then classifying all instances into sub-groups.

Algorithm 1 Classifying contact instances based on minimal rules

Inputs: CIs [G,H] is a set of *Contract Instances*, i.e., a list of pairs of UML object graphs
Outputs: setMRule [minRule[G,H]] is a list of sets of *Contract Instances* that share the same effects.

Begin
1: **initialise** *CIs* in a database for processing
2: **set** setMRule= $\emptyset$
3: **for each** *G and H* **in** *CIs* **do**
4: **get** the minimalRule from $G \Rightarrow H$
5: **for each** mRuleGroup **in** *setMRule* **do**
6: **if** mRuleGroup.*minRule* isIsomorphismMatching (minimalRule)
such that *minimalRuleG* $\rightarrow$ *LHS of minRule*
and *minimalRuleH* $\rightarrow$ *RHS of minRule* **then**
7: mRuleGroup.*minRule*.**add(**minimalRule**)**
8: break
9: **add** *new mRuleGroup* in *setMRule* **then** add(minimalRule)
End

4.2.2 Rule with Shared Contexts

Each classified set as shown under Figure 4.2 (2) represents instances that have the same minimal rule (i.e., the same effect), but possibly different preconditions (i.e, different context elements). Here, our aim is to generalise all instances from each set by one so called *maximal rule*. This rule extends the minimal rule by all the context that is present in all instances, essentially the intersection of all its instances' preconditions, see Figure 4.2 (3).

Figure 4.1 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).

In fact, minimal or maximal rules are not just generalisations of instances, but provide a constructive specification. Given an object graph G, a rule can be applied if there is a match $m : L \rightarrow G$, such that L is (isomorphic to) a subgraph of G and removing (an image of) $L \setminus R$ from G, the resulting structure is a graph. The derived object graph H is obtained by adding a copy of $R \setminus L$. Unsurprisingly, applying a rule extracted from a contract instance $b =$

$op(a_1, \ldots, a_n) : G \Rightarrow H$ to the pre-graph G of that instance, we obtain its post-graph H, but we can also apply the same rule to other given graphs deriving transformations not previously observed.

4.2.2.1 Inferring Maximal Rules

We propose Algorithm 2 to infer a single maximal rule from each classified set, produced by Algorithm 1. The idea is to identify the intersection of all elements in pre-graphs, allowing to add shared context elements, i.e., exist in all instances, to the left-hand side of the minimal rule.

We initialise the computation of intersections using the smallest pair of graphs, called *MAR* for Maximal Abstract Rule, which is obviously an upper bound for the intersection, see *line 3* in Algorithm 2. In order to make graph elements distinguishable during matching, we define a signature for each node element. The signature is a collection of node type, connected incoming and outgoing edge types, *distance* and *abstract id*. The distance is the shortest path to an element in the minimal rule. In Figure 4.1 (A) this is indicated by the numbers in the first pre-graph. The abstract id is a unique identifier in MAR.

Discovering the intersection of the additional contexts of contract instances in $minRule[G, H]$ is carried out in *lines 4-10*. The difficulty at this point arises when matching contexts that have similar signatures. Figure 4.3 describes such a situation. Assuming the process is at distance (6) and all nodes and edges are identical based on their signatures, it is unclear which node should be removed to arrive at the accurate intersection. We overcome this problem by marking the contexts and leaving the decision to the step of updating MAR after finishing the second loop in *line 10*. In Table 4.1, we explain all the situations with the decisions taken by the algorithm.

Algorithm 2 Inference of maximal rules

Inputs: minRule [G,H] is a set of *Contract Instances* that share the same effects
Outputs: maxRule is a single maximal rule

Begin
1: **load** *minRule* from the database for processing
2: **set** the distances for each node(s) and edge(s) v ∈ **G** in *minRule*
(set *distance*=0 for all minimal rule elements)
3: **let** MAR *Max-Abstract-Rule* = the smallest pair of graphs in *minRule*
4: **for each** *G and H* **in** *minRule* where *G and H* ≠ MAR **do**
5: **for each** *v* **in** *LHS of MAR* (ascending order by distances) **do**
6: **if** v.*distance* >0 **then**
7: **discover** the intersections of the contexts between *L* of *MAR* ∩ *G*
8: **set** *r* = the possibility of deleting contexts in MAR that are out of the intersection
9: **if** *r*=1 **then** *remove context* **Otherwise set** *r++* **and** mark the contexts
10: **update MAR()** *delete and update the context(s) that has been remarked in* **line 9** (in a descending order by distances).
11: **set** maxRule= MAR
End

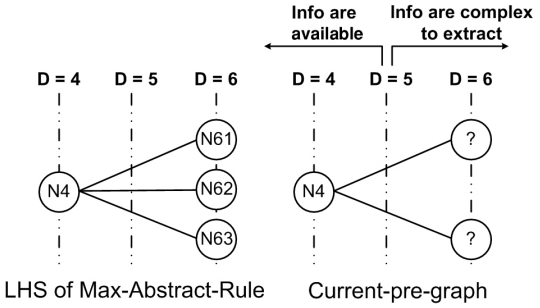


FIGURE 4.3: Complex decision to define the accurate intersection

The process of cutting down unnecessary contexts while updating *MAR* as in *line 10* must always preserve the validity of the graph structure. Deletion is performed in a descending order, i.e., from maximum distance back to the elements of the minimal rule at *distance=0*. For example, nodes that are marked to be deleted ($r \geqslant 1$) and still connected to at least one required incoming or outgoing edge must be preserved. Consider the example given in Figure 4.4. It supposes that the process is at distance (8), as illustrated in the right snapshot, and only one node of either *(N81, N82 and N83)* needs to be removed safely without damaging the structure of the graph as their $r = 1$. Mark r indicates the number of required deletions for a particular node signature at a specific distance.

While it is possible to choose either *(N82 and N83)* and then update r to be $r = 0$, node *N81* cannot be the one because it has a required edge linked with

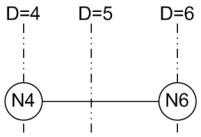
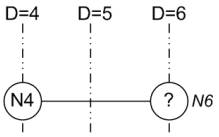
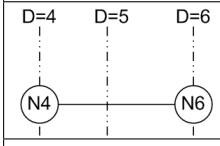
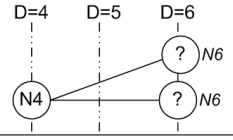
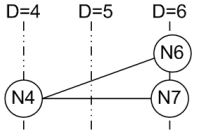
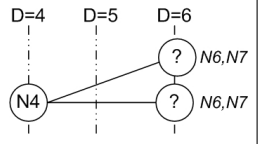
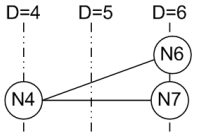
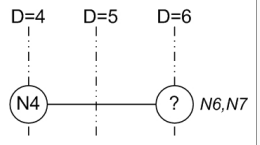
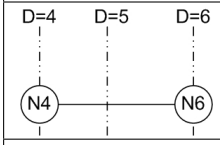
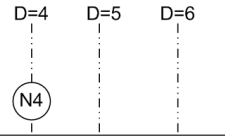
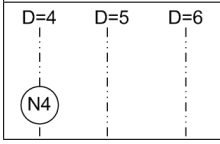
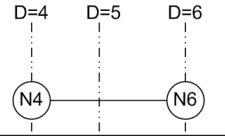
Left-hand side of MAR	Pre-graph of current instance	Decision
Following the example described in Figure 4.3, the aim is to find the matches while the process at distance (6). Assuming the node *N4* has been set as a one-to-one		
		*Obvious case*: the abstract id (N6) will be set in the current-pre-Graph as a one-to-one match.
		*Obvious case*: the abstract id (N6) will be set in the all identical nodes.
		*duplicated case*: the abstract ids (N6 and N7) will be set in the all identical nodes
		*Difficult case*: the abstract ids (N6 and N7) will be set as many-to-one and their marks r for will be increased (+1), which indicate the required number of deletion, such that $N6.r = 1$ and $N7.r = 1$.
		*Obvious case*: the abstract id (N6) will be removed immediately from the MAR
		*Obvious case*: (N6) will not be considered as it does not exists in the MAR.

TABLE 4.1: Cases of matching elements in MAR with elements in a contract instance

N101. The decision in the left snapshot, i.e. in the next distance at (7), becomes obvious as the remaining only one option which is the deletion of *N62*.

In case that there is more than one option with not enough evidences to delete, the algorithm will preserve the context without deletion. As a result, the relationships between minimal, maximal and the projected original $\pi(p)$ is : $\min(p) \subseteq \pi(p) \subseteq \max(p)$.

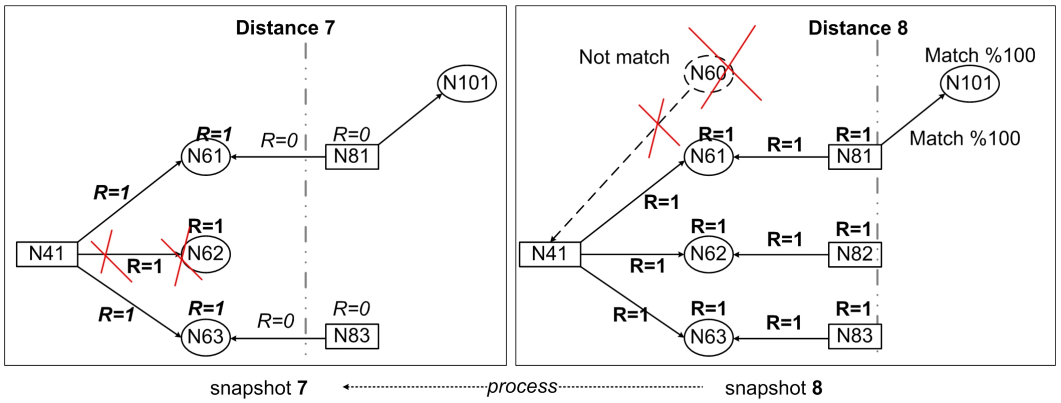


FIGURE 4.4: Two snapshots explaining the process of updating MAR

4.2.2.2 Complexity of the Construction

Maximal rules are not unique, not even up to isomorphism. Consider the example in Figure 4.5 showing a modified model of a Rental Agency Rt with two branches Br to which vehicles of types Car, Tr (*for trucks*) and SUV are allocated. Taking I_1 and I_2 as the left-hand sides of two given instances (and assuming the left-hand side of the shared minimal rule to be given by Rt only), both M_1 and M_2 are maximal rules for this set. They represent different generalisations of the two cases.

Intuitively, we could read M_1 as part of a requirement that each branch should be equipped with vehicles for both passengers (Car and SUV) and goods (Tr and SUV), while M_2 could be part of a weaker requirement stating the same for the rental agency Rt as a whole. Both requirements are satisfied by both instances, but they are mutually unrelated and there is no unique way of merging I_1 and I_2 such that the result retains a subgraph isomorphism to both I_1 and I_2.

This is related to the more general problem that graphs and subgraph isomorphisms do not form a lattice, causing complications, e.g., for the concept-based mining of graphical data (Ganter et al. 2004) where a concept has to be represented by a set of graphs rather than a single graph pattern. In our case, this approach would require to extract all maximal rules from a set of given

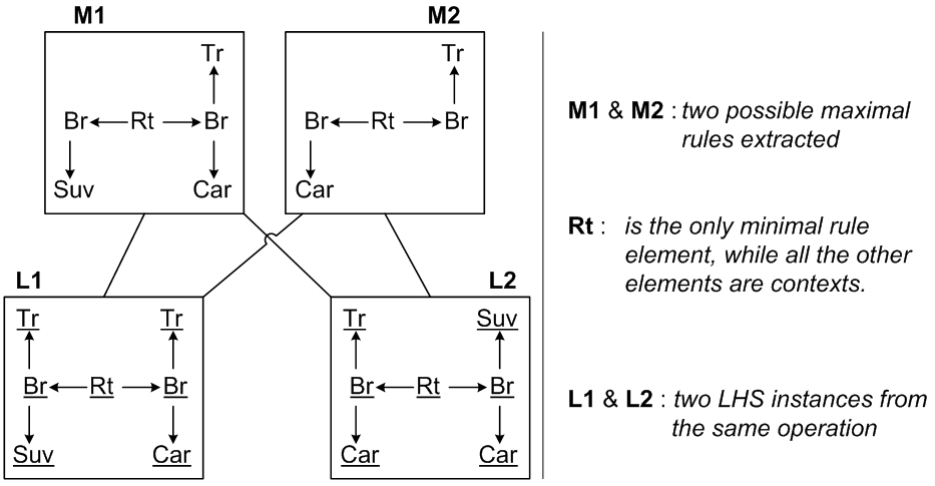


FIGURE 4.5: Maximal shared subgraphs are not unique

rule instances. Apart from being significantly more expensive computationally, this solution would result in a set of maximal rules, each one generalising all instances and the interpretation of them is unclear.

We have instead opted for a test that creates a warning in case a maximal rule is not unique. In our practice based on rule instances extracted from observing executions of real code we have not yet encountered this situation. In fact, analysing the example, it seems that the problem lies in the presence of two nested unordered containers, i.e., Rt carrying a set of Br nodes and each Br containing a set of vehicles.

As explained, Algorithm 2 aims to produce a single maximal rule from a set of instances that share the same effects (i.e., in this case from $L1$ and $L2$). This means the inferred maximal rule should be either $M1$ or $M2$. Generally, such inference by our algorithm depends on (1) which instance comes first in the list *minRule*, after sorting all instances, and then on (2) the selection of MAR, i.e., located in the top of the list, see line 3 in Algorithm 2.

Using the given complex example in Figure 4.5, Algorithm 2 will always produce a rule like $M1$. In either case of selecting $L1$ or $L2$ to be a MAR, the algorithm

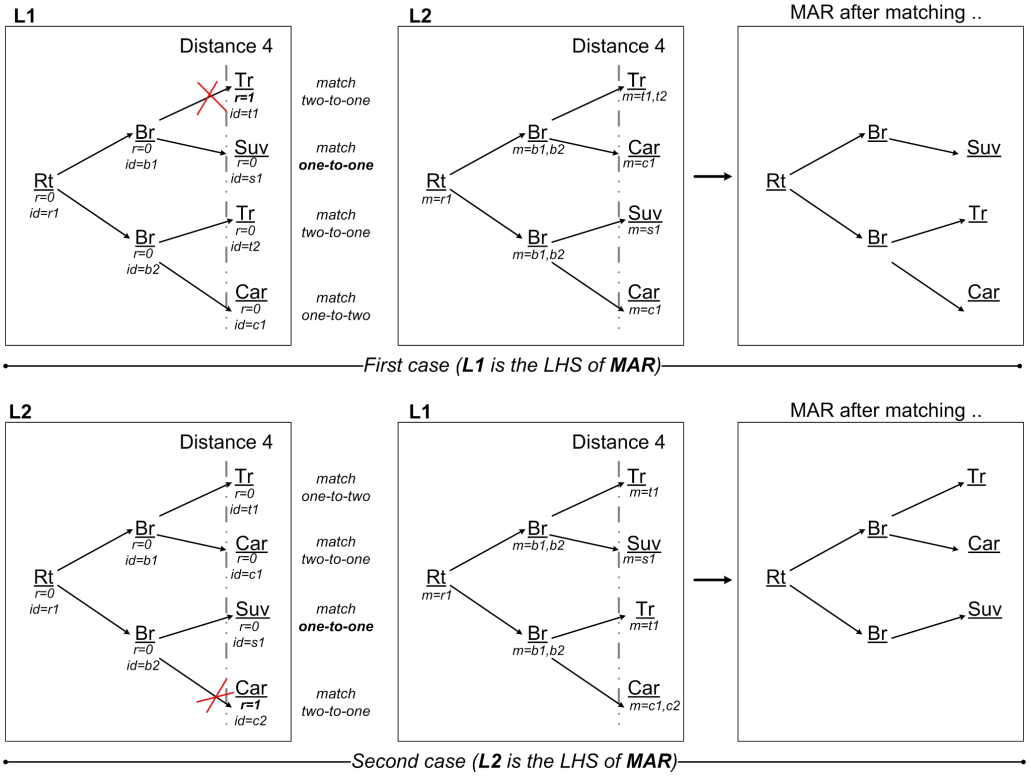


FIGURE 4.6: Applying the complex example (Figure 4.5) in our algorithm

cannot produce something $M2$. This is because, the process of matching elements, particularly when updating MAR in line 10, always starts with certain cases (i.e., one-to-one match and $r = 0$, such as matching Rt or Suv nodes) and then uncertain cases, including $r = \{1, 2, \cdots n\}$. In other words, considering Figure 4.6, while the process is at $distance = 4$, the algorithm will confirm matching the Suv node first for both cases, as this node has a one-to-one match and its $(r = 0)$. Based on this decision, inferring $M2$ that does not include Suv node is not possible.

4.3 Incremental Inference

Using a relational database for recoding traces and/or contracts not only allows to manage and maintain large contracts, but also to implement a continuous

```
    // get a single instances that still not being classified in any group ..
    CachedRowSetImpl crsGetSingleRuleInstances= DBRecord.getByQueryStatement(
        "select "
        + "TblBasicRule.MethodSignatureUniqueID, "
        + "TblBasicRule.Observation_IDREFF, "
        + "TblGraph.GraphID "
        + "from TblBasicRule INNER JOIN TblGraph "
        + "on TblBasicRule.Observation_IDREFF= TblGraph.Observation_IDREFF "
        + "where TblGraph.graphType=false "
        + "and TblBasicRule.groupID is null "
        + "and TblBasicRule.hasEffect=true "
        + "order by TblBasicRule.MethodSignatureUniqueID limit 1;", true);
}
```

LISTING 4.1: SQL statement to fetch any unclassified contract instance

learning mechanism. Rather than running the algorithm each time over all contract instances from scratch, to include new behaviours of instances not observed before, we store all the needed information to incrementally build visual contracts.

The SQL code shown in Listing 4.1 is used to fetch any new contract instance not classified yet, i.e., $groupID = null$. The Algorithm 1 will be triggered whenever an unclassified contract instance (say ci) is fetched, attempting to allocate it in the right group rg of existing minimal rule sets. Accordingly, by running again Algorithm 2, the latest MAR inferred from rg will be updated by matching it only with the new instance ci.

4.4 Summary

The main concern of this chapter was to discuss our approach for generalising sets of contract instances, i.e., pairs of graphs representing transformations into more concise and comprehensive rules, increasing understandability.

We have discussed the formal definitions of minimal and maximal rules that are produced by our algorithms. These rules are very useful when simplifying

the representation of many and potentially big instances into a smaller rules describing precisely the actual effects and common contexts.

We have not yet come across a real example that shows the need of constructing more than one maximal rule from a set of contract instances sharing the same minimal rules. As a consequence, we limit the proposed algorithms to generate only a single maximal rule from each set of rule instances.

In the next chapter, we will discuss a further generalisation by introducing multi objects and the derivation of logical constraints over attribute and parameter values.

Chapter 5

Inference of Advanced Rule Features

This chapter presents the third step of our methodology focusing on inferring advanced features to enhance generalised contracts (Alshanqiti and Heckel 2015). In two independent steps

- we raise the level of abstraction by inferring rules with universally quantified (multi) objects from generalised contracts (maximal rules).
- we enhance contract descriptions at the level of maximal rules by inferring conditions on parameter and attribute values.

The inference of multi objects supports the construction of concise and comprehensive rules and increases understandability. Attribute constraints support the analysis of tests based on a visual representation of operations' behaviour. We discuss the inferences of these features by dividing this chapter into two sections: inferring multi objects in Section 5.1 and deriving attribute/parameter constraints in Section 5.2.

5.1 Inferring Universally Quantified Multi Objects

The contracts extracted as presented in Chapters 3 and 4 may use a number of rules to describe the same operation. In the case of iteration over containers, for example, the set of minimal rules is potentially unbounded, but many may only differ in the number of objects manipulated while performing the same actions across all of them. Rules with multi objects (MOs) provide a concise way to specify constraints and actions across a set of objects of potentially unknown cardinality.

5.1.1 Definition of GT Rules with Multi Objects

A multi-object node represents a set of nodes in an instance graph the rule is applied to and carries a cardinality constraint for that set. Consider the illustrative example Figure 5.1 where the two patterns shown have two or three occurrences of node (: O) with identical context. They lead to the MO node (shown with a 3D shadow) inferred on the right, with the cardinality 2..3

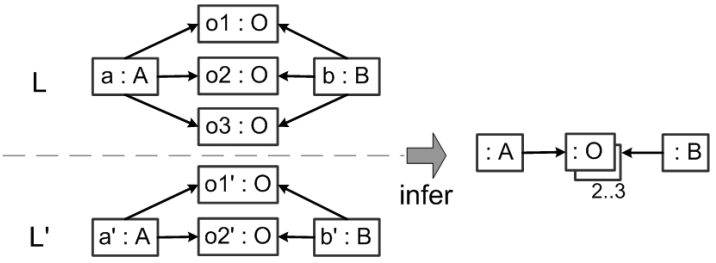


FIGURE 5.1: Inference of multi-object nodes

Following the formal definitions of graph transformation rules introduced in subsection 1.3.2, a *multi-object (MO) rule* $mr = (L \Rightarrow R, M, card)$ is a rule with a set $M \subseteq (L_V \setminus L_D)$ of MO nodes and cardinality constraints $card : M \rightarrow$

$(\mathbb{N} \cup \{*\}) \times (\mathbb{N} \cup \{*\})$. It states how many concrete objects each MO can be instantiated by. The set of MO rules over TG is $MRule_{TG}$.

Application of MO rules is defined by expanding MO nodes into sets of regular nodes. *Expansions of mr* are all (regular) rules obtained by successively replacing each MO node $m \in M, card(m) = (u, v)$ by $c(m)$ copies for some chosen $u \leq c(m) \leq v$. This includes copying all incoming and outgoing edges so that for each node $m \in M$ and chosen $c(m)$ we get L^m as

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

where $L_E(m) = \{e \mid src^L(e) = m \vee tar^L(e) = m\}$ is the set of edges attached to node m. Sources, targets and types of new edges and nodes are inherited from L.

The expansion extends to R on the MO nodes shared with L. Due to the associativity of the product $\times$ up to isomorphism, the resulting rule is essentially independent of the order of the MO nodes expanded. Note that for two MO nodes m_1, m_2 connected by an edge we will create $c(m_1) * c(m_2)$ edges between the copies of m_1 and m_2. An *application of an MO rule* to an object graph G is an application of a maximal applicable expansion.

5.1.2 Approach

The running example in Section 3.2 has two operations that could produce rules with multi objects. One of them is *showClientReservations(..)*. For example, node *Reservation* in Figure 5.2 (C) is an MO node with cardinality (1..2), applicable to object graphs with 1 or 2 *Reservation* nodes connected to the *Client*. The contracts (regular rules) of two corresponding transformations are shown in (A) and (B).

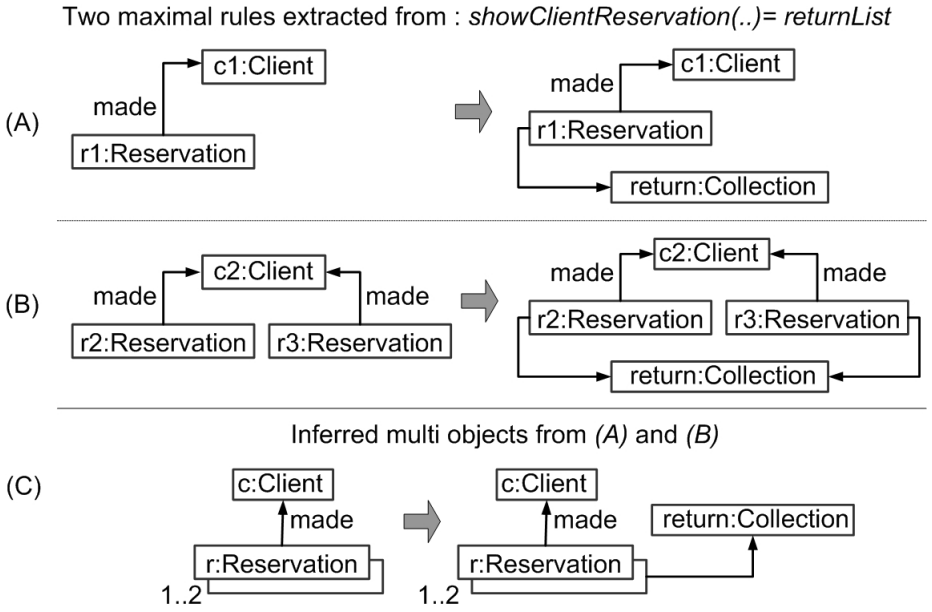


FIGURE 5.2: Inferring rule with MO from *showClientReservation(..)*

In order to derive MO rules from such regular rules, we have to discover sets of nodes that have the same structure and behaviour, then represent them by a single multi-object node. Here, 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 5.2 (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 5.2 (C), which also generalises (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 (1) are 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 (and thus specify

the same actions). Assuming for every operation *op* a set of maximal rules $R(op)$ as constructed in Chapter 4, we derive MO rules from two perspectives.

Merging 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 $MOR(op)$.

Combining isomorphic rules. A maximal set of structurally equivalent rules in $MOR(op)$, differing only in their object identities and cardinalities of their MO nodes, forms an isomorphism class. For each such class we derive a single rule by selecting a representative MO rule and assigning to each of its MO nodes the union of cardinalities of corresponding nodes in all the rules in the class. The resulting set of combined MO rules is $CMOR(op)$. An example is the derivation of Figure 5.2 (C), a combination of basic maximal rule (A) with the MO rule derived from (B) whose cardinalities of 1 and 2 for the *Reservation* node are merged to 1..2.

5.1.3 Algorithm for Inferring Multi-Objects

Isomorphic MO rules can be inferred from rules extracted from the same operation but with different maximal rules. The number of rules captured by an MO rule, as well as the number of nodes represented by each MO node are both indicators of the confidence (more instances captured leading to a higher level of confidence) and its effectiveness in the sense of significant abstraction. Our solution allows domain experts to define the minimum number of nodes required to represent a MO node on which their confidence levels can be measured.

Algorithm 3 Inferring multi objects (MOs) from a given rules

Inputs: rule $L \Rightarrow R$ and $iConf$ is a minimum number of nodes required to represent a MO node.
Outputs: modified rule $L \Rightarrow R$

```
Begin
 1: for each node v1 in L ∪ R do
 2:     if v1 ≠ null then
 3:         set v1.isMultiObject = false
 4:         set v1.card = 1
 5:     else
 6:         continue
 7:     for each node v2 in L ∪ R do
 8:         if v1 ≠ v2
            and v1.type == v2.type
            and v1.isMinimal == v2.isMinimal == true
            and isContextMatched(v1, v2) == true
            and isEffectMatched(v1, v2) == true) then
 9:             set v1.isMultiObject = true
10:             set v1.card = v1.card + 1
11:             if v1.card < iConf then
12:                 continue
13:             removeFrom(L, v2)
14:             if v2 ∈ R then
15:                 removeFrom(R, v2)
End
```

The pseudo code shown in Algorithm 3 gives a high-level view of our algorithm for extracting multi-object rules. For example, *line (8)* explains the main conditions used to confirm if two given nodes are structurally and behaviourally matched, which are then used to infer multi-objects. *line (11)* checks the number of occurrence (of nodes confirming the conditions in *line (8)*) with $iConf$ that is specified by domain experts. The Java implementation of the algorithm can be found in our repository[1].

5.2 Deriving Constraints on Attribute and Parameter Values

So far we have focussed on structural preconditions and effects for generalising visual contracts, disregarding the data held in objects' attributes or passed as parameters. At implementation level, manipulation of object structure and data are tightly integrated. However, at model level, a distinction is visible through

[1] `https://github.com/AMahfodh/IGTRRep/tree/AMaster/IGTR`

the use of associations for object-valued fields vs. attributes for data-valued fields but even here contracts cover both structural and data constraints and actions.

While we have seen that the structural view is naturally expressed by graphical patters, constraints or assignments over basic data types are more adequately expressed in terms of logic. In this section we explain how visual contracts handle attribute and parameter manipulation, and how their constraints can be learned from information extracted from the structural analysis described.

5.2.1 Overview of Learning Invariant Constraints

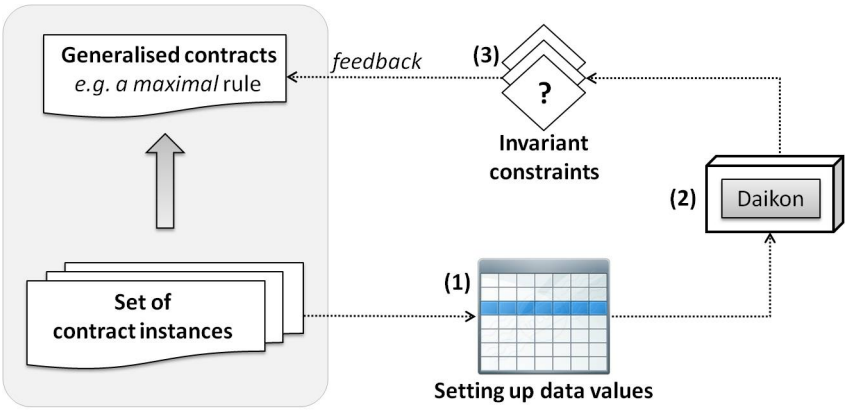


FIGURE 5.3: Process of learning constraints on attributes and parameters

Figure 5.3 illustrates the process of inferring conditions on attributes and parameters that involves the use of an invariant detector tool Daikon (Ernst et al. 2007). Since each constructed rule instance by tracing includes actual data values for all its node attributes, passed parameters and return. This gives us additional flexibility to collect data values for detecting invariants.

Under the first step (1), we describe the process of collecting, mapping and setting up data values from all rule instances that belong to a specific generalised (or maximal) rule. By considering such data, we can use them as *input* or training set to Daikon, see step (2). Then, in step (3), Daikon acts to discover

constraints to be fed back to the generalised rule. This allows to increase the accuracy of generalised rules by including precise conditions on attributes and parameters. We give more details about these steps in the next subsections.

5.2.2 Rules with Attribute and Parameter Constraints

In Figure 5.4, the contract for *registerClient(cityID: String, client: String)* describes the creation of a *Client* object linked to the *Branch* whose *city* matches the parameter $cityID$ (cf Section 3.2). This is expressed by the equality $branch.city = cityID$ in the *Branch* object.

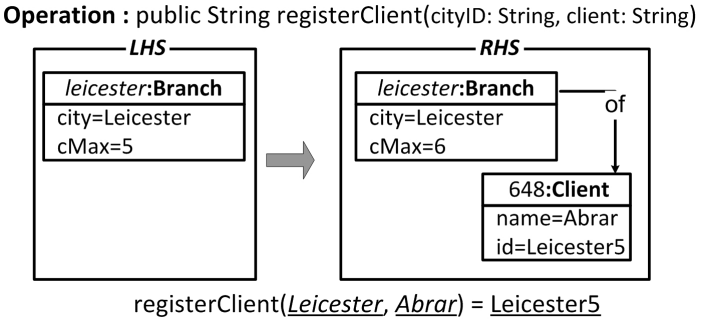


FIGURE 5.4: Contract instance extracted from *registerClient(..)*

Formally, $branch.city$ and $cityID$, as well as the right-hand side counterpart $branch.city'$ of $branch.city$, 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 $m : L \to G$, $branch.cityID$ is instantiated by the value of the *city* attribute of $m(branch)$, i.e., $m(branch.cityID) = m(branch).cityID$. 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 (Ehrig et al. 2006).

While access and changes to structure as well as data can be logged in the same process, the learning of logical data constraints requires a different approach

from the structural one described so far. 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 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. As described in Figure 5.3 (3), these constraints are fed back into the graphical part of the contract, where each becomes part of the pre- or postcondition depending on whether the variables used occur only in L or in L, R and the parameters. This approach allows the separation of structural and constraint learning.

5.2.3 Setting up Attributes and Parameters Values

Since the aim is to attach inferred invariants to generalised contracts, such as maximal rules, the actual values of rule attributes and parameters can be obtained from rule instances (cf. Chapter 3). This requires mapping node elements from maximal rules to all corresponding nodes in the relevant rule instances, and then fixing the order of attributes. Here, we use *node abstract id* for mapping node elements, which are mainly used for discovering intersections among rule instances, explained in Chapter 4. For fixing the order of the values we rely

on index arguments and node attribute names that are unique for each object (node).

In Table 5.1, we illustrate an example of the data values obtained from *register Client(..)*. The first two rows list variables specifying the order of nodes and also their attributes. In the other rows, we systematise all traced values from rule instances by using mapped *node abstract id, index arguments* and *attribute name*.

In Table 5.1, apart from the first two, each row represents values obtained from a single rule instance. For example, the data values shown in the third row are obtained from the rule instance described in Figure 5.4.

Input parameters		*Attributes in LHS*		*Attributes in RHS*				*Output*
City_ Par1	client Name_Par2	Branch L city	Branch L cMax	Branch R city	Branch R cMax	Client R cID	Client R cName	return
Leicester	Abrar	Leicester	5	Leicester	6	Leicester5	Abrar	Leicester5
London	Reiko	London	2	London	3	London2	Reiko	London2
London	Abdullah	London	3	London	4	London3	Abdullah	London3
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

TABLE 5.1: Mapping and ordering node elements and setting up their values

5.2.4 Learning Using Daikon

Daikon[2] is a machine learning tool, implemented for inferring program invariants from specific program point(s) (Ernst et al. 2007). Considering a Java method as a program point P, Daikon observes the values of variables in the scope of P, including (I/O) values, in order to detect all possible true properties or invariants. These values are usually obtained by tracing from several executions.

The inferred invariants can represent conditions for a single variable, e.g. for a column in Table 5.1 (second row), or the relationships among variables (i.e.

[2]The documentation of Daikon tool is available at `http://pag.csail.mit.edu/daikon/`

Invariants over	Detection
any variable	- constant value x = a - small value set x ∈ {a, b, c}
a single numeric variable	- range limits a ≤ x ≤ b - nonzero x ≠ 0 - modulus x ≡ a (mod b) - nonmodulus x ≢ a (mod b)
two numeric variables	- linear relationship $y = ax + b$ - ordering comparison $x < y, x \leq y$, $x > y, x \geq y$, $x = y, x \neq y$ - any invariant over $x + y$ - any invariant over $x - y$, e.g. $x - y > a$
three numeric variables	- linear relationship $z = ax + by + c$, $y = ax + bz + c$ or $x = ay + bz + c$
a single sequence variable	- range: *minimum* and *maximum* sequence values, ordered lexicographically - element ordering: *nondecreasing*, *nonincreasing* or *equal* - invariants over all sequence elements (*treated as a single large collection*)
two sequence variables	- linear relationship $y = ax + b$ elementwise - element comparison $x < y, x \leq y, x > y, x \geq y, x = y, x \neq y$ - subsequence relationship: x is a subsequence of y or vice versa - reversal: x is the reverse of y
a sequence (s) and a numeric variable (i)	- membership $i \in s$ e.g. element at the index $s[i]$ or $s[i-1]$

TABLE 5.2: Summary or the main cases of detecting invariants (Ernst et al. 2001)

among multiple columns). Table 5.2 summaries the main types of Daikon invariants by assuming (x, y and z) to be variables, while (a, b and c) are constant values (Ernst et al. 2001).

The program point P in our approach is a maximal rule and its variables are obtained from attributes and parameters of rule instances. As explained in the previous section, the values of these variables are extracted from rule instances during the tracing.

By giving Daikon inputs such as Table 5.1, the output will be similar to Listing 5.1. Here 14 invariant constraints have been inferred, some of which are describing the relation between attributes (e.g. $BranchLcMax - BranchRcMax +$

```
Daikon version 5.1.14, released December 22, 2014; http://plse.cs.washington.edu/daikon.
Reading declaration files .

(read 1 decls file)

Processing trace data; reading 1 dtrace file:

Finished reading RegisterClient.dtrace

==========================================================
aprogram.point:::POINT
child_Par1 == "objectRef"
CityPar1 one of { "Leicester", "London" }
CityPar1 != clientNamePar2
CityPar1 == BranchLcity
CityPar1 == BranchRcity
CityPar1 < ClientRcID
CityPar1 != ClientRcName
CityPar1 < return
clientNamePar2 != ClientRcID
clientNamePar2 >= ClientRcName
clientNamePar2 != return
BranchLcMax - BranchRcMax + 1 == 0
ClientRcID != ClientRcName
ClientRcID >= return
ClientRcName != return
Exiting Daikon.
```

Listing 5.1: An example of printing out Daikon constraints

$1 == 0$) and also between attributes and parameters (e.g. $CityPar1 == BranchLcity$). An example of conditions inferred for a single variable can be seen in $(CityPar1 - one - of - \{"Leicester", "London"\})$.

5.3 Summary

In this chapter, we introduced an inference technique to extend the learning of basic contracts, discussed in Chapter 4, by the derivation of general rules with multi objects and attribute constraints. This technique presents the third phase of our methodology as discussed in Section 1.3.3

The inference of multi objects enhances regular contracts as it adds more concise constraints to specify actions across sets of objects of different cardinalities. It helps in reducing the size of as well as the number of contracts by combing several generalised contracts (i.e., many maximal rules) whose effects only differ by the number of elements affected. We have discussed our proposed MO algorithm to discover sets of nodes in rules that have the same structure and behaviour, then represent them by a single multi-object node with proper cardinality.

In order to derive attribute constraints we used Daikon (Ernst et al. 2007) taking the actual data values of attributes and parameters as input and producing true invariant constraints. The actual data values are obtained from tracing during construction of rule instances. The invariants increase the accuracy of generalised contracts by including precise conditions on attributes and parameters.

The last technical contribution of this thesis has been given in this chapter, hence, proceed to evaluate and conclude the thesis in Part III.

Part III

Evaluation and Conclusion

Chapter 6

Evaluation

Our target, in this chapter, is to investigate empirically the efficiency and effectiveness of the technique introduced in Part II. We consider the following research questions:

- Can the correctness and completeness of extracted contracts be demonstrated or at least improved?
- Can the extracted contract instances be used for improving recall and accuracy of detecting faults in test reports? Do they help developers and for which kinds of faults they are most effective?
- How well does our approach scale to large rules and/or large number of rules? What is the cost of each individual step involved in the process, starting from tracing to the inference of contract features?

We firstly illustrate the implementation of the approach by a proof-of-concept tool in Section 6.1 and then discuss correctness and completeness of extracted contracts in Section 6.2. Section 6.3 reports on a user experiment, conducted to

assess the utility of test reports and localising faults using visual contracts. Scalability is evaluated based on three case studies in Section 6.4 and the summary of the chapter is given in Section 6.5.

6.1 Prototype Tool

The approach is implemented by a tool whose high-level architecture is shown in Figure 6.1. It consists of a *Tracer* observing the behaviour of selected classes using AspectJ and constructing contract instances (cf. Chapter 3), a *Generaliser* learning minimal, maximal (cf. Chapter 4), MO rules and attribute constraints (cf. Chapter 5) both supported by a database connection and a *Visualiser* for selective display and analysis of contracts. An export to the graph and model transformation tool Henshin (Arendt et al. 2010) is used to simulate contracts for validation.

The tool is implemented in Java[1] and relies on Daikon (Ernst et al. 2007) to learn data constraints (cf. Chapter 5). It uses a mySQL database as a backend to handle efficiently large (numbers of) contract. The use of such relational database provides the means to formulate complex operations on contract graphs as declarative queries.

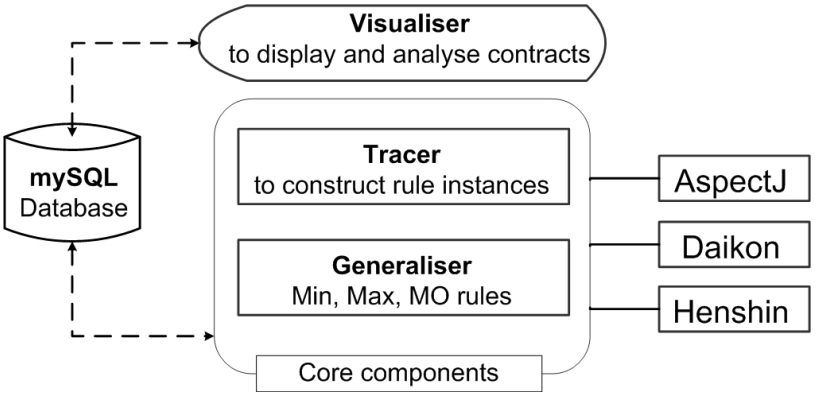


FIGURE 6.1: Architecture of the Tool

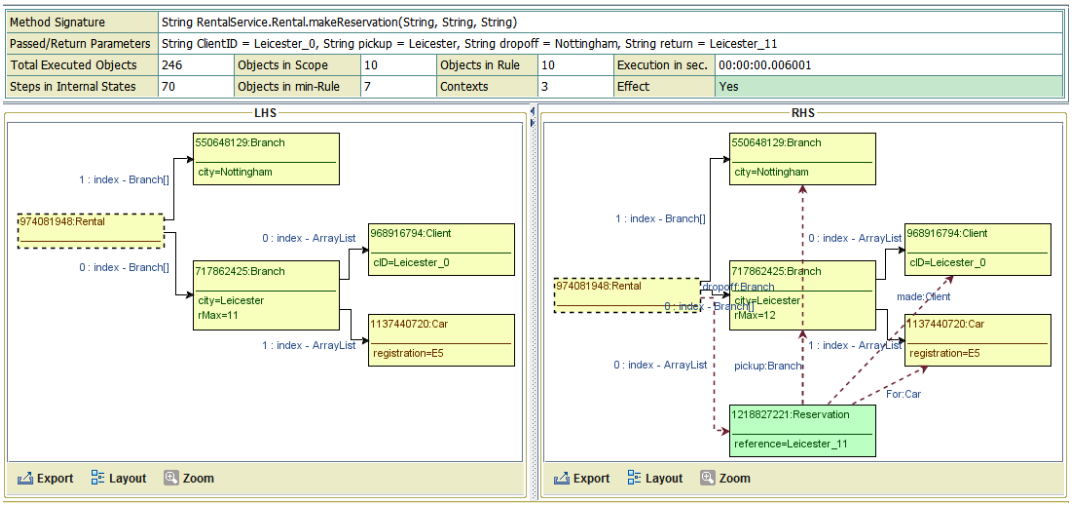
[1]The source code is available at `https://github.com/AMahfodh/IGTRRep`

In this section, we focus on the Visualiser to illustrate how results are presented and how they could be used to aid program understanding. The main task of the Visualiser (see Figure 6.2) is to organise, browse and display extracted contracts. To this end we support:

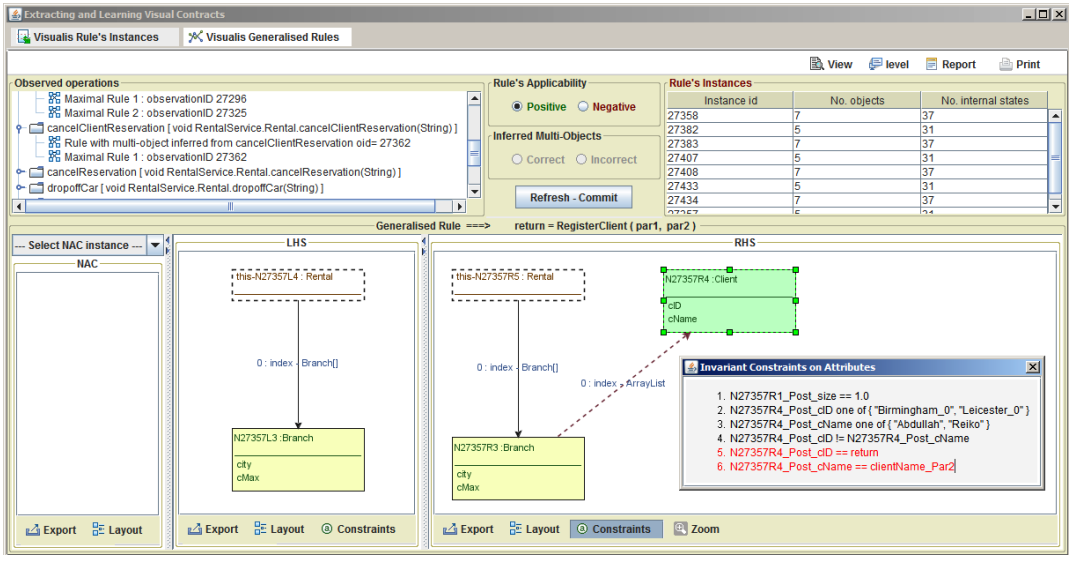
- the distinction in colour and style between elements of the minimal and larger maximal rule, e.g., dotted edges and nodes with coloured background (green for creation and light-golden for updated node attributes) represent elements of minimal rules, while nodes with white background and solid edges are context elements;
- the alternative display of collections as (to-many) associations or using explicit collection objects;
- the selective visualisation of rules, for example of the minimal rule or the precondition only, with the flexibility to change graph layouts;
- user interaction to confirm if inferred features are correct;
- export contracts to other formats, rather than Henshin (Arendt et al. 2010), e.g. as a pair of (GXL, JPG-images or DOT graphs).

6.1.1 Visualisation of Rule Instances

Figure 6.2 shows two screenshots of the main interface. In (a), we present an instance extracted from *makeReservation()* (cf. Section 3.2). The upper part of (a) gives information on the operation signature, actual parameters and the extraction process. Apart from the rule showing the precondition and effect at a high level, we provide information on the access to individual objects with the corresponding locations in the code. They are available through a pop-up window like the one in Figure 6.3 activated by clicking on the *:Reservation* node in the right-hand side of the contract.



(a) Rule instance



(b) Generalised rules interface

FIGURE 6.2: Visualiser interface

6.1.2 Visualisation of Advanced Rules

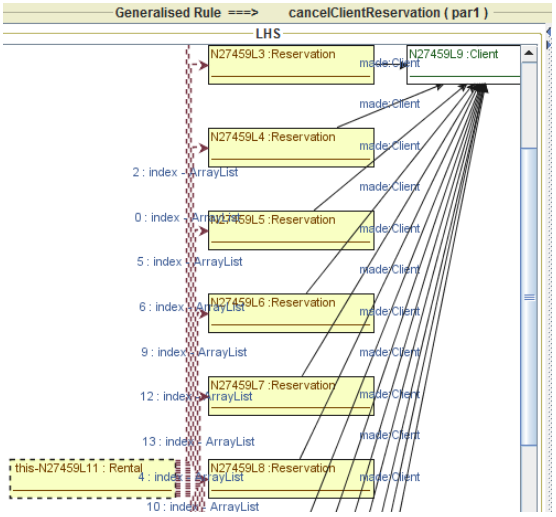
Figure 6.2 (b) shows how generalised (maximal and MO) rules are displayed. The top left shows a list of the rules organised by their operation signatures. When selecting, e.g., a maximal rule, all its rule instances will appear in the table, see the top right of (b). The lower part shows the maximal rule and

Node Details [954064616:Reservation]

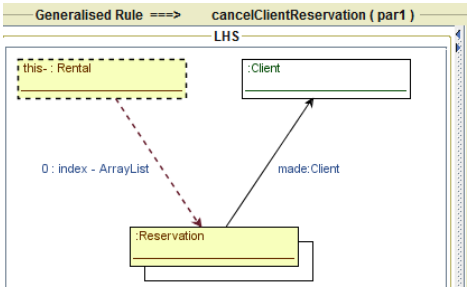
Access and code location details

Access Type	Internal State (step No)	Code Location (line No)
read	49	Rental.java - line 296
initialise	50	Reservation.java - line 21
write (made)	51	Reservation.java - line 13
write (pickup)	52	Reservation.java - line 14
write (dropoff)	53	Reservation.java - line 15
write (For)	54	Reservation.java - line 16

FIGURE 6.3: Object access and code locations



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



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

FIGURE 6.4: Extraction of rule with multi object

6 attribute constraints for *RegisterClient()* (cf. Section 3.2) that describe the relation between attribute values, input and return parameters. For example, the 5th constraint states that the value of the *cID* is returned while the 6th requires that the *cName* attribute of the new Client object has the same value as the 2nd parameter. An example of a rule with multi object is shown in Figure 6.4 (b) as extracted from the maximal rule in (a) for *cancelClientReservations()*.

6.2 Accuracy of Extracted Contracts

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 Chapter 3) 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. Generally, the models extracted will be neither correct nor complete.

6.2.1 Correctness

In software engineering, correctness means that the software's behaviours is consistent with the specification. This can be established by two different approaches: (1) formal verification and (2) software testing (Priestly 2005). In the context of formal verification, correctness is a boolean property, i.e., software is either correct or incorrect. However, correctness based on testing cannot only give a boolean result, but also level of expectation represented, e.g, by the number of tests passed. The use of correctness in this thesis follows the notion

of testing, and hence, improving correctness means to increase the number of invocations where the specification's prediction matches the implementation's behaviour.

Generally, proving correctness of our approach fails because the model is extracted for a certain part of the system only as identified by the implementation classes selected for tracing. Anything outside this scope of observation is not recorded and therefore not represented by the model. That means, if the implementation checks a condition on the state of an object outside scope, this check is not reflected in the precondition of the contract. If this check fails, a step in the model may not be reflected by a step in the implementation.

A weaker condition taking into account this limitation is that of *effect correctness*. It states that, if both preconditions are satisfied, the observable effect of the implementation-level step should match the effect of the model-level step. Here the comparison is moderated via the the mapping $G(\_)$ of implementation states to object graphs, which also takes account of the scope.

6.2.2 Completeness

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. Even if we assume that we have a complete statement and/or branch coverage of executing implementation, the extracted models can describe an over-approximation of the observed

behaviours, resulted by inferring additional features, that do not have a direct match to the implementation.

Despite the difficulty of generating a complete model, our technique allows to improve completeness by incrementally covering new behaviours when observing additional contracts. In Section 4.3, we explained how the existing (i.e., last inferred) maximal rules can be modified by covering new contract instances. In the remainder of this chapter, we illustrate an example of how this work, see Table 6.5.

6.2.3 Manual Inspection

We used manual inspection on the Car Rental Service case study (cf. Section 3.2) to validate if the models extracted by the tool satisfy the baseline/moderated notions of correctness and completeness. The limited amount of code and our familiarity with the application allowed us to perform a detailed review for every method in the interface, validating for all execution paths that there exists a rule in the corresponding contract capturing the path's combined precondition and effect, and vice versa for every rule that the behaviour described is fully implemented. This process was aided by the export of extracted contracts to the Henshin model transformation tool (Arendt et al. 2010), which provides a facility to simulate contracts based on their operational semantics as graph transformation rules.

A more automated solution is partially implemented, discussed in the future work section in Chapter 8, where Henshin was used to execute the model in parallel to the implementation, with the same invocations from the original set of tests being executed by both and the outputs compared for consistency. However, we could not fully prove the consistency of behaviours from both executions, as the current release (version 1.2.0) of Henshin does not support assigning rule attributes by variables.

Consider the source code fragment in Figure 6.5 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 6.5 shows the left-hand sides of the three rules extracted from *dropoffCar()*. For example, (a) reflects the behaviours of statements 1-5 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.

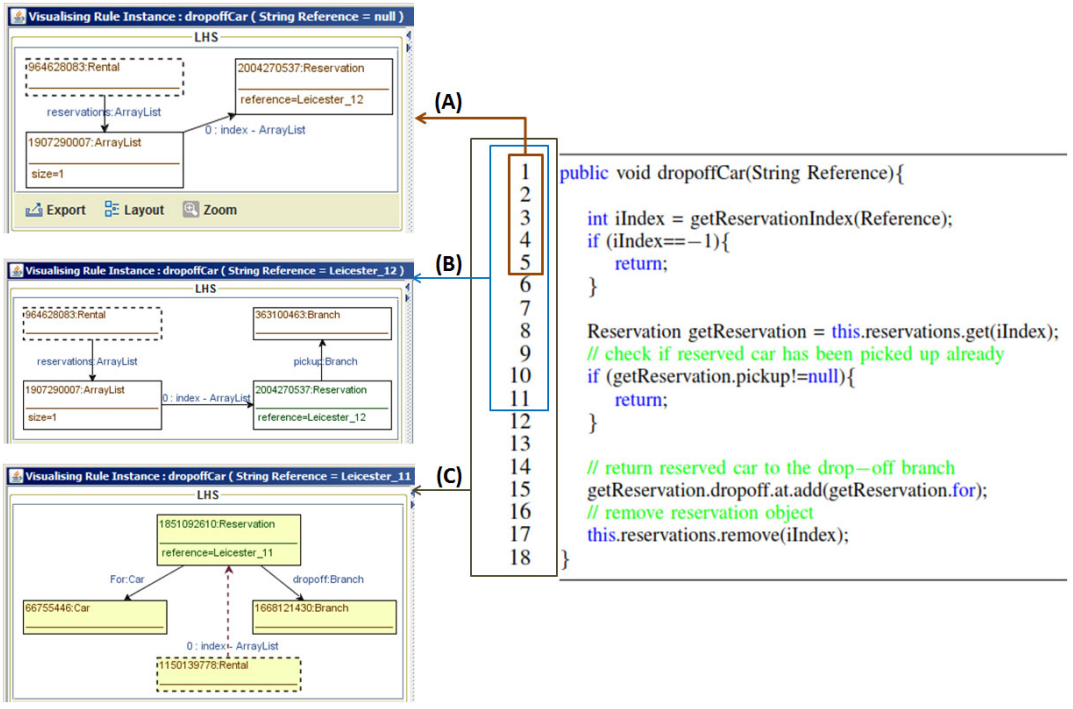


FIGURE 6.5: Implementation and rule instances for *dropOffCar()*

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

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

The fact that, in general, models are only representative of the behaviour they were extracted from is an obstacle to some applications, such as their use in verification, where automated extraction has to be followed by a manual review and completion of contracts.

In the next section, we demonstrate an application to program understanding in the context of testing and debugging that does not rely on completeness or correctness beyond the set of tests executed.

6.3 Utility in Assessing Test Reports and Localising Faults

Using the Car Rental Service case study (cf. Section 3.2), 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.

6.3.1 Experimental Setup

To conduct the experiment, an implementation of the Rental Car Service was documented in natural language (cf. Figure 3.3), seeded with 8 faults and provided with several short test cases able to detect them, see Figures 6.6, 6.7, 6.8, 6.9,6.10. The rest of details can be found in Appendix A.

Tests were executed and results recorded in two different formats:

(A) as sequences of invocations and returns of operations from the interface, with queries added to display details of the internal state after each step and

(B) as sequences of visual contracts extracted from the same invocations.

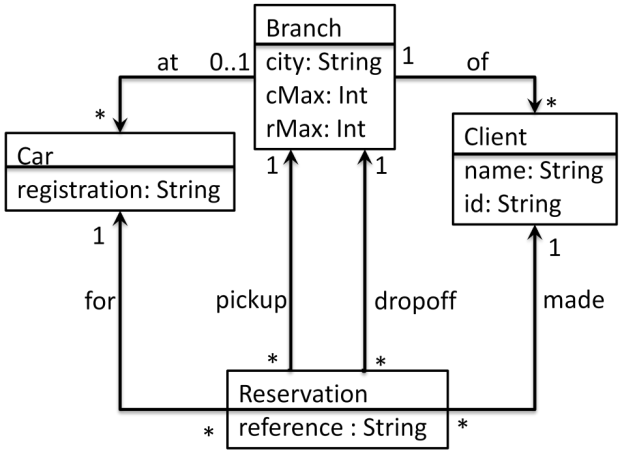
Students were asked to (1) identify invocations where the observed behaviour deviated from the expected based on the documentation and (2) locate the faults responsible in the code provided. Both groups received reports from 4

tests of 4-5 invocations each, containing a total of 20 failures to be traced down to the 8 seeded faults.

Service Specification

The specification below describing the interface of a car rental agency service consists of a class diagram modelling the available data, a list of operation signatures and a informal description of the preconditions and effects of those operations.

Data Model:



Operation Signatures:

- registerClient(city: String, client: String): String
- makeReservation(client: String, pickup: String, dropoff: String): String
- pickupCar(reference: String)
- dropoffCar(reference: String)
- cancelReservation(reference: String)

and for the queries:

- showClientReservations(client: String): Reservation[]
- showCars(reservation: String): Car[]
- showBranch(city: String): Branch
- showClients(city: String): Client[]

FIGURE 6.6: Service specification

```
1   public class Rental implements IRental{
2           private static final long serialVersionUID = 6324598725198583458L;
3           ...
4           public String registerClient(String city, String clientName){
5
6                   Branch cBranch = getBranch(city);
7                   if (cBranch !=null){
8
9                           Client newClient = new Client();
10                          newClient.name =clientName;
11                          newClient.id = cBranch.city + "_" +  (cBranch.of.size());
12
13                          cBranch.of.add(newClient);
14                          return newClient.id;
15                  }
16                  return null;
17          }
18
19          public String makeReservation(String ClientID, String pickup, String dropoff){
20
21                  Branch pickupBranch = getBranch(pickup);
22                  Branch dropOffBranch = getBranch(dropoff);
23
24                  Client clientMade = getClient(pickupBranch, ClientID);
25
26                  if (clientMade==null){
27                          clientMade = getClient(dropOffBranch, ClientID);
28                  }
29
30                  Car car = getCar(pickupBranch);
31
32                  if (pickupBranch==null
33                                  || dropOffBranch==null
34                                  || clientMade==null
35                                  || car==null){
36                          return null;
37                  }
38
39                  pickupBranch.rMax++;
40                  Reservation mReservation = new Reservation(
41                                  pickupBranch.city + "_" + pickupBranch.rMax,
42                                  clientMade,
43                                  pickupBranch,
44                                  dropOffBranch,
45                                  car);
46
47                  this.reservations.add(mReservation);
48                  return mReservation.reference;
49          }
50
51          public void cancelReservation(String Reference){
52
53                  for (int iIndex=this.reservations.size()-1; iIndex>=0; iIndex--){
54
55                          Reservation readRes= this.reservations.get(iIndex);
56                          if (!readRes.made.equalsIgnoreCase(Reference)){
57                                  continue;
58                          }
59                          else {
60                                  this.reservations.remove(iIndex);
61                          }
62                  }
63          }
64  ...
65
```

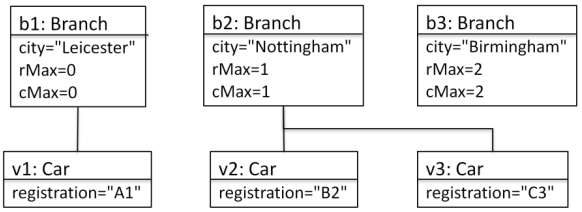
FIGURE 6.7: Implementation of the Rental Car Service

Class Test - Not Assessed

Full Name _____ CFS _____

You will be given 5 test reports (each consisting of a sequence of invocations) and the source code of their implementation. Your task is to find out which test reports show failures of the service to satisfy the specification and locate the faults responsible for the failures in the code.

At the start of each test case the database is initialised with 3 branches and 3 cars using the following data:



Question 1) Please use the following table to report the failed steps and explain briefly how the behaviour differs from the specification. The first row has been completed as an example of what is expected.

Failed step only		Brief Justification
Test Report ***No***	Step ***No***	
1	1	No change in attribute *Branch(Leicester).cMax* , which should be increased from 0 to 1.

FIGURE 6.8: Worksheet for explaining failed steps and localising faults

Test Report 1

Step	Operation invocation	* to show the state after Output
1	registerClient("Leicester", "Reiko")	"Leicester_0"
*	showClients("Leicester")	[0]={cName="Reiko", cID="Leicester_0"}
	showBranch("Leicester")	{city=" Leicester", rMax=0, cMax=0}
2	registerClient("Leicester", "Abdullah")	"Leicester_1"
*	showClients("Leicester")	[0]={cName="Reiko", cID="Leicester_0"} [1]={cName="Abdullah", cID="Leicester_1"}
	showBranch("Leicester")	{city=" Leicester", rMax=0, cMax=0}
3	makeReservation("Leicester_1", "Leicester", "Nottingham")	"Leicester_1"
*	showClientReservations("Leicester_1")	[0]={ reference="Leicester_1", made="Leicester_1", pickup="Leicester", dropoff="Nottingham", for="A1"}
	showBranch("Leicester")	{city=" Leicester", rMax=1, cMax=0}
4	makeReservation("Leicester_1", "Birmingham", "Leicester")	null
*	showClientReservations("Leicester_1")	[0]={ reference="Leicester_1", made="Leicester_1", pickup="Leicester", dropoff="Nottingham", for="A1"}
	showBranch("Leicester")	{city=" Leicester", rMax=1, cMax=0}
5	cancelReservation ("Leicester_1")	-
*	showClientReservations("Leicester_1")	null

Test Report 2

Step	Operation invocation	* to show the state after Output
1	registerClient("Nottingham", "Reiko")	"Nottingham_0"
*	showClients("Nottingham")	[0]={cName="Reiko", cID="Nottingham_0"}
	showBranch("Nottingham")	{city="Nottingham", rMax=1, cMax=1}
2	makeReservation("Nottingham_0", "Nottingham", "Nottingham")	"Nottingham_2"
*	showClientReservations("Nottingham_0")	[0]={reference="Nottingham_2", made="Nottingham_0", pickup=" Nottingham", dropoff="Nottingham", for="B2"}
	showBranch("Nottingham")	{city=" Nottingham", rMax=2, cMax=1}
3	makeReservation("Nottingham_0", "Leicester", "Nottingham")	"Leicester_1"
*	showClientReservations("Nottingham_0")	[0]={reference="Nottingham_2", made="Nottingham_0", pickup=" Nottingham", dropoff="Nottingham", for="B2"} [1]={reference="Leicester_1", made="Nottingham_0", pickup="Leicester", dropoff="Nottingham", for="A1"}
	showBranch("Nottingham")	{city="Nottingham", rMax=2, cMax=1}
	showBranch("Leicester")	{city="Leicester", rMax=1, cMax=0}
4	cancelReservation ("Nottingham_2")	-
*	showClientReservations("Nottingham_0")	null

FIGURE 6.9: Group A: Test reports used for detecting faults

Test Report 1

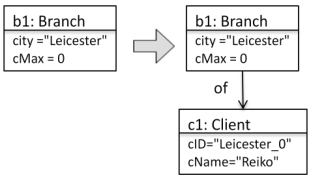
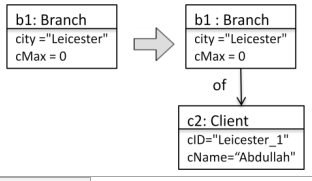
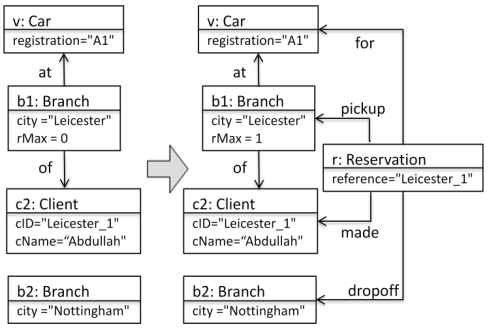
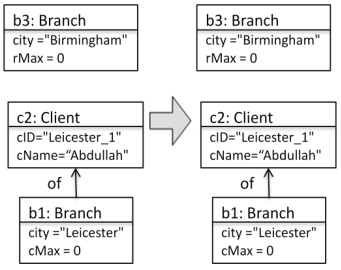
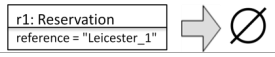
Step	Operation invocation	Extracted visual contracts	Access in the code *line number*
1	registerClient("Leicester", "Reiko")	Return : "Leicester_0" 	b1:Branch - 6, 11, 13 c1:Client - 9, 10, 11, 14
2	registerClient("Leicester", "Abdullah")	Return : "Leicester_1" 	b1:Branch - 6, 11, 13 c2:Client - 9, 10, 11, 14
3	makeReservation("Leicester_1", "Leicester", "Nottingham")	Return : "Leicester_1" 	b1:Branch - 21, 24, 39, 40 b2:Branch - 22, 40 c2:Client - 24 v:Car - 30, 40 r:Reservation - 40, 47, 48
4	makeReservation("Leicester_1", "Birmingham", "Leicester")	Return : null 	b1: Branch - 22, 27 b3:Branch - 21, 24, 30 c2:Client - 27
5	cancelReservation ("Leicester_1")	No return 	r1:Reservation - 55, 56

FIGURE 6.10: Group B: Test reports used for detecting faults,

6.3.2 Data Collection and Analysis

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. The ethical approval to conduct the experiment was given by the Ethics Board of the Department of Informatics before the experiment took place. We could use data from previously submitted coursework, one on modelling and one on implementation and testing, to check that the average level of qualification of participants in both groups was comparable. The groups A and B were selected randomly (handing out worksheets A and B alternatingly), resulting in 32 students in group A with an average coursework mark of 67.4% and 34 students in group B with an average coursework mark of 68.1%.

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

Group A achieved an avg. recall of 0.215 (identifying 1.7 out of the 8 faults) and an avg. precision of 0.232 (with 1.7 correct out of 7.4 responses). Group B had an avg. recall of 0.3 (correctly identifying 2.41 out of 8) and an avg. precision of 0.35 (with 2.41 correct out of 6.88 responses). This represents a factor of improvement *recall B / recall A* of $0.3/0.215 = 1.4$ and *precision B / precision A* of $0.35/0.232 = 1.5$.

In both cases, the t-test for independent two-sample experiments (for unequal

variances and population sizes) showed that the results are statistically significant with a probability (p-value) of 0.033 for recall and 0.013 for precision. The p-value was calculated using an online tool[2] for a degree of freedom of 64 (the sum of population sizes −2), a significance level of 0.05, and a one-tailed hypothesis (there is a reasonable expectation that group B would perform better than group A). That means, assuming the null hypothesis that "the different representations of test reports in both groups have no effect on the resulting scores" is true, there is a 0.033 resp. 0.013 probability of observing the same results due to random sampling error. The key figures are summarised in Table 6.1[3].

	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

TABLE 6.1: Statistical data for groups A and B

We investigated more closely which faults in which operations were detected more frequently by which group, see examples of visual representation of extracted contracts in Figure 6.11 or in Appendix A for complete version. 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.

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

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

The visual representation seems to be less effective for detecting faults in post-conditions than in preconditions. In fact, there are two examples of structural postcondition faults, see them in Figure 6.11, that were detected with higher frequency by group A than B, i.e.,

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

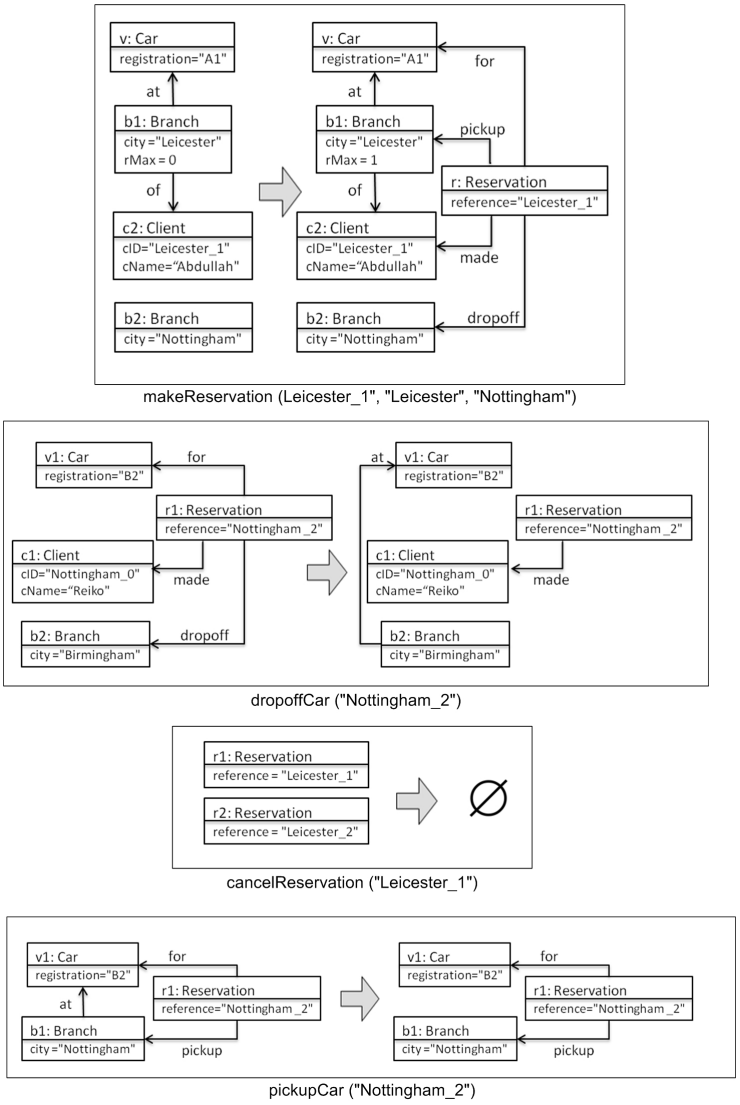


FIGURE 6.11: Examples of extracted contracts used for detecting faults.

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()* 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.

6.3.3 Discussion and Threats to Validity

While it is unlikely 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, discussed in the next section.

6.4 Performance and Scalability

Observing large applications can generate too many rules or rules with many objects. In this section, we assess how well our approach scales to large observations based on rule size and number of rules extracted. We explore the general impact on performance across a range of different rules, starting from extracting contract instances to inferring rule features. We conducted several

experiments, applied on three case studies. Furthermore, we will use one of these experiments to illustrate in more details the benefits and general validity of inferred rules.

6.4.1 Case Studies and Test Cases

Beside the Rental Car Service case study (cf. Section 3.2), we use (*NanoXML - a small non-validating XML parser for Java* 2016) and (*JHotDraw as Open-Source Project by Java* 2016) to evaluate scalability to large numbers of invocations as well as large object graphs. Both case studies are popular benchmarks for software testing and analysis, and representative of the kind of system our method would be appropriate for, i.e., with significant and dynamic object structures in their core model. In NanoXML this is the object representation of the XML tree, for JHotDraw that of graphics' objects.

NanoXML is a small non-validating XML parser for Java, which provides a light-weight and standard way to manipulate XML documents. We use version 2.2.1 which consists of three packages and 24 Java classes. We focus on two classes, *XMLElement* and *XMLAttribute*, which provide the functionalities to manipulate XML documents. We monitor 41 *XMLElement* methods, executing 5605 test cases in order to evaluate the handling of large numbers of invocations.

The original test cases were generated by CodePro[4], some of which are modified and completed manually to improve coverage. These tests cover 2099 out of 5836 instructions.

JHotDraw is a highly customisable Java GUI framework for technical and structured graphics editing, developed as an exercise in good software design

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

using patterns. At runtime, this GUI provides a set of features including toolbox menu to be invoked by the end-users, such as opening new window, inserting image, label or shapes etc. We used version 5.3 which comprises 2,080 methods implemented in 243 classes (34,710 instructions), focussing on the top level methods for the manipulation of graphs, such as **.addFigure(..), *.DeleteFigure(..), *.copyFigure(..), *.DecoratorFigure(..)* and all undoable actions in **.CommandMenu. actionPerformed(comExe)*. We use GUI testing using WindowTester[5] to generate test cases by recording user interactions. We executed 405 test cases that cover 9284 of 34710 instructions.

6.4.2 Extraction and Inference

In order to assess the scalability based on large inputs, we consider both *number of rules* and *rule size*. This includes time taken to (1) construct contract instances by tracing and (2) infer different rules with features from each instance.

For number of rules we focus on NanoXML and JHotDraw to evaluate rule size. Consider Figure 6.12 results from NanoXML, we plot the time taken to execute different batch sizes of tests, from 59 to 2183. Each test generates a single contract instance from which minimal and maximal rules, multi-objects and constraints are extracted. Tracing, contract instance construction and extraction of minimal rules are essentially linear, as is the derivation of constraints and multi objects. The construction of maximal rules requires to compare all rule instances with shared minimal rules, which is quadratic in the number of rule instances that share the same effect.

Based on the recorded test cases of JHotDraw, the total runtime of the extraction is about 3 hours 15 mins. Scalability is analogous to NanoXML, see

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

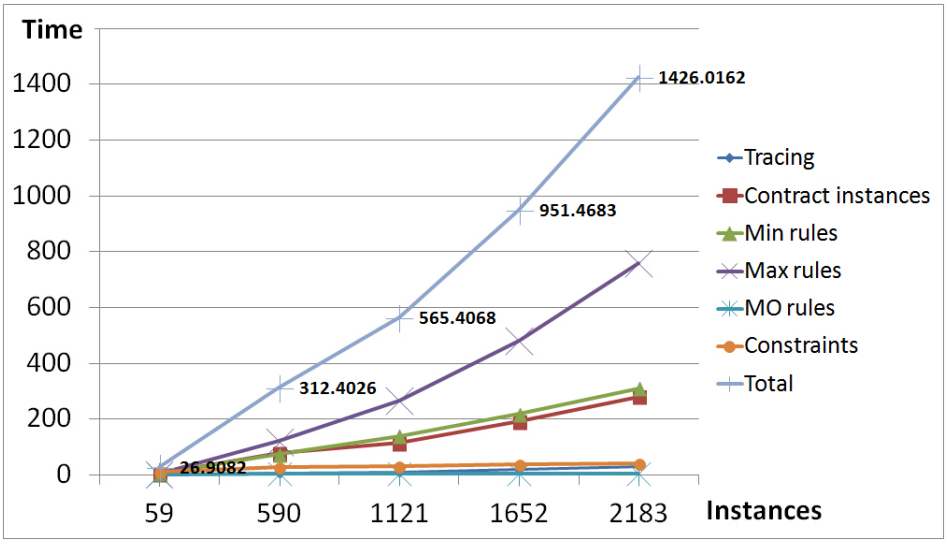


FIGURE 6.12: Scalability for extracting contracts from NanoXML

Figure 6.13, but the quadratic component of maximal rule extraction is less significant due to the smaller overall number of rule instances.

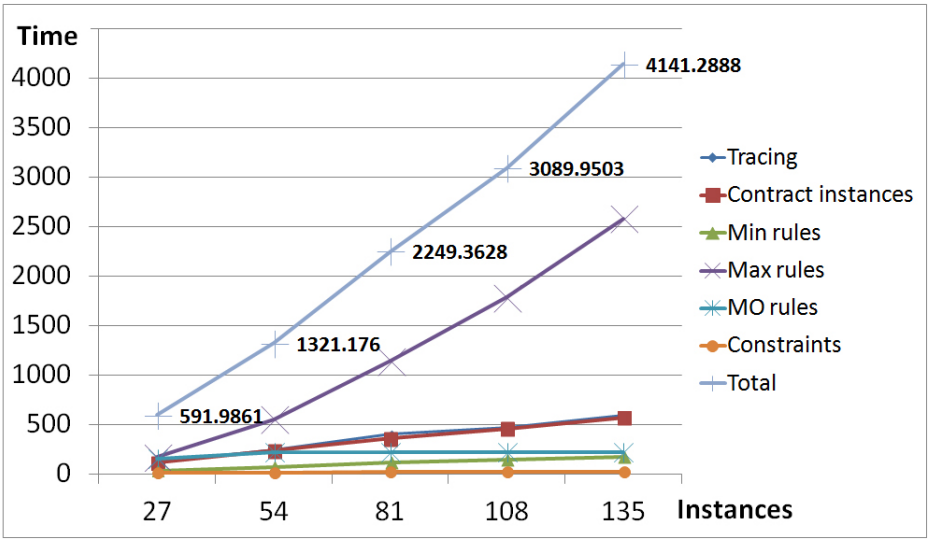


FIGURE 6.13: Scalability for extracting contracts from JHotDraw

Unlike NanoXML where the number of invocations / contract instances is large but the size of each contract instance small, JHotDraw produces contract instances up to several hundreds of objects, see Table 6.2. For more details,

Table 6.3 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), extracted from 135 instances, i.e., the last batch in Figure 6.13.

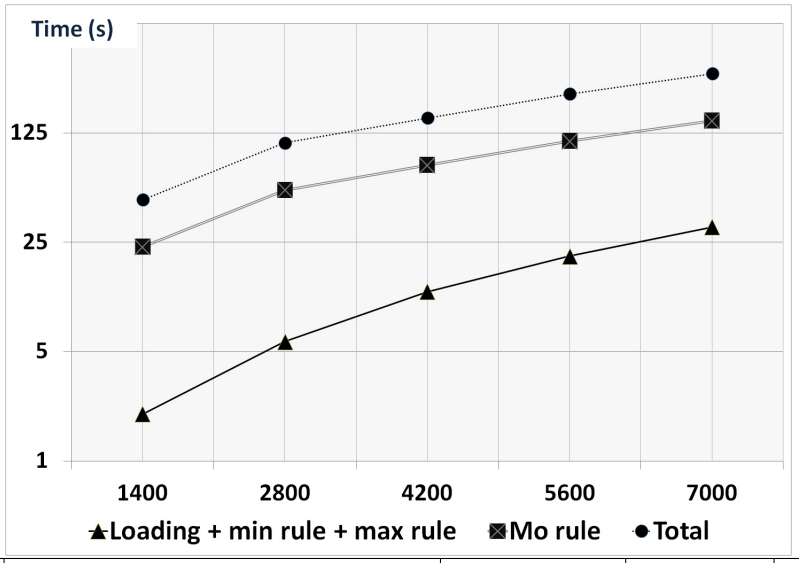
	Java methods	contract instances	max rules	MO rules
NanoXML	41	2183 (9081)	41 (125)	1 (38)
JHotDraw	5	135 (26141)	19 (3032)	2 (416)
		number (and size) of rules		

TABLE 6.2: Overall extractions vs. number of methods based on the last batch size (2183 and 135) of Figure 6.12 and Figure 6.13 respectively.

Executed method signature	accessed 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			

TABLE 6.3: JHotDraw objects accessed and processed for the construction of contracts from 135 instances, see Figure 6.13

Another conducted experiment to assess the scalability based on rule size, applied on Rental Car service. Here, we focus on contract instances extracted from operation *showClientReservations()*. These contracts have different minimal rules but produce isomorphic MOs rules. Figure 6.14 depicts the results by comparing different rules containing from 1400 to 7000 nodes. The results suggest that the relationship between contract size and time taken to load, construct min/max rules as well as to infer MOs is linear. The third column depicts the number of matching nodes used to produce a single MO node as well as the number of MO nodes itself. For example, in the first row, 2 MOs are extracted from 1393 objects.



rule size	loading+ min-rule+ max-rule	No-of *MOs*	MO rule	total
1400	1.993	1393(2)	21.39	23.38
2800	5.785	2793(2)	48.155	53.94
4200	12.072	4139(2)	65.855	77.93
5600	20.398	5593(2)	90.461	110.86
7000	31.339	6993(2)	118.299	149.64

All contracts extracted from *showClientReservations()*

Time - measured in seconds

FIGURE 6.14: Performance based on rule size

no	rule instance	No-of-instance	No-of *MOs*	loading+ min-rule+ max-rule	MO rule	total
1	registerClient	5	0	2.614	0.082	2.70
2	makeReservation	5	0	3.992	0.15	4.14
3	cancelReservation	5	0	2.191	0.031	2.22
4	cancelClientReservation	5	12(2)	3.373	1.042	4.42
5	pickupCar	5	0	4.764	0.046	4.81
6	dropoffCar	5	0	2.764	0.031	2.80
7	showClientReservations	5	12(2)	3.673	1.137	4.81
8	showClients	5	0	0.415	0.071	0.48
9	showCars	5	0	0.426	0.069	0.49

Time - measured in seconds **Mo** - Multi objects

TABLE 6.4: Performance of generalising 45 rule instances

6.4.3 Benefits and Validity of Generalisation

One of the experiment on the Rental Car Service (cf. Section 3.2) was aimed at illustrating benefits and general validity of inferred rules. For this aim, we have

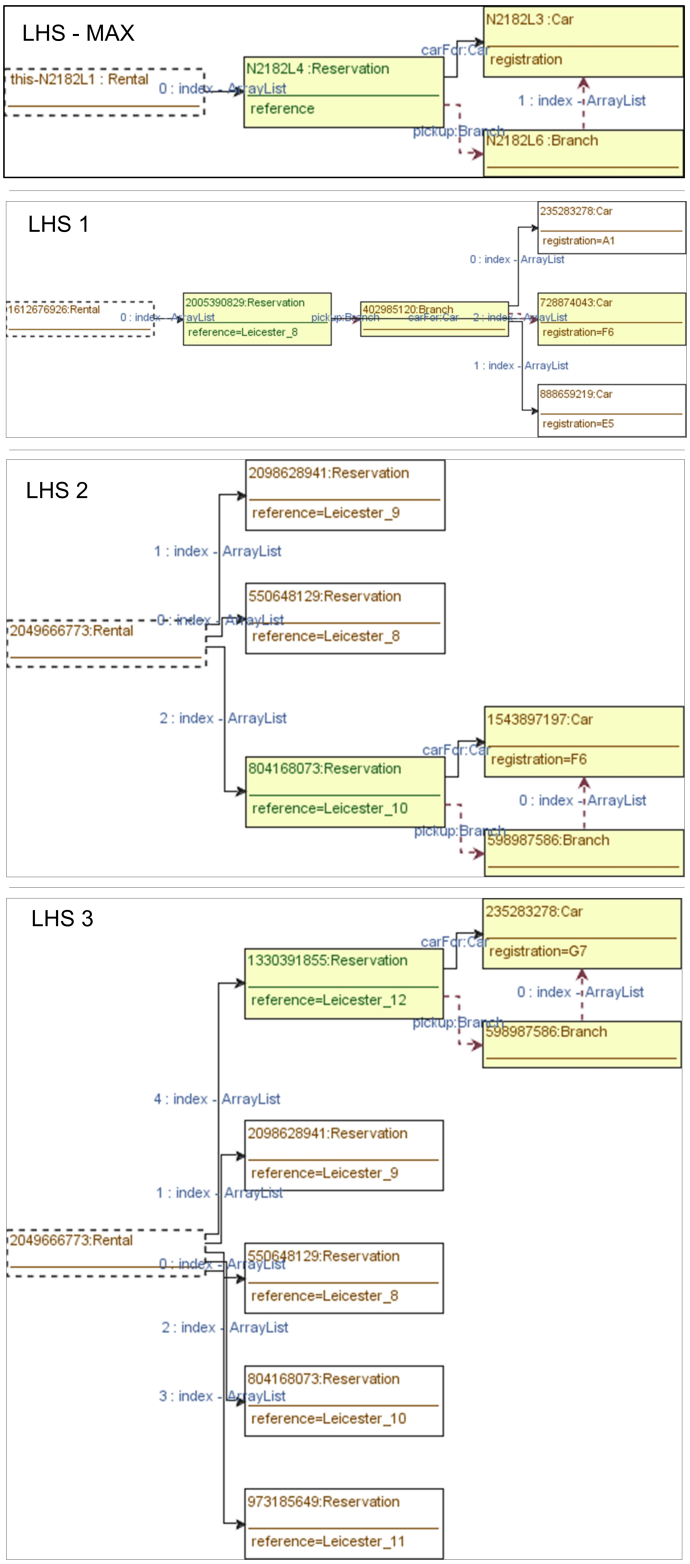


FIGURE 6.15: Generalised maximal rule for *pickupCar()*. *Best viewed at 350% zoom.*

generalised 45 rule instances into 9 sets of rules, classified by their operation signature. The result is reported in Table 6.4, which also shows the time taken to extract and infer rules with their features. The number of rule instances in each set is 5, as shown in the third column. Only two sets produce MOs: *cancelClientReservation()* and *showClientReservations()*.

The results also suggest that the time taken to run the MO algorithm on rules that do not contain any MO is less than for rules that containe MOs. This is thanks to the use of efficient database queries, which allow to establish quickly if two or more nodes share the same context. The effect is even more pronounced with larger rule sizes.

Benefit of constructing maximal rules: Rule *pickupCar()* in row 5 of Table 6.4 has been generalised from 5 rule instances. We present the left-hand side of only three of them as well as the generalised rule in Figure 6.15, see LHS1, LHS2, LHS3 and LHS MAX. This demonstrates the benefit of constructing maximal rules as unnecessary context is being ignored in (LHS MAX), such as the repeated *Car* objects in LHS1 or the *Reservation* object in LHS2 and LHS3. This simplifies the specification while making it more general. Note that the process of computing the LHS MAX is performed in an incremental mechanism, explained in Chapter 4, which make it easier and very efficient to include any new LHS instances not used before for inferring or updating the latest LHS MAX extracted. Table 6.5 illustrates the idea of this process, as we consider a pair of LHSs each time to either produce a new LHS MAX such as the first row or updating the previous LHS MAX extracted as in the other rows.

Iteration	first LHS	second LHS	extracted LHS-MAX
1	LHS 1	LHS 2	1^{st} LHS-MAX
2	1^{st} LHS-MAX	LHS 3	2^{nd} LHS-MAX
3	2^{nd} LHS-MAX	LHS 4	3^{th} LHS-MAX
4	3^{th} LHS-MAX	LHS 5	4^{th} LHS-MAX (latest)

Table 6.5: Incremental process of computing maximal rule

Benefit of inferring multi objects: Figure 6.4 provides a good example for raising the level of abstraction by inferring multi objects, resulting in a significant reduction in particular for operations performed in analogous form on a large set of objects.

Validity of Generalisation: according to the object types and the type of their relations, extracted from the class diagram in Figure 3.4, all generalised rules are valid. More significantly, all generalised rules always preserve the structure of minimal rules. For example, By comparing, manually, (LHS MAX) and (LHS 1) in Figure 6.15, the structure of minimal elements still have an exact match. For contexts elements, all generalised rules are also applicable in a sense, they can be applied in any instances from which they have been extracted.

6.4.4 Analysis of Results and Threats to Validity

One of the challenges we faced is how to execute systems to trace and extract visual contracts for evaluation purpose. Using normal execution or online monitoring, in the case of insufficient test cases available, might affect performance and lead to observe limited behaviours, resulting from constraints in settings or system configurations. Moreover, it is difficult to determine which scenario needs to execute in order to trigger the interesting part of the system (Cornelissen et al. 2009). The obvious solution is to generate test cases automatically but the challenges remain in *performance* and *completeness*. For example, the generated test cases can be very large and expensive to execute. In spite of this, they might not provide good coverage, which would affect the completeness of the extracted VCs.

Although the generation of test cases is outside the scope of this thesis, we have used different methods to execute systems, such as our manual implementation

of test cases and using test cases tools to automatically generate them from on-line executions and/or source code analysis.

Based on the results obtained 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 but they covered a large number of objects that may be hard to survey without this added level of abstraction.

Our evaluation is potentially biased by the selection of the case studies and the choice of operations and test cases within them. The NanoXML case study is one of a selection of benchmarks in the Software-artifact Infrastructure Repository[6] frequently used for evaluating automated testing and program analysis in Java. At 7646 LOC, 24 classes it is not a large API, but what we have seen from the evaluation suggests that it is the complexity of the individual operations, in terms of the number of objects involved, that determines the effort. This in turn depends on the number of classes we choose to observe.

NanoXML case study was selected because it offers a range of operations of significant structural complexity, creating objects and creating and destroying links. Moreover, we have developed/implemented the Car Rental Service case study to observe more complex structural preconditions. In that sense these two case studies are the kind of example for which an approach like ours may be useful. Analysing our own case study, the choice of NanoXML's operations to be traced and the test cases used for them are motivated by trying to explore all aspects of such complexity, and our test cases do indeed cover all the cases, as given in Table 3.1.

[6] `http://sir.unl.edu`

JHotDraw was selected to assess the scalability on a large API that has more than 18,000 non-documented LOC (Marin 2004). Furthermore, the focus was on the top level methods (e.g. **.CommandMenu.actionPerformed(comExe)*) that control GUI toolbar and manipulate drawing/editing all kind of shapes. This allows us to see how well our approach can handle the extraction of contracts and their required analyses from thousands of accessed objects at runtime. Rather than executing the system by normal test cases, recording user interactions to generate test cases also allows to estimate performance based on on line monitoring.

6.5 Summary

In this chapter, we have demonstrated the efficiency and effectiveness of extracting visual contracts from Java applications, proposed in Part II. We have presented our prototype tool to illustrate the extraction process as a proof-of-concept and explained that proving the completeness and/or correctness of extracted contracts dynamically is not always feasible. Despite this, our proposed technique allows to improve completeness by considering new behaviours each time after execution in an incremental mechanisms. We have used manual inspection to validate if the models extracted satisfy the baseline/moderated notions of correctness and completeness.

We have evaluated the usability of the approach based on 66 participating MSc students for analysing test reports and identifying faults. Here, we have shown 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. We also have evaluated scalability in experiments on three case studies, and shown that the scalability may be acceptable for batch processing moderately sized test suites, but not necessarily for interactive testing.

The overall evaluation provides some confidence in the validity of the proposed 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. Ultimately, through our discussions, 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. Such claims are beyond the scope of the thesis, but open new research directions to be explored in future.

Chapter 7

Comparison to the State of the Art

This research intends to extract dynamically visual contracts from object oriented systems (Java) in three steps: *model extraction*, *model generalisation* and *feature inference*. Similar steps have been considered in different research areas for which a range of related work exists. To organise them and discuss our investigation of approaches that are closely related to what we have achieved, we divide this chapter into three sections:

Model Extraction. This step involves observing actual behaviour by tracing, to be represented by instantiated versions of visual contracts. Here we discuss related approaches with respect to dynamic analysis techniques. The state of the art of reverse engineering more generally has already been discussed in Chapter 2.

Model Generalisation. We compare the relevant approaches with our learning mechanism based on graph transformation and graph pattern matching for rule generalisation. We explore recent work in two different aspects: applications and algorithmic solutions.

Feature Inferences. We investigate shared ideas for inferring multi-objects and rule attribute/parameter constraints that exists in the state of the art.

7.1 Model Extraction

Automated reverse engineering is based on static or dynamic analysis. The static approach, exemplified by (Rountev, Volgin, and Reddoch 2005; Sarkar, Chatterjee, and Mukherjee 2013; Tonella and Potrich 2003), 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 (Rountev, Volgin, and Reddoch 2005), but limited in its ability to detect dynamic object-oriented behaviours such as dynamic binding. For example, (Tonella and Potrich 2003) propose a static approach for generating sequence and collaboration diagrams from C++ code, thereby potentially over-approximating the actual behaviour. The drawback of a dynamic approach, such as our but also (Brito et al. 2012; Zhao, Kong, and Zhang 2010; Ziadi et al. 2011), is that the extracted model represents only those behaviours that are actually executed. Next, we discus so-called Architecture Driven Modernization (ADM), then we narrow down to the relevant tracing techniques and types of extracted models, including contracts.

7.1.1 Architecture-Driven Modernization

As mentioned in Chapter 2, MDA aims to separate implementation and design from architecture and focuses on forward engineering, i.e., transferring models from higher levels to lower and more concrete levels (Chen et al. 2006; Siegel 2014). The reverse method so-called Architecture Driven Modernization (ADM)

is proposed by OMG's ADMTF[1] (*The Architecture-Driven Modernization Task Force (ADMTF)* 2016). They define ADM as "the process of understanding and evolving existing software assets for the purpose of software improvement, modifications, interoperability, refactoring, restructuring, reuse, porting, migration, language translation, enterprise application integration, SOA, MDA migration and making existing applications more agile".

Following a similar objective to ours, ADM is necessary to evolve existing software in the case of out-of-date PIM and/or PSM's documentation, unmodelled software or modelled software that did not follow MDA's standards (Chen et al. 2006). In Figure 2.3, we explained the idea of ADM within the MDA environment, which is very analogue to our approach in discovering the PSM model (visual contract) from the implementation.

7.1.2 Tracing Approaches

Many dynamic reverse engineering approaches take advantage of a customised debugger, source transformation (e.g. using the TXL language[2]) or aspect oriented programming for software analysis and instrumentation at different levels. We use AspectJ to instrument low-level Java bytecode and generate visual contract instances at runtime. A similar strategy lies behind (Brito et al. 2012) generating hierarchical object graphs or (Zhao, Kong, and Zhang 2010) extracting graph grammar. At the level of models (Amar et al. 2010) discovers relationships to support model traceability (Galvao and Goknil 2007), e.g., the relation between elements of source and target models in a model transformation.

[1] ADM is the backward of MDA, so ADMTF stands for Architecture-Driven Modernization Task Force

[2] The TXL Programming Language (Source Transformation)- `http://www.txl.ca/`

The requirement to trace systems dynamically normally considers (1) an analysis strategy, by e.g., instrumentation, (2) points of interest to observe program elements at runtime such as point-cut in AOP or breakpoints in debuggers, and (3) the representation of generated logs to be used for constructing target models. In Table 7.1, we particularly describe these requirements to differentiate the main similarities and differences between existing tracing techniques with ours (presented in Chapter 3).

	Analysis strategy	Program elements	Logs representation	Extracted model
ours	AOP-AspectJ to instrument *byte/source code*	generic -initialization, -set/get field access, -and method calls *with parameters*	recorded sequences of accessed objects	instances of visual contracts
[A]	* AOP-AspectJ	* generic -initialization, -set/get field access, -and method calls *with parameters*	no logs recorded, but *online extraction and visualization*	hierarchical object graphs
[B]	customized debugger	* method calls but *without parameters*	Labeled Transition Systems (LTS)	sequence diagrams
[C]	TXL source transformation	* method calls	Labeled Transition Systems (LTS)	context-based behaviour models
[D]	* AOP-AspectJ	* method calls but *without parameters*	GraphML representing *call graph*	graph grammar *context-free*
[E]	* AOP-AspectJ integrated with model transformation tool	* specific -initialization, -set/get field access, -and method calls *with parameters*	model based on Java/EMF	model traceability *links model from EMF transformations*
[F]	* AOP-AspectJ	* method calls	recorded sequences of accessed objects	model close to: - Control Flow Graph - Finite State Machine
* indicates similarities vertically in each column				

References

A Brito et al. 2012
B Ziadi et al. 2011
C Duarte, Kramer, and Uchitel 2006
D Zhao, Kong, and Zhang 2010
E Amar et al. 2010
F Wasylkowski, Zeller, and Lindig 2007

TABLE 7.1: Similarities with tracing approaches based on aspect oriented

A closely related approach, also based on Java bytecode instrumentation, is (Zhao, Kong, and Zhang 2010) aiming at 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. Not surprisingly, their tracing strategy focusses on method calls only.

They use a mining approach to infer the graph grammar by processing the set of extracted call graphs.

(Wasylkowski, Zeller, and Lindig 2007) propose a code smells detection approach that takes Java code examples to infer legal sequences of method calls. They use mining and pattern detection techniques to find locations of abnormal object usage (defect candidates) in programs in two steps. They first analyse statically code to create a 'method model' closely related to the control flow graph representation. Then, they create an 'object usage model' by data-flow analysis to check all places in the code where the object could be a target or a parameter of a method call. Their object model represents finite state machine with unknown states and transitions but labeled with method calls.

The approach to model traceability in (Amar et al. 2010) infers relations based on aspect-oriented programming. They trace access to objects by creation, update and deletion actions based on transformation events and aggregate them into a relation model. Despite a similar family of observed actions, our approach in this stage works at the level of Java programmes, not models.

7.1.3 Type of Extracted Models

The current literature reports a significant number of reverse engineering tools and approaches for extracting different types of models, particularly UML models. For instance, static, dynamic and hybrid analysis have been applied, within the MDA Framework, to build structural models, such as class/state diagrams (Favre, Martinez, and Pereira 2009) and use case diagrams (Claudia, Liliana, and Liliana 2011; Muhairat and Al-Qutaish 2009), and also behavioural models including sequence diagrams (Ziadi et al. 2011) and activity diagrams (Martinez, Pereira, and Favre 2011). Other models (which do not follow MDA concepts) have also been considered, such as hierarchical object graphs (Brito et

al. 2012), graphical user interfaces (Kull 2012), ontology models (Alnusair and Zhao 2009), finite-state models (Corbett et al. 2000; Pradel and Gross 2009), entity data models from databases (Malpani et al. 2010) and graph grammars based on call graphs (Zhao, Kong, and Zhang 2010).

Our extracted models describe visually the high-level behaviour of software operations (Meyer 1992) in the form of a pair of UML object graphs. They are used to specify the changes of system's states before (pre-condition) and after (post-condition) the execution of an operation. The only similar reverse engineering approach we are aware of has been proposed in (Porres and Rauf 2010), discussed in the next section.

7.1.4 Extraction of Contract Models

The approach proposed by (Porres and Rauf 2010) aims at generating *class contracts* from protocol state machines that specify a pre- and post-condition for each operation, required to trigger a transition in the state machine. These class contracts are asserted by JML (Leavens, Baker, and Ruby 2006) and can be used to generate test oracles (i.e. run-time assertions) or to generate test cases.

The shared idea with ours is extracting actual behaviour of contract instances. Apart from this, our approach is completely different. Fundamentally, our contracts are visual, extracted from real execution and rely on object diagrams not state machines. We also do not consider contracts at code level as they propose by JML. Additionally, we get ahead by supporting the inference of attribute/parameter conditions and multi-objects.

7.2 Model Generalisation

Reverse engineering visual contracts is a process of learning rules from transformations. This has been suggested in a number of areas, including the modelling of real-word business processes (Bruggink 2014), biochemical reactions (You, Holder, and Cook 2009) and model transformations (Dolques et al. 2011). 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 this section into: application domains and published graph pattern mining solutions for learning rules.

7.2.1 Application Domain

There are three application domains we have come across so far, discussed in the following sections:

7.2.1.1 Business Processes

(Bruggink 2014) discusses five mining algorithms, namely, *maximal, minimal, context, genetic and greedy mutation*, for learning graph transformation models from transition systems. These models can be used to understand behaviour of business processes and/or to model behaviour of human experts. Given an observed trace, some transition sequences, to be an input for their learning algorithms, they first encode manually transitions by identifying isomorphic graphs for each sequential states. This means each transition is limited to describe creation and/or deletion of edge elements only. In general, this initial step may affect usability, and obviously requires knowledge about the system under analysis.

The first algorithm maximal rule aims to produce a (large) rule for each transition, assuming the whole states before and after to be the LHS and RHS of the generated rule, respectively. The applicability as well as understandability of such generated rules are problematic, as too much context is present. In their minimal rule algorithm, the produced rules are similar to ours. They describe the deletion and/or creation of elements without including contexts elements. As all the transitions are already identified by isomorphic graphs in their preprocessing step, computing graph differences (i.e., specifying deletion/creation of edges only, while we also consider nodes) is straightforward.

Despite this, the quality of generated minimal rules are low as stated. They measure this by comparing transition system generated by the inferred models to the input transition system that were used as input to the algorithms. Since there is no context in the minimal rules, the model allows arbitrary the movement of some transitions.

The 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 context from a chosen contract instance, which makes it easier to maintain the graph structure as valid against the type-graph. In addition, we are dealing with more complex (UML object) graph structures, supporting the inference of advanced rule features, such as multi-objects and attribute/parameter conditions.

The fourth genetic algorithm has an exponential time complexity. The author has studied the possibility of combining genetic and greedy mutation algorithms to produce better results, these algorithms suffer from recognising disjunctive application conditions, resulting in the inability to generate meaningful common context in the rules. The correctness of produced rules is not evaluated.

The suggestion of how one can measure quality by means of *conformance* and *simplicity* is discussed, but not released.

7.2.1.2 Biological Systems

In (You, Holder, and Cook 2009)'s work, 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.

7.2.1.3 Model Transformation

As modelling a software system often requires many different steps, with more than one model considered at each step, MDA introduces an architecture that categorise the various kinds of models used with respect to their level of abstraction. This architecture includes three main layers: Computation Independent Model (CIM)[3], Platform Independent Models (PIM) and Platform Specific Models (PSM) (Siegel 2014).

Broadly speaking, PIM focuses on business concepts and solving domain model problems without specifying any technological platform. PSM is tied to a specific technology, i.e., focuses on modelling how a system can be implemented by a given technology or platform (by e.g., a particular programming language or operating system) (Siegel 2014).

One of the key ideas in MDA, regarding its architecture, is to support model transformation that aims to transform a given model into a target model (Mens and Van Gorp 2006). In general, MDA transformation specifications "provide

[3]CIM - models of the actual people, places, things, and laws of a domain. The instances of these models are *real things*, not representations of those things in an information system.(Siegel 2014)

the mechanisms to transform between representations and levels of abstraction or architectural layers"(Siegel 2014). However, MDA puts emphasis on forward engineering (Chen et al. 2006), i.e., transferring models from higher-level PIM (e.g., formal specifications) to lower-level PSMs (e.g. UML models) or from PSM to code level, see the left arrow in Figure 2.3. In (Berrisford 2004)'s work, the article describes superficially the four possible transformations (CIM↔PIM) and (PIM↔PSM) in more details.

In order to permit model transformation, several languages have been defined with various purposes and features. A general survey of some of these approaches are discussed in (Czarnecki and Helsen 2006), such as Henshin (Arendt et al. 2010), VIATRA[4], AndroMDA[5] and ATLAS[6] etc. The latter is widely used in MDA (Allilaire and Idrissi 2004), which is based on OMG standard QVT [7].

When considering approaches to learning model transformations, surveyed in (Kappel et al. 2012) and, more recently, in (Baki and Sahraoui 2016), we have to distinguish two types of transformations (Mens and Van Gorp 2006), i.e., *in-place or endogenous transformations* (Langer, Wimmer, and Kappel 2010) where source and target have the same metamodel, as in animating object diagrams, and *out-place or exogenous transformations* (Avazpour, Grundy, and Grunske 2015; Baki and Sahraoui 2016; Faunes, Sahraoui, and Boukadoum 2013; Kühne et al. 2016), where the metamodels are different, such as when transforming UML class diagrams to relation schemata. However, in-place can implement out-place by creating a joint metamodel. In addition, (Kappel et al. 2012) classifies the proposed approaches into model transformation by-example (MTBE) and model transformation by demonstration (MTBD).

[4]VIATRA: An Event-driven and Reactive Model Transformation Platform, Eclipse Modelling Project - `http://www.eclipse.org/viatra/`

[5]AndroMDA is an open source MDA framework- `http://www.andromda.org/`

[6]ATLAS is a model transformation language and toolkit, Eclipse Modelling Project - `https://eclipse.org/atl/`

[7](QVT) Query/View/Transformation - `http://www.omg.org/spec/QVT/`

The underlying principle of MTBE, initially introduced in (Varró 2006), is to use interrelated source and target models as examples. These pairs of example models are defined manually by a domain expert. The idea of MTBD (Langer, Wimmer, and Kappel 2010) is to observe transformations by stepwise recording of editing actions such as adding, updating or removing model elements. To derive rules that describe transformation patterns applicable to other models, this approach relies on templates created by demonstration by the user. Concerning the post-2012 approaches to learning model transformation specifications from examples, (Agirre, Sagardui, and Etxeberria 2014; Baki and Sahraoui 2016; Faunes, Sahraoui, and Boukadoum 2013; Kühne et al. 2016) are based on the principle of MTBE, while (Avazpour, Grundy, and Grunske 2015; Langer, Wimmer, and Kappel 2010) fall into the latter category of MTBD. Our approach in the tradition of MTBE targeting in-place transformation.

However, referring to the general MTBE process sketched in (Kappel et al. 2012), the most distinguishing aspect of our approach is to support features such as the inference of multi-objects and conditions on parameters and data attributes. To the best of our knowledge, none of these features have been previously addressed by any learning approach. This inference is based on specific graph transformation theory, such as minimal rule extraction and rule matching. Existing MTBE approaches sharing with our approach the idea of using larger sets of examples as input for learning are rather driven by different heuristics, e.g. based on genetic programming (Baki et al. 2014; Faunes, Sahraoui, and Boukadoum 2013), association-based mining (Dolques et al. 2010), inductive logic programming (Balogh and Varró 2009), and other heuristic algorithms (Dolques et al. 2011) or machine learning.

Table 7.2 summaries model transformation approaches for learning rules. For out-place transformation, (Dolques et al. 2011) represent input and output models as meta-model instances supporting concepts such as attributes, inheritance,

aggregation, etc. As their transformations mode are out-place, their input-output pairs would not typically represent the result of applying a single rule, but potentially a process consisting of several steps. (Balogh and Varró 2009; Faunes, Sahraoui, and Boukadoum 2012; Faunes, Sahraoui, and Boukadoum 2013; Varró 2006) also propose the learning of out-place transformation rules, while our approach is of the in-place variety.

The approach proposed by (Langer, Wimmer, and Kappel 2010) address the learning of in-place model transformations. This approach is interactive, requiring user involvement to confirm the rules proposed by the algorithm on the basis of observed transformations. Our approach does not have 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.

The approach presented in (Avazpour, Grundy, and Grunske 2015) focuses on usability by combining both MTBE and MTBD in out-place mode. Their aim is to provide domain experts with features to visualise source and target examples, recording actions and define correspondence elements to generate complex model transformation. Our approach is more flexible in that we only rely on dynamic tracing to define examples in in-place mode applications.

7.2.2 Graph Pattern Mining

An algorithmic problem closely related to the extraction of rules from example transformations is graph pattern discovery. Current approaches can be classified into statistical and node signature-based approaches:

Approach	Learning inputs	learning technique	Learning outputs
Our approach (*in-place*)	- Pair of UML object graphs constructed by tracing that represent operation behaviour	- GT theory - Our algorithm based on *pattern graph matching*	- Minimal and maximal rules, - Rule with multi-objects
[A] *in-place*	- Metamodel-independent - An empty context-free rule	- User involvement to manually configure annotated templates *in 4 iterations* - State-based comparison *to identify the new /deleted elements*	- Complex rules that can dependent
[B,C,D] *out-place*	- *Source / target* metamodels, - Class diagram to rel. schema example	Genetic programming algorithm (*heuristic search*)	- Many-to-many transformation rules - Control (rule order) [D] - Negative conditions [D]
[E] *out-place*	- *Source / target* metamodels, - Class diagram to rel. schema example	Anchor-prompt matching	Relational model with source and target links to support MTBE
[F,G] *out-place*	- *Source / target* metamodels, - Mapping metamodels and models - Class diagram to rel. schema example	Inductive logic programming	- Sets of rules for generating: *(a) target nodes* and *(b) target edges*, - Negative conditions
[H] *out-place*	- *Source / target* metamodels, - Mapping metamodels, and pair of models with traces	- Multi-phase learning - Genetic programming algorithm (*heuristic search*)	Sets of rules, describing transformation programs

References

A Langer, Wimmer, and Kappel 2010
B Faunes, Sahraoui, and Boukadoum 2012
C Faunes, Sahraoui, and Boukadoum 2013
D Baki et al. 2014
E Dolques et al. 2011
F Balogh and Varró 2009
G Varró 2006
H Baki and Sahraoui 2016

TABLE 7.2: Comparison with learning rules from model transformation

7.2.2.1 Statistical Approaches

Finding graph patterns by statistical means is popular in machine learning (Qiu et al. 2010b). They can produce a large variance in results, depending on the frequency of a pattern. For instance, an object that is not accessed, but always present in the context, would be considered an important element of the rule. (Qiu et al. 2010b) apply decision tree learning, starting to discover matches from predefined anchor points in a hierarchical search pattern, resulting in exponential effort.

7.2.2.2 Node Signature-Based Approaches

The use of node signatures, as in our approach, can reduce this effort, but the problem remains NP-complete. (Conte et al. 2004; Dahm et al. 2015) discuss research in exact and best graph pattern matching. A crucial point in graph or sub-graph matching is how to make nodes distinguishable when they are candidates for possible matches. For example in (Jouili, Mili, and Tabbone 2009), a node signature for attributed graphs is based on node/edge types and node attribute(s). We use a node signature-based approach with added structural information and metadata, extended from subgraph to subrule matching, taking into account shared minimal rules and parameters.

7.3 Feature Inferences

To the best of our knowledge, no work has been done on inferring multi objects for visual contracts. However, similar approaches for inferring invariant conditions have be considered in component-based verification (Mariani 2004), and software testing, such as addressing the test oracles problem (Barr et al. 2015) or generating logical test inputs (Artzi et al. 2006). We share with them the technique of using program invariant detection by Daikon (Ernst et al. 2007).

(Mariani 2004) focuses on verification of component-based systems by monitoring I/O and interactions and using Daikon to detect invariants. Their inferred invariants are not intended to address testing problems or to derive test suites, but used for behavioural verification. For example, they are used to verify the behaviour of similar components used in other systems, by invariants that define relations between system requests and component results, and also between observed interactions.

(Barr et al. 2015) discusses research on deriving test oracles from development artefacts, particularly from system execution trace. Their main idea is to infer invariants that capture program behaviours by observing sets of inputs/outputs at runtime, e.g. from passed test cases. These invariants can be then used to validate program correctness after any modifications, e.g., when applying regression testing or when checking new test inputs.

(Artzi et al. 2006) proposes an automated solution to generate logical test inputs in two steps: inferring a model (by tracing) that represents a call sequence graph, and then using this model along with random test generation and Daikon to generate test inputs. We share with them the initial idea of using trace information that includes method calls, parameters/return values and values of all accessed object attributes for analysis and inferences. In our case, the inference output represents invariant constraints on rule attributes/parameters and their possible matches (with primitive and string types). In contrast, (Artzi et al. 2006) uses their inference to support test input generation for creating valid and logical inputs

7.4 Summary

In this chapter, we have compared our approach for extracting and inferring visual contracts with the current state of the art in different research fields, including dynamic analysis techniques to reverse engineering models, model transformations and graph pattern mining. A brief summary of the relevant research are given below.

Dynamic Tracing. To reverse engineering models from existing systems at run-time, we use AOP (*AspectJ* 2016) and propose an original strategy to construct contract instances Chapter 3. We have given some detailed comparison

with dynamic tracing approaches that share with us the same underlying concept of extracting models. However, constructing visual contract (instance level) from code has not been investigated in the literature.

Generalisation of Rule Instances. The underlying idea of our learning is quite similar to some MTBE approaches for learning rules from model transformations. We have pointed to approaches such as learning rules form behaviours of business process as well as biological network systems. The novelty in our approach lies in the extraction of maximal rule, discussed for the first time in (Alshanqiti, Heckel, and Khan 2013), and of rule with multi objects.

Inference of Rule Features. We are not aware of any approaches in the literature for inferring multi-objects from rules as an advanced feature. However, we share with software testing/verification approaches the technique of using dynamic invariant detection by Daikon (Ernst et al. 2007).

In the next and last chapter, we conclude the work introduced in this thesis with a discussion on the main limitations, ongoing work and our outlook for potential future work and research directions.

Chapter 8

Conclusion and Future Work

This thesis introduced an integrated approach and tool for learning visual contracts from instrumentation of Java code and observation of tests to the derivation of general rules with multi objects and attribute constraints. It supports the analysis of test reports based on a concise, visual and comprehensive representation of operations' behaviour. The remainder of this last chapter concludes the thesis with a summary, our contributions, limitations, and discussion on a possible future work and research directions.

8.1 Summary of the Thesis

The introductory Part I presented the research objectives, motivations, related literature and background that are useful to understand many parts of the thesis, particularly visual contracts and reverse engineering techniques.

In Part II, we presented a running example, illustrated and defined formally by UML and graph transformation concepts to introduce our main technical contributions, published in (Alshanqiti and Heckel 2014; Alshanqiti and Heckel

2015; Alshanqiti, Heckel, and Kehrer 2016; Alshanqiti, Heckel, and Khan 2013). It begins with introducing:

- a dynamic reverse engineering proposal for behaviour models of Java code represented as instantiated versions of visual contracts,
- a graph pattern matching technique for generalising visual contracts, and
- a learning technique for inferring features including multi objects and attribute constraints.

In the concluding Part III, we have evaluated the validity of the resulting models, usability and scalability in experiments on three case studies. The comparisons with related work are also presented in this part.

8.2 Contributions in a Nutshell

The outputs obtained from the investigation process that we went through in this thesis, and from our attempts of addressing the thesis statement (cf. Section 1.2.2), might contribute to the MDE field in general, and significantly to model reverse engineering. These contributions, presented in Part II and Chapter 6, are summarised below.

- A novel approach for extracting visual contract instances based on accessed objects at runtime (cf. Chapter 3).
- A novel approach for extracting maximal rules from observations of graph transformations, based on graph pattern matching (cf. Chapter 4).
- A novel learning approach with an algorithm for inferring multi objects, an advanced feature in graph transformation rules. This increases the generalisation of the extracted specification (cf. Chapter 5).

- A learning approach for inferring rule conditions on parameters and attributes by adopting an invariants detector tool called Daikon (Ernst et al. 2007), discussed in Chapter 5.

- An implementation of all our proposals in a prototype tool as a proof of concept. The source code of the tool is available at GitHub repository[1].

- An evaluation of the usefulness of the approach based on 66 participating MSc students for analysing test reports and identifying faults.

- An evaluation of completeness and correctness of extracted contracts. We have used manual inspection to validate if the contracts extracted satisfy the baseline/moderated notions of correctness and completeness. In addition, our proposed technique allows to improve completeness by adopting incremental observation and learning mechanism.

- Several experiments on three Java applications, (*NanoXML - a small non-validating XML parser for Java* 2016), and (*JHotDraw as Open-Source Project by Java* 2016) and our own Section 3.2, for evaluating performance and scalability of each part of our approach.

8.3 Limitations

There are three drawbacks, discussed in Part II, that prevent us from producing a comprehensive approach for inferring visual contracts. We briefly review these drawbacks in the following subsections.

[1]The source code of our prototype is available at `https://github.com/AMahfodh/IGTRRep`

8.3.1 Observing Deleted Objects

One limitation of the approach, due to the semantics of Java and AspectJ (*AspectJ* 2016), is the inability to detect the deletion of objects. This is handled implicitly by Java's garbage collector. As long as the main application of our technique is in program understanding, this aspect of Java's semantics is reflected correctly in the extracted contracts. Where a more high-level, language-independent model is sought, this limitation may have to be addressed.

8.3.2 Concurrency in Multi-Thread Applications

Observing distributed concurrent system using AspectJ (*AspectJ* 2016) is complicated and may result in generating non-deterministic models. This is because AspectJ supports only sequential weaving, not parallel executions. It matches only a single event at a time (cross-cutting concerns) and the executed join point cannot distinguish among similar accesses to an object in different threads. Since the proposed tracing technique (cf. Chapter 3) is built on AspectJ, this has limited our approach to consider only sequential Java applications.

8.3.3 Dependence on a Single Maximal Rule Extracted

The third limitation of the approach, due to the complexity of constructing maximal rules discussed in Chapter 4, is that we generate a single maximal rule even if this is not unique. While constructing all possible maximal rules could be interesting, the computational effort would be prohibitive and the interpretation of more than one rule generalising all instances would be unclear Section 4.2.2.2.

8.4 Outlook and Future Directions

To extend the work proposed in this thesis, we next present a range of possible future research directions (Alshanqiti, Heckel, and Kehrer 2016), some of which are presently considered in our ongoing work.

8.4.1 Integration with Henshin Tool

Henshin (Arendt et al. 2010) is a visual model transformation language for the formal specification of graph transformation rules. It provides a friendly front-end editor for specifying rules in a descriptive way, allowing to execute them. Its underlying logics and syntax are built based on graph transformation concepts (Ehrig et al. 2006).

The integration of our tool with Henshin allows to: (a) evaluate extracted contracts, and (b) support MBE tools in editing and designing operations. This integration is mostly done (including the import and export features of rules, examples and types in EMF-ECore (Budinsky 2004) formats) but exploring their use is planned as future work. In particular, we will investigate how to exploit the extracted contracts with Henshin to support editing operations and adaptive testing. We discuss these in more details in the following subsections.

8.4.1.1 Edit Operations on Models

Models have to be frequently modified to meet new changing requirements. Re-engineering tasks such as re-factoring, merging or updating parts of a model are supported in MDE tools. This is complex, time-consuming and needs a high level of understanding and knowledge about the specific domain. Therefore, failure to correctly modify models may result in generating inconsistent or invalid models based on their defined meta-model.

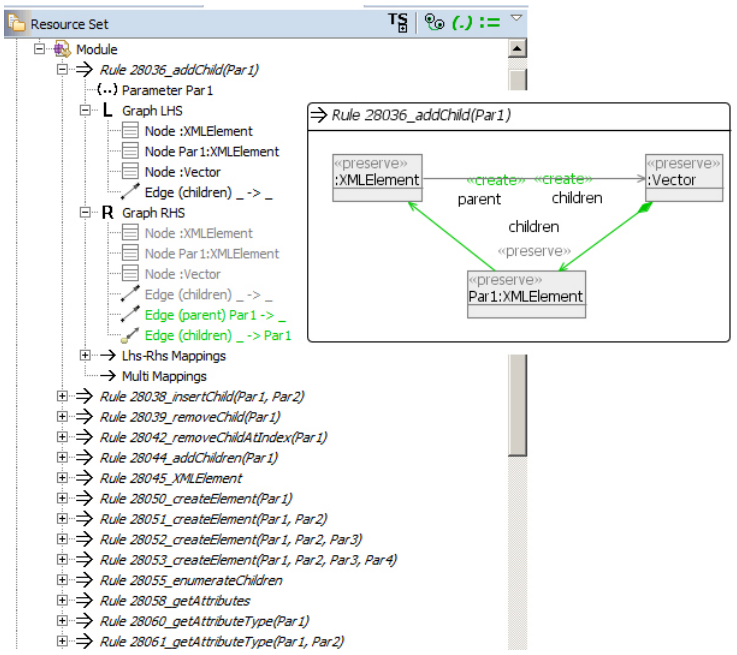


FIGURE 8.1: Henshin representation of extracted contracts from (*NanoXML - a small non-validating XML parser for Java* 2016)

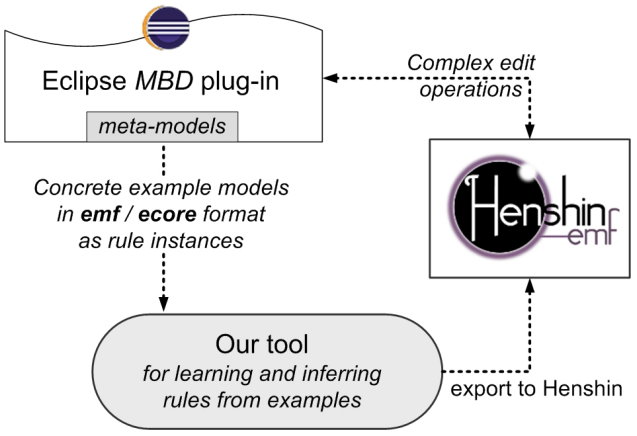


FIGURE 8.2: Integration with Henshin (Arendt et al. 2010) to learn model editing rules

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 (Arendt and Taentzer 2013), or to describe the changes between two versions of a model in a meaningful way (Bürdek et al. 2015; Kehrer, Kelter, and Taentzer 2011).

However, MDE platforms such as the Eclipse Modeling Framework (EMF) offers only a generic low-level API (by means of changing their low-level Abstract Syntax Graph (ASG)) for model modification. Using such API allows to create and delete individual model elements, but in turn, it may violate model constraints according to their meta-models. In order to specify precisely the change in a model, e.g. using EMF Refactor, the rules must be defined. To this end, Kehrer et al (Rindt, Kehrer, and Kelter 2014) propose a meta-tool approach to generate edit rule operations based on EMF Henshin for a given meta-model. Rather than generating edit rules from a metamodel that may affect usability, a better solution would be to focus on concrete models that are closer to the users understanding.

Likewise, editing operations generated from meta-models, e.g. as proposed by (Kehrer et al. 2013; Kehrer et al. 2016), 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 (Alshanqiti, Heckel, and Kehrer 2016). As a consequence, we want to integrate both approaches to learn editing rules as described in Figure 8.2. The aim of this ongoing work is to investigate how to automate manual tasks required for generating general edit rule operations, and evaluate the usefulness of inference of advanced rule features.

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 our tool Section 6.1, 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 (Kappel et al. 2012), the aim here is to stick entirely to the concrete syntax notation domain experts are familiar with.

8.4.1.2 Execution of Inferred Contracts

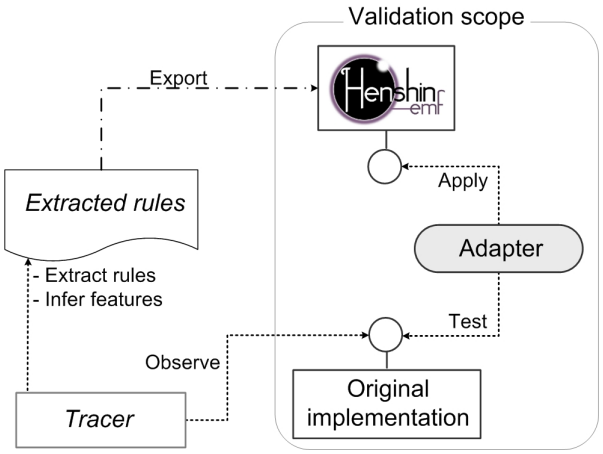


FIGURE 8.3: Integration with Henshin (Arendt et al. 2010) to execute extracted contracts

Figure 8.3 describes the integration of our tool with Henshin that particularly aims at evaluating extracted contracts. This involves invoking the model alongside the original implementation with the same set of tests, comparing outputs for consistency. Executing the tests the contracts were extracted from can improve their correctness, but more interestingly we can try a range of additional cases to evaluate how well contracts capture the wider behaviour, beyond the directly observed.

A related idea is the use for adaptive testing (Cai et al. 2005) where test cases are generated from contracts in a cycle of test generation, execution, and contract extraction. Currently, we work on addressing consistency problem by matching test outputs generated by executing both model and original implementation, see e.g. the generated report from our tool in Appendix C. Checking such

consistency in all probability requires first determining identical started states (or similar start graphs), and then matching all changes after each invocation at the last state.

8.4.2 Inferring Negative Application Conditions

As part of our ongoing work for increasing the accuracy of the extracted specification, we intend to infer Negative Application Conditions (NACs) from constructed contracts. A NAC allows to specify forbidden graph structures before (as pre NAC) or after (post NAC) applying the rule.

We have made a preliminary study on inferring pre NACs from contracts that were extracted from deterministic operations. Our assumption here is limited to consider only minimal rules that are extracted from the same operation but have semantically disjoint preconditions.

In order to address this, we come up with two possible solutions. The first one depends on the inclusion of rules based on graph pattern matching, with the aim of discovering which rules structurally include others. In the second solution, we assume that each rule can be a NAC to the other rules based on determining their priority. This requires a static analysis technique at code level and probably needs extracting CFG to define paths of all rules extracted.

8.4.3 VCs for Debugging

Traditional debugging, as supported by an IDE such as Eclipse or DDD (Zeller 2000), 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.

Using extracted visual contract instances can serve as an alternative approach which can be considered as *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 (Lindeman, Kats, and Visser 2011; Pavletic et al. 2015). Existing approaches, however, mainly focus on debugger frameworks for dedicated domain-specific languages and are not applicable to mainstream Java programs. The monitoring approach presented in (Hamann, Hofrichter, and Gogolla 2012) shares similarities with our idea, but we plan to investigate the usability and scalability of using visual contract instances as an IDE plug-in for run-time debugging.

8.4.4 Supporting Multi-Thread Application

To provide a general solution for learning visual contracts, not limited to sequential applications, we will investigate techniques such as Hyper/J (Ossher and Tarr 2000) for constructing contract instances. Unlike (*AspectJ* 2016), Hyper/J supports multi-dimensional separation of concerns, allowing to instrument and trace multithreaded applications.

8.4.5 On-fly Software Development

With the rapid change in the world of computing, reverse engineering is becoming more sophisticated and connected with many technologies and innovations. Traditionally, it has been used to understand legacy systems, while currently, this trends seem heading to more wider and complex uses. (Canfora, Di Penta, and Cerulo 2011; CanforaHarman and Di Penta 2007) suggest reverse engineering could be a promising method to address problems in the next generation of software technologies. They discuss several emerging trends, one of them is *on-fly-development* that aims at supporting forward engineering with information extracted by reverse engineering as a feedback.

This idea originally discussed in (Müller et al. 2000) argues that involving reverse engineering within forward engineering activities would early give a clear picture of what is designed and created. This may lead to, e.g., reducing development errors by constantly tracking the consistency between model and implementation (Binkley 2007).

In this context, our approach can support manual implementation of operations or method bodies that are hard to generate from models in MDD approach (Stahl, Voelter, and Czarnecki 2006), see Section 2.1.2, while preserving the consistency between models and manual changes in code.

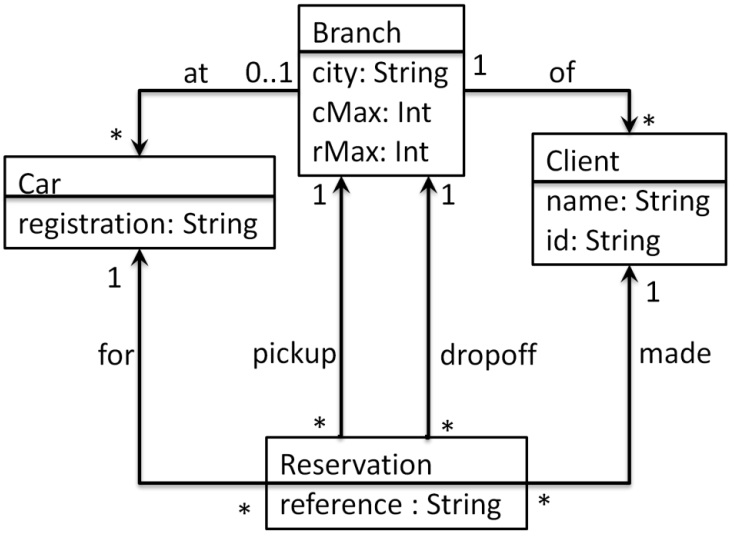
Appendix

Appendix A : Case study to evaluate the use of extracted VC in testing

Service Specification

The specification below describing the interface of a car rental agency service consists of a class diagram modelling the available data, a list of operation signatures and a informal description of the preconditions and effects of those operations.

Data Model:



Operation Signatures:

- registerClient(city: String, client: String): String
- makeReservation(client: String, pickup: String, dropoff: String): String
- pickupCar(reference: String)
- dropoffCar(reference: String)
- cancelReservation(reference: String)

and for the queries:

- showClientReservations(client: String): Reservation[]
- showCars(reservation: String): Car[]
- showBranch(city: String): Branch
- showClients(city: String): Client[]

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.

`Void` **`pickupCar`** (`String reference`)

Removes linkes *pickup* and *for* between reservation object and *pickedup* branch. The reserved *car* can only be picked up once. If there is no suitable reservation, the operation does not have an effect on the state.

Parameters:

`reference` – non-null string value used to get *reservation* object by reference.

Returns:

`no return`

`Void` **`dropoffCar`** (`String reference`)

Creates new link *at* by returning reserved car to the *dropoff* branch, and removes reservation object with all its links, namely: *made*, *pickup*, *dropoff* and *for*. If there is no suitable reservation, the operation does not have an effect on the state.

Parameters:

`reference` – non-null string value used to get *reservation* object by reference.

Returns:

`no return`

`Void` **`cancelReservation`** (`String reference`)

Removes reservation object that matches *`reference`* (if it exists) with all its links, namely: *made*, *pickup*, *dropoff* and *for*. If there is no suitable reservation, the operation does not have an effect on the state.

Parameters:

`reference` – non-null string value used to get *reservation* object by reference.

Returns:

`no return`

```
public class Rental implements IRental{

	private static final long serialVersionUID = 6324598725198583458L;
	...
	public String registerClient(String city, String clientName){

		Branch cBranch = getBranch(city);
		if (cBranch !=null){

			Client newClient = new Client();
			newClient.name =clientName;
			newClient.id = cBranch.city + "_" +  (cBranch.of.size());

			cBranch.of.add(newClient);
			return newClient.id;
		}
		return null;
	}

	public String makeReservation(String ClientID, String pickup, String dropoff){

		Branch pickupBranch = getBranch(pickup);
		Branch dropOffBranch = getBranch(dropoff);

		Client clientMade = getClient(pickupBranch, ClientID);

		if (clientMade==null){
			clientMade = getClient(dropOffBranch, ClientID);
		}

		Car car = getCar(pickupBranch);

		if (pickupBranch==null
				|| dropOffBranch==null
				|| clientMade==null
				|| car==null){
			return null;
		}

		pickupBranch.rMax++;
		Reservation mReservation = new Reservation(
				pickupBranch.city + "_" + pickupBranch.rMax,
				clientMade,
				pickupBranch,
				dropOffBranch,
				car);

		this.reservations.add(mReservation);
		return mReservation.reference;
	}

	public void cancelReservation(String Reference){

		for (int iIndex=this.reservations.size()-1; iIndex>=0; iIndex--){

			Reservation readRes= this.reservations.get(iIndex);
			if (!readRes.made.equalsIgnoreCase(Reference)){
				continue;
			}
			else {
				this.reservations.remove(iIndex);
			}
		}
	}
...
```

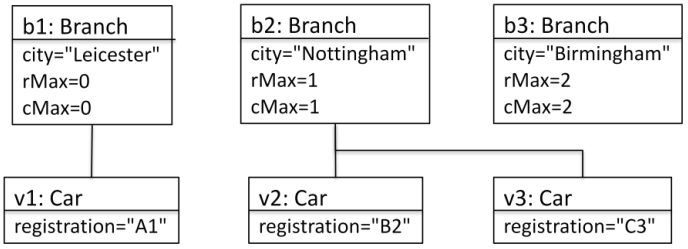
```
66          public void pickupCar(String Reference){
67
68                  int iIndex = getReservationIndex(Reference);
69
70                  if (iIndex==-1){
71                          return;
72                  }
73
74                  Reservation getReservation = this.reservations.get(iIndex);
75
76                  // check if it hasn't been picked up already
77                  if (getReservation.pickup==null){ return; }
78
79                  // check if the reserved car still exists in the pick-up branch
80                  iIndex=-1;
81
82                  for (int iCarIndex=0; iCarIndex <getReservation.pickup.at.size(); iCarIndex++){
83
84                          if (getReservation.pickup.at.get(iCarIndex)
85                                  .getRegistration().equalsIgnoreCase(
86                                  getReservation.for.getRegistration())){
87
88                                  iIndex=iCarIndex;
89                                  break;
90                          }
91                  }
92
93                  if (iIndex==-1){return; }
94
95                  // remove car from pickup branch
96                  getReservation.pickup.at.remove(iIndex);
97          }
98
99          public void dropoffCar(String Reference){
100
101                 int iIndex = getReservationIndex(Reference);
102
103                 if (iIndex==-1){ return;}
104
105                 Reservation getReservation = this.reservations.get(iIndex);
106
107                 // check if it has been picked up already
108                 if (getReservation.pickup!=null || getReservation.made==null){
109                         return;
110                 }
111
112                 // add car to drop-off branch
113                 getReservation.dropoff.at.add(getReservation.for);
114
115                 getReservation.dropoff=null;
116                 getReservation.for=null;
117         }
118
119 }
120
```

Class Test - Not Assessed - Group A

Full Name | | CFS | |

You will be given 5 test reports (each consisting of a sequence of invocations) and the source code of their implementation. Your task is to find out which test reports show failures of the service to satisfy the specification and locate the faults responsible for the failures in the code.

At the start of each test case the database is initialised with 3 branches and 3 cars using the following data:



Question 1) Please use the following table to report the failed steps and explain briefly how the behaviour differs from the specification. The first row has been completed as an example of what is expected.

Failed step only		Brief Justification
Test Report ***No***	Step ***No***	
1	1	No change in attribute *Branch(Leicester).cMax* , which should be increased from 0 to 1.

Failed step only		Brief Justification
Test Report ***No***	Step ***No***	
		....
		....
		....
		....

Question 2) Please locate the faults responsible for the failures you identified under 1). Use line numbers to indicate where these faults occur in the code and explain briefly. Again, the first row provides an example.

Failed step only		line number and Statement if it exists
Test Report ***No***	Step ***No***	
1	1	Between lines (8 and 13): missing a statement to update attribute *cMax*

As a part of our survey, please answer the following questions:

- How many years of experience do you have in OO programming? ☐
- For how long, if at all, have you worked in software development outside university? ☐
- If any, how much of this time (in%) involved testing and debugging? ☐

Test Report 1

		* to show the state after
Step	Operation invocation	Output
1	registerClient("Leicester", "Reiko")	"Leicester_0"
*	showClients("Leicester")	[0]={cName="Reiko", cID="Leicester_0"}
	showBranch("Leicester")	{city=" Leicester", rMax=0, cMax=0}
2	registerClient("Leicester", "Abdullah")	"Leicester_1"
*	showClients("Leicester")	[0]={cName="Reiko", cID="Leicester_0"} [1]={cName="Abdullah", cID="Leicester_1"}
	showBranch("Leicester")	{city=" Leicester", rMax=0, cMax=0}
3	makeReservation("Leicester_1", "Leicester", "Nottingham")	"Leicester_1"
*	showClientReservations("Leicester_1")	[0]={ reference="Leicester_1", made="Leicester_1", pickup="Leicester", dropoff="Nottingham", for="A1"}
	showBranch("Leicester")	{city=" Leicester", rMax=1, cMax=0}
4	makeReservation("Leicester_1", "Birmingham", "Leicester")	null
*	showClientReservations("Leicester_1")	[0]={ reference="Leicester_1", made="Leicester_1", pickup="Leicester", dropoff="Nottingham", for="A1"}
	showBranch("Leicester")	{city=" Leicester", rMax=1, cMax=0}
5	cancelReservation ("Leicester_1")	-
*	showClientReservations("Leicester_1")	null

Test Report 2

		* 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}
2	makeReservation("Nottingham_0", "Nottingham", "Nottingham")	"Nottingham_2"
*	showClientReservations("Nottingham_0")	[0]={reference="Nottingham_2", made="Nottingham_0", pickup=" Nottingham", dropoff="Nottingham", for="B2"}
	showBranch("Nottingham")	{city=" Nottingham", rMax=2, cMax=1}
3	makeReservation("Nottingham_0", "Leicester", "Nottingham")	"Leicester_1"
*	showClientReservations("Nottingham_0")	[0]={reference="Nottingham_2", made="Nottingham_0", pickup=" Nottingham", dropoff="Nottingham", for="B2"} [1]={reference="Leicester_1", made="Nottingham_0", pickup="Leicester", dropoff="Nottingham", for="A1"}
	showBranch("Nottingham")	{city="Nottingham", rMax=2, cMax=1}
	showBranch("Leicester")	{city="Leicester", rMax=1, cMax=0}
4	cancelReservation ("Nottingham_2")	-
*	showClientReservations("Nottingham_0")	null

Test Report 3

Step	Operation invocation	* to show the state after Output
1	registerClient("Leicester", "Reiko")	"Leicester_0"
*	showClients("Leicester")	[0]={cName="Reiko", cID="Leicester_0"}
	showBranch("Leicester")	{city=" Leicester", rMax=0, cMax=0}
2	registerClient("Leicester", "Abdullah")	"Leicester_1"
*	showClients("Leicester")	[0]={cName="Reiko", cID="Leicester_0"} [1]={cName="Abdullah", cID="Leicester_1"}
	showBranch("Leicester")	{city=" Leicester", rMax=0, cMax=0}
3	makeReservation("Leicester_0", "Leicester", "Birmingham")	"Leicester_1"
*	showClientReservations("Leicester_0")	[0]={ reference="Leicester_1", made="Leicester_0", pickup="Leicester", dropoff="Birmingham", for="A1"}
	showBranch("Leicester")	{city="Leicester", rMax=1, cMax=0}
4	makeReservation("Leicester_1", "Leicester", "Leicester")	"Leicester_2"
*	showClientReservations("Leicester_1")	[0]={ reference="Leicester_2", made="Leicester_1", pickup="Leicester", dropoff="Leicester", for="A1"}
	showBranch("Leicester")	{city="Leicester", rMax=2, cMax=0}
5	cancelReservation ("Leicester_1")	-
*	showClientReservations("Leicester_0")	null
	showClientReservations("Leicester_1")	null

Test Report 4

Step	Operation invocation	* to show the state after 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}
3	pickupCar("Nottingham_2")	-
*	showCars("Nottingham")	[0] = {Registration="C3"}
	showCars("Birmingham")	Null
	showClientReservations("Nottingham_0")	[0]={ reference="Nottingham_2" , made=" Nottingham_0" , pickup="Nottingham" , dropoff= "Birmingham", for="B2"}
4	dropoffCar ("Nottingham_2")	-
*	showCars("Nottingham")	[0] = {Registration="C3"}
	showCars("Birmingham")	[0] = {Registration="B2"}
	showClientReservations("Nottingham_0")	[0]={ reference="Nottingham_2" , made=" Nottingham_0" , pickup="Nottingham" , dropoff=null, for=null}

Test Report 5

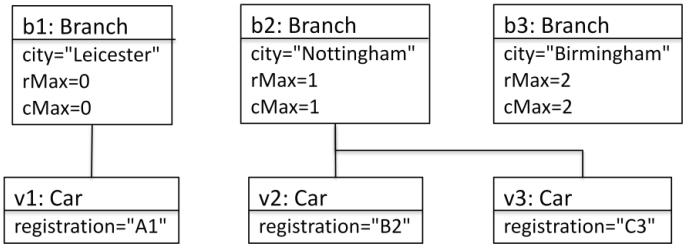
		* to show the state after
Step	Operation invocation	Output
1	registerClient("Birmingham", "Reiko")	"Birmingham_0"
*	showClients("Birmingham")	[0]={cName="Reiko", cID="Birmingham_0"}
	showBranch("Birmingham")	{city=" Birmingham", rMax=2, cMax=2}
2	registerClient("Birmingham ", "Abdullah")	"Birmingham_1"
*	showClients("Birmingham")	[0]={cName="Reiko", cID="Birmingham_0"} [1]={cName=" Abdullah ", cID="Birmingham_1"}
	showBranch("Birmingham")	{city=" Birmingham", rMax=2, cMax=2}
	showCars("Birmingham")	null
3	makeReservation("Birmingham_0", "Birmingham", "Birmingham")	null
*	showClientReservations("Birmingham_0")	null
	showBranch("Birmingham")	{city="Birmingham", rMax=2, cMax=2}
4	makeReservation("Birmingham_1", "Birmingham", "Leicester")	null
*	showClientReservations("Birmingham_1")	null
	showBranch("Birmingham")	{city=" Birmingham", rMax=2, cMax=2}

Class Test - Not Assessed - Group B

Full Name | | CFS | |

You will be given 5 test reports (each consisting of a sequence of invocations) and the source code of their implementation. Your task is to find out which test reports show failures of the service to satisfy the specification and locate the faults responsible for the failures in the code.

At the start of each test case the database is initialised with 3 branches and 3 cars using the following data:



Question 1) Please use the following table to report the failed steps and explain briefly how the behaviour differs from the specification. The first row has been completed as an example of what is expected.

Failed step only		Brief Justification
Test Report ***No***	Step ***No***	
1	1	No change in attribute *Branch(Leicester).cMax* , which should be increased from 0 to 1.

Failed step only		Brief Justification
Test Report ***No***	Step ***No***	
		....
		....
		....
		....

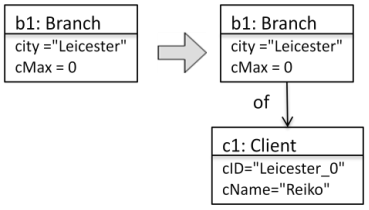
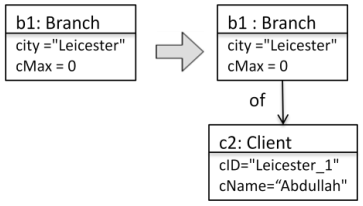
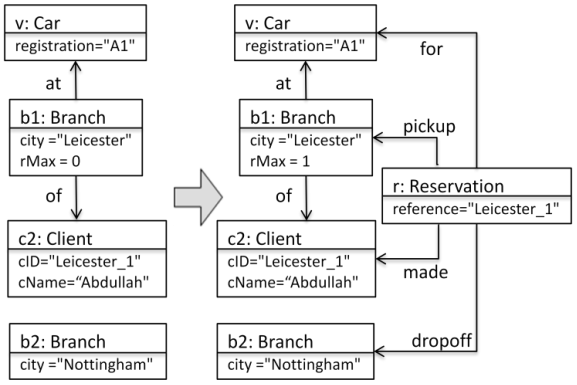
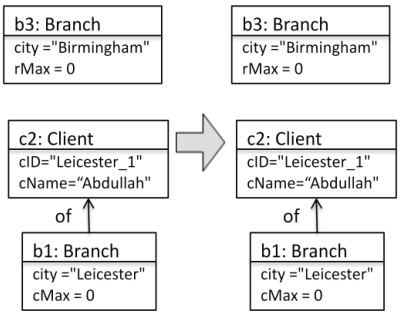
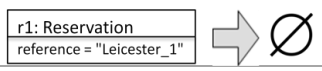
Question 2) Please locate the faults responsible for the failures you identified under 1). Use line numbers to indicate where these faults occur in the code and explain briefly. Again, the first row provides an example.

Failed step only		line number and Statement if it exists
Test Report ***No***	Step ***No***	
1	1	Between lines (8 and 13): missing a statement to update attribute *cMax*

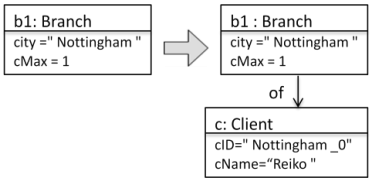
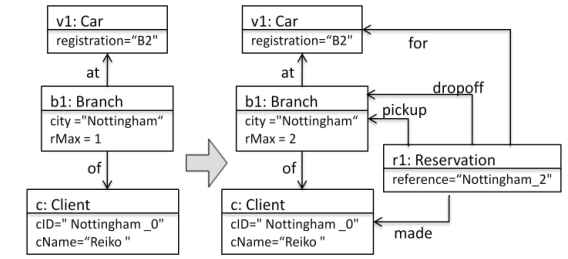
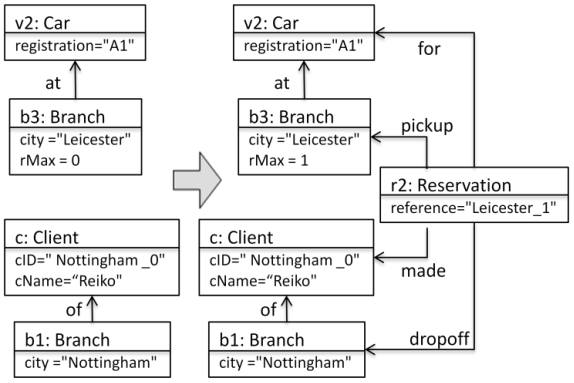
As a part of our survey, please answer the following questions:

- How many years of experience do you have in OO programming? ☐
- For how long, if at all, have you worked in software development outside university? ☐
- If any, how much of this time (in%) involved testing and debugging? ☐

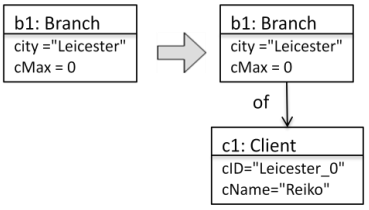
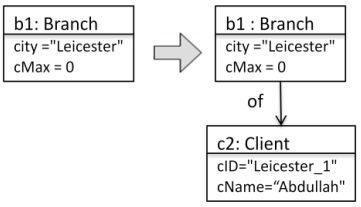
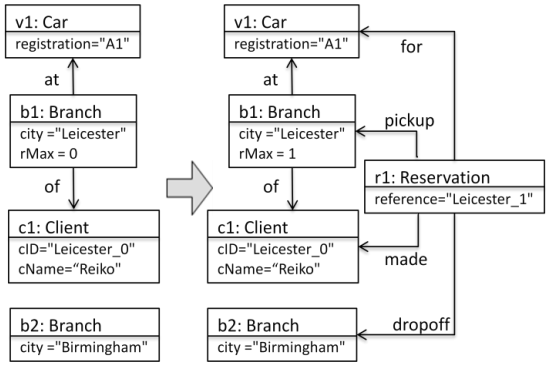
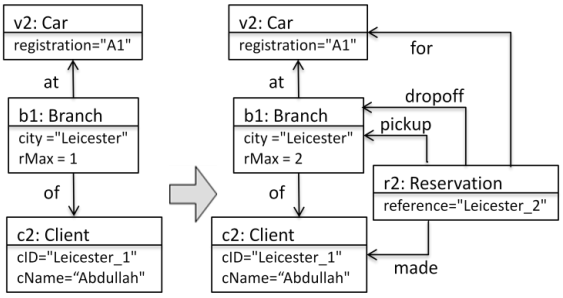
Test Report 1

Step	Operation invocation	Extracted visual contracts	Access in the code *line number*
1	registerClient("Leicester", "Reiko")	Return : "Leicester_0" 	b1:Branch - 6, 11, 13 c1:Client - 9, 10, 11, 14
2	registerClient("Leicester", "Abdullah")	Return : "Leicester_1" 	b1:Branch - 6, 11, 13 c2:Client - 9, 10, 11, 14
3	makeReservation("Leicester_1", "Leicester", "Nottingham")	Return : "Leicester_1" 	b1:Branch - 21, 24, 39, 40 b2:Branch - 22, 40 c2:Client - 24 v:Car - 30, 40 r:Reservation - 40, 47, 48
4	makeReservation("Leicester_1", "Birmingham", "Leicester")	Return : null 	b1: Branch - 22, 27 b3:Branch - 21, 24, 30 c2:Client - 27
5	cancelReservation ("Leicester_1")	No return 	r1:Reservation - 55, 56

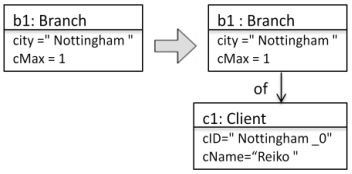
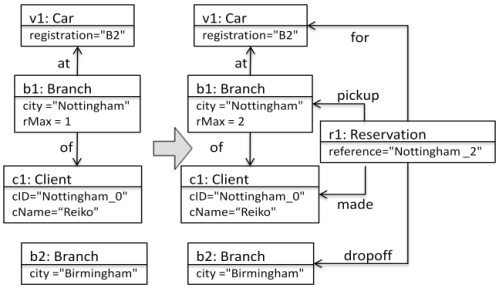
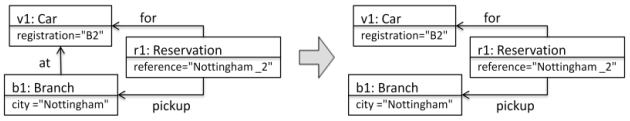
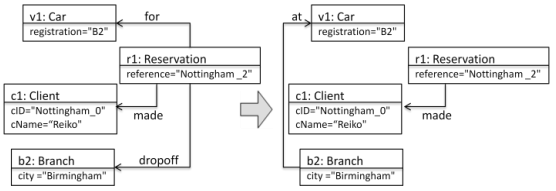
Test Report 2

Step	Operation invocation	Extracted visual contracts	Access in the code *line number*
1	registerClient("Nottingham ", "Reiko")	Return : " Nottingham_0" 	b1:Branch - 6, 11, 13 c:Client - 9, 10, 11, 14
2	makeReservation("Nottingham_0", "Nottingham", "Nottingham")	Return : "Nottingham_2" 	b1: Branch - 21, 22, 24, 39, 40 c:Client - 24 v1:Car - 30, 40 r1:Reservation - 40, 47, 48
3	makeReservation("Nottingham_0", "Leicester", "Nottingham")	Return : "Leicester_1" 	b1:Branch - 22, 27, 40 b3: Branch - 21, 24, 39, 40 c:Client - 24, 27 v2:Car - 30, 40 r2:Reservation - 40, 47, 48
4	cancelReservation ("Nottingham_2")	No return 	r1:Reservation - 55, 56 r2:Reservation - 55, 56

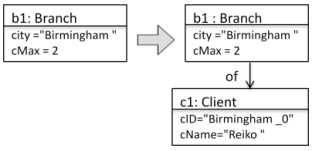
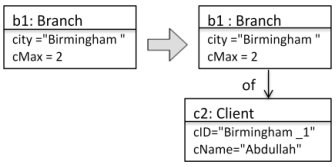
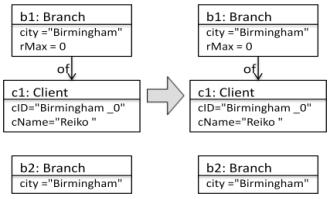
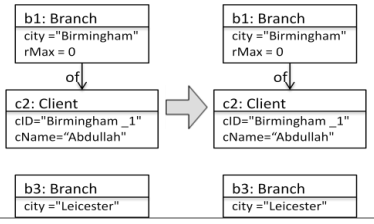
Test Report 3

Step	Operation invocation	Extracted visual contracts	Access in the code *line number*
1	registerClient("Leicester", "Reiko")	Return : "Leicester_0" 	b1:Branch - 6, 11, 13 c1:Client - 9, 10, 11, 14
2	registerClient("Leicester", "Abdullah")	Return : "Leicester_1" 	b1:Branch - 6, 11, 13 c2:Client - 9, 10, 11, 14
3	makeReservation("Leicester_0", "Leicester", "Birmingham")	Return : "Leicester_1" 	b1: Branch - 21, 24, 39, 40 b2:Branch - 22, 40 c1:Client - 24 v1:Car - 30, 40 r1:Reservation - 40, 47, 48
4	makeReservation("Leicester_1", "Leicester", "Leicester")	Return : "Leicester_2" 	b1: Branch - 21, 24, 39, 40 b3:Branch - 22, 40 c2:Client - 24 v2:Car - 30, 40 r2:Reservation - 40, 47, 48
5	cancelReservation ("Leicester_1")	No return 	r1:Reservation - 55, 56 r2:Reservation - 55, 56

Test Report 4

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
3	pickupCar("Nottingham_2")	No return 	b1:Branch - 77, 82, 84, 96 v1:Car - 84 r1:Reservation - 74, 77, 82, 84, 96
4	dropoffCar ("Nottingham_2")	No return 	b2:Branch - 108 v1:Car - 113 c1:Client - 108 r1:Reservation - 105, 108, 113, 115, 116

Test Report 5

Step	Operation invocation	Extracted visual contracts	Access in the code *line number*
1	registerClient("Birmingham", "Reiko")	Return : " Birmingham _0" 	b1:Branch - 6, 11, 13 c1:Client - 9, 10, 11, 14
2	registerClient("Birmingham ", "Abdullah")	Return : " Birmingham _1" 	b1:Branch - 6, 11, 13 c2:Client - 9, 10, 11, 14
3	makeReservation("Birmingham_0", "Birmingham", "Birmingham")	Return : null 	b1: Branch - 21, 24 b2: Branch - 22 c1:Client -24
4	makeReservation("Birmingham_1", "Birmingham", "Leicester")	Return : null 	b1: Branch - 21, 24 b2: Branch - 22 c1:Client -24

Appendix B

Simulate Executing Visual Contracts

Validation Report

Summary

*	Number of rules that are	
A	correct	37
B	not applicable (unmatched structure) [alert case2]	3
C	not applicable (invalid constraints) [alert case1]	0
D	not found [alert case1]	1
E	not executed	1
Total rules in the model		41

Details..

Implementation		Model		Daikon	Evaluation
Test case Id	Rule signature	Rule id	isApplicable	isValid	a.correct b.case 1 c.case 2
tc01	void net.n3.nanoxml.XMLElement.addChild(IXMLElement)	0_28036_addChild	true	?	correct
tc02	void net.n3.nanoxml.XMLElement.insertChild(IXMLElement, int)	1_28038_insertChild	true	?	correct
tc03	void net.n3.nanoxml.XMLElement.addChildren(IXMLElement[])	4_28044_addChildren	false	~ ~	case 2
tc03	void net.n3.nanoxml.XMLElement.addChildren(IXMLElement[])	33_28094_addChildren	false	~ ~	case 2
tc03	void net.n3.nanoxml.XMLElement.addChildren(IXMLElement[])	39_28363_addChildren	false	~ ~	case 2
tc03	void net.n3.nanoxml.XMLElement.addChildren(IXMLElement[])	40_28364_addChildren	true	?	correct
tc04	void net.n3.nanoxml.XMLElement.removeChild(IXMLElement)	2_28039_removeChild	true	?	correct
tc05	void net.n3.nanoxml.XMLElement.removeChildAtIndex(int)	3_28042_removeChildAtIndex	true	?	correct
tc06	Enumeration net.n3.nanoxml.XMLElement.enumerateChildren()	10_28055_enumerateChildren	true	?	correct
tc07	boolean net.n3.nanoxml.XMLElement.hasChildren()	25_28081_hasChildren	true	?	correct
tc08	boolean net.n3.nanoxml.XMLElement.isLeaf()	26_28082_isLeaf	true	?	correct
tc09	IXMLElement net.n3.nanoxml.XMLElement.getChildAtIndex(int)	23_28079_getChildAtIndex	true	?	correct
tc10	int net.n3.nanoxml.XMLElement.getChildrenCount()	27_28083_getChildrenCount	true	?	correct
tc11	IXMLElement net.n3.nanoxml.XMLElement.getParent()	29_28085_getParent	true	?	correct
tc12	String net.n3.nanoxml.XMLElement.getName()	30_28086_getName	true	?	correct
tc13	String net.n3.nanoxml.XMLElement.getFullName()	31_28087_getFullName	true	?	correct
tc14	Vector net.n3.nanoxml.XMLElement.getChildren()	32_28088_getChildren	true	?	correct
tc15	void net.n3.nanoxml.XMLElement.setName(String)	20_28069_setName	true	?	correct
tc16	void net.n3.nanoxml.XMLElement.setName(String, String)	21_28072_setName	true	?	correct
tc17	IXMLElement net.n3.nanoxml.XMLElement.createElement(String)	6_28050_createElement	true	?	correct
tc18	IXMLElement net.n3.nanoxml.XMLElement.createElement(String, String)	7_28051_createElement	true	?	correct
tc19	IXMLElement net.n3.nanoxml.XMLElement.createElement(String, String, int)	8_28052_createElement	true	?	correct
	IXMLElement				

tc20	net.n3.nanoxml.XMLElement.createElement(String, String, String, int)	9_28053_createElement	true	?	correct
tc21	IXMLElement net.n3.nanoxml.XMLElement.createPCDataElement()	34_28163_createPCDataElement	true	?	correct
tc22	void net.n3.nanoxml.XMLElement.setAttribute(String, String)	35_28215_setAttribute	true	?	correct
tc23	void net.n3.nanoxml.XMLElement.setAttribute(String, String, String)	36_28325_setAttribute	true	?	correct
tc24	String net.n3.nanoxml.XMLElement.getAttributeType(String)	12_28060_getAttributeType	true	?	correct
tc25	String net.n3.nanoxml.XMLElement.getAttributeType(String, String)	13_28061_getAttributeType	true	?	correct
tc26	Properties net.n3.nanoxml.XMLElement.getAttributes()	11_28058_getAttributes	true	?	correct
tc27	int net.n3.nanoxml.XMLElement.getAttribute(String, String, int)	15_28063_getAttribute	true	?	correct
tc28	String net.n3.nanoxml.XMLElement.getAttribute(String, String))	14_28062_getAttribute	true	?	correct
tc29	String net.n3.nanoxml.XMLElement.getAttribute(String, String, String)	15_28063_getAttribute	true	?	correct
tc30	Enumeration net.n3.nanoxml.XMLElement.enumerateAttributeNames()	37_28327_enumerateAttributeNames	true	?	correct
tc31	boolean net.n3.nanoxml.XMLElement.hasAttribute(String)	24_28080_hasAttribute	true	?	correct
tc32	int net.n3.nanoxml.XMLElement.getAttributeCount()	28_28084_getAttributeCount	true	?	correct
tc33	String net.n3.nanoxml.XMLElement.getAttributeNamespace(String)	17_28065_getAttributeNamespace	true	?	correct
tc34	Properties net.n3.nanoxml.XMLElement.getAttributesInNamespace(String)	18_28066_getAttributesInNamespace	true	?	correct
tc35	void net.n3.nanoxml.XMLElement.removeAttribute(String)	38_28335_removeAttribute	true	?	correct
tc36	void net.n3.nanoxml.XMLElement.removeAttribute(String, String)	19_28068_removeAttribute	true	?	correct
tc37	Test a non-existent rule	~ ~	~ ~	~ ~	case 1
tc38	void net.n3.nanoxml.XMLElement.setContent(String)	22_28073_setContent	true	?	correct
tc39	void net.n3.nanoxml.XMLElement()	5_28045_XMLElement	true	?	correct

Not executed rules

- 28064_getAttribute

Not found rules

- TestAnonExistentRule

Bibliography

Graph Transformation and Visual Contracts

Agirre, Joseba Andoni, Goiuria Sagardui, and Leire Etxeberria (2014). "Model Transformation by Example Driven ATL Transformation Rules Development Using Model Differences". In: *Software Technologies - 9th International Joint Conference (ICSOFT)*, pages 113–130 (cited on page 135).

Alshanqiti, Abdullah M. and Reiko Heckel (2014). "Towards Dynamic Reverse Engineering Visual Contracts from Java". In: *Electronic Communications of the EASST* 67. URL: http://journal.ub.tu-berlin.de/eceasst/article/view/940 (cited on pages 15, 18, 20, 47, 141).

– (2015). "Extracting Visual Contracts from Java Programs (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 104–114. DOI: 10.1109/ASE.2015.63. URL: http://dx.doi.org/10.1109/ASE.2015.63 (cited on pages 16, 18, 20, 79, 141).

Alshanqiti, Abdullah M., Reiko Heckel, and Timo Kehrer (2016). "Visual contract extractor: a tool for reverse engineering visual contracts using dynamic analysis". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 816–821. URL: http://doi.acm.org/10.1145/2970276.2970287 (cited on pages 19, 20, 142, 145, 147).

Amálio, Nuno and Christian Glodt (2015). "A tool for visual and formal modelling of software designs". In: *Science of Computer Programming* 98, Part 1. Fifth issue of Experimental Software and Toolkits (EST): A special issue on Academics Modelling with Eclipse (ACME2012), pages 52 –79. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2014.05.002. URL: http://www.sciencedirect.com/science/article/pii/S0167642314002305 (cited on pages 7, 32, 43).

Arendt, Thorsten et al. (2010). "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations". In: *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, pages 121–135. DOI: 10.1007/978-3-642-16145-2_9. URL: http://dx.doi.org/10.1007/978-3-642-16145-2_9 (cited on pages 94, 95, 100, 134, 145, 146, 148).

Avazpour, Iman, John Grundy, and Lars Grunske (2015). "Specifying model transformations by direct manipulation using concrete visual notations and interactive recommendations". In: *Journal of Visual Languages & Computing* 28, pages 195–211 (cited on pages 134–136).

Baki, Islem and Houari Sahraoui (2016). "Multi-Step Learning and Adaptive Search for Learning Complex Model Transformations from Examples". In: *ACM Trans. Softw. Eng. Methodol.* 25.3, 20:1–20:37 (cited on pages 134, 135, 137).

Baki, Islem et al. (2014). "Learning Implicit and Explicit Control in Model Transformations by Example". In: *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 636–652. DOI: 10.1007/978-3-319-11653-2_39. URL: http://dx.doi.org/10.1007/978-3-319-11653-2_39 (cited on pages 135, 137).

Balogh, Zoltán and Dániel Varró (2009). "Model transformation by example using inductive logic programming". English. In: *International Journal - Software and Systems Modeling* 8.3 (3), pages 347–364. ISSN: 1619-1366 (cited on pages 135–137).

Baresi, Luciano and Reiko Heckel (2002). "Tutorial Introduction to Graph Transformation: A Software Engineering Perspective". In: *Proceedings of the First International Conference on Graph Transformation*. ICGT '02. London, UK, UK: Springer-Verlag, pages 402–429. ISBN: 3-540-44310-X. URL: http://dl.acm.org/citation.cfm?id=647562.730670 (cited on page 32).

Bisztray, Dénes, Reiko Heckel, and Hartmut Ehrig (2009). "Verification of Architectural Refactorings: Rule Extraction and Tool Support". In: *Proceedings of the Doctoral Symposium at the International Conference on Graph Transformation - Electronic Communications of the EASST* 16 (cited on pages 15, 69).

Bruggink, H.J.Sander (2014). "Towards Process Mining with Graph Transformation Systems". English. In: *Graph Transformation*. Edited by Holger Giese and Barbara König. Volume 8571. Lecture Notes in Computer Science. Springer International Publishing, pages 253–268. ISBN: 978-3-319-09107-5. DOI: 10.1007/978-3-319-

09108-2_17. URL: http://dx.doi.org/10.1007/978-3-319-09108-2_17 (cited on page 131).

Dai, Guilan et al. (2007). "Contract-Based Testing for Web Services". In: *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*. Volume 1, pages 517–526. DOI: 10.1109/COMPSAC.2007.100 (cited on page 44).

Dolques, Xavier et al. (2010). "Learning Transformation Rules from Transformation Examples: An Approach Based on Relational Concept Analysis". In: *Workshops Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference*, pages 27–32 (cited on page 135).

Dolques, Xavier et al. (2011). "Easing model transformation learning with automatically aligned examples". In: *Proceedings of the 7th European conference on Modelling foundations and applications*. ECMFA'11. Birmingham, UK: Springer-Verlag, pages 189–204. ISBN: 978-3-642-21469-1 (cited on pages 131, 135, 137).

Dotti, FernandoLuis et al. (2006). "Verifying Object-based Graph Grammars". English. In: *Software and Systems Modeling* 5.3, pages 289–311. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0014-z (cited on page 44).

Ehrig, H. et al. (2006). *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. pp 12-21-22 Secaucus. NJ, USA: Springer-Verlag New York, Inc. ISBN: 3540311874 (cited on pages 9, 65, 86, 145).

Engels, Gregor, Baris Güldali, and Marc Lohmann (2007). "Towards Model-Driven Unit Testing". English. In: *Models in Software Engineering*. Edited by Thomas Kühne. Volume 4364. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pages 182–192. ISBN: 978-3-540-69488-5. DOI: 10.1007/978-3-540-69489-2_23. URL: http://dx.doi.org/10.1007/978-3-540-69489-2_23 (cited on page 43).

Engels, Gregor et al. (2006a). "Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract". In: *Graph Transformations, Third International Conference, ICGT 2006, Natal, Rio Grande do Norte, Brazil, September 17-23, 2006, Proceedings*, pages 336–350. DOI: 10.1007/11841883_24. URL: http://dx.doi.org/10.1007/11841883_24 (cited on page 4).

– (2006b). "Model-Driven Monitoring: An Application of Graph Transformation for Design by Contract". English. In: *Graph Transformations*. Edited by Andrea Corradini et al. Volume 4178. Lecture Notes in Computer Science. Springer Berlin

Heidelberg, pages 336–350. ISBN: 978-3-540-38870-8. DOI: 10.1007/11841883_24. URL: http://dx.doi.org/10.1007/11841883_24 (cited on pages 41, 43).

Faunes, Martin, Houari Sahraoui, and Mounir Boukadoum (2012). "Generating model transformation rules from examples using an evolutionary algorithm". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, pages 250–253. ISBN: 978-1-4503-1204-2 (cited on pages 136, 137).

Faunes, Martin, Houari A. Sahraoui, and Mounir Boukadoum (2013). "Genetic Programming Approach to Learn Model Transformation Rules from Examples". In: *Theory and Practice of Model Transformations - 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, pages 17–32. DOI: 10.1007/978-3-642-38883-5_2. URL: http://dx.doi.org/10.1007/978-3-642-38883-5_2 (cited on pages 134–137).

Guldali, B. et al. (2009). "Model-Based System Testing Using Visual Contracts". In: *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 121–124. DOI: 10.1109/SEAA.2009.42 (cited on page 44).

Hamann, Lars, Oliver Hofrichter, and Martin Gogolla (2012). "OCL-Based Runtime Monitoring of Applications with Protocol State Machines". In: *Modelling Foundations and Applications - 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*, pages 384–399. DOI: 10.1007/978-3-642-31491-9_29. URL: http://dx.doi.org/10.1007/978-3-642-31491-9_29 (cited on page 150).

Hausmann, Jan Hendrik, Reiko Heckel, and Marc Lohmann (2005). "Model-Based Development of Web Services Descriptions Enabling a Precise Matching Concept". In: *Int. J. Web Service Res.* 2.2, pages 67–84. DOI: 10.4018/jwsr.2005040104. URL: http://dx.doi.org/10.4018/jwsr.2005040104 (cited on page 4).

Heckel, Reiko and Marc Lohmann (2005). "Towards Contract-based Testing of Web Services". In: *Electronic Notes in Theoretical Computer Science* 116. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)Test and Analysis of Component Based Systems 2004, pages 145 – 156. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2004.02.073. URL: http://www.sciencedirect.com/science/article/pii/S1571066104052831 (cited on pages 30, 31, 42, 44).

– (2007). "Model-driven development of reactive information systems: from graph transformation rules to JML contracts". English. In: *International Journal on Software Tools for Technology Transfer* 9.2, pages 193–207. ISSN: 1433-2779. DOI:

10.1007/s10009-006-0020-z. URL: http://dx.doi.org/10.1007/s10009-006-0020-z (cited on pages 26, 41, 43, 44).

Kappel, Gerti et al. (2012). "Conceptual Modelling and Its Theoretical Foundations". In: *Conceptual Modelling and Its Theoretical Foundations*. Edited by Antje Düsterhöft, Meike Klettke, and Klaus-Dieter Schewe. Berlin, Heidelberg: Springer-Verlag. Chapter Model transformation by-example: a survey of the first wave, pages 197–215. ISBN: 978-3-642-28278-2 (cited on pages 134, 135, 148).

Karaorman, Murat and Parker Abercrombie (2005). "jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation". English. In: *Formal Methods in System Design* 27.3, pages 275–312. ISSN: 0925-9856. DOI: 10.1007/s10703-005-3400-1. URL: http://dx.doi.org/10.1007/s10703-005-3400-1 (cited on pages 41–43).

Khan, Tamim Ahmed, Olga Runge, and Reiko Heckel (2012a). "Testing against Visual Contracts: Model-Based Coverage". In: *ICGT*, pages 279–293 (cited on pages 4, 44).

– (2012b). "Visual Contracts as Test Oracle in AGG 2.0". In: *ECEASST* 47 (cited on page 44).

Kühne, Thomas et al. (2016). "Patterns for Constructing Mutation Operators: Limiting the Search Space in a Software Engineering Application". In: *European Conference on Genetic Programming*. Springer, pages 278–293 (cited on pages 134, 135).

Langer, Philip, Manuel Wimmer, and Gerti Kappel (2010). "Model-to-Model Transformations By Demonstration". In: *Proceedings of the Third international conference on Theory and practice of model transformations*. Edited by Laurence Tratt and Martin Gogolla. Volume 6142. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pages 153–167. ISBN: 978-3-642-13687-0 (cited on pages 134–137).

Leavens, Gary T, Albert L Baker, and Clyde Ruby (2006). "Preliminary design of JML: A behavioral interface specification language for Java". In: *ACM SIGSOFT Software Engineering Notes* 31.3, pages 1–38 (cited on pages 29, 41–43, 130).

Lohmann, M., G. Engels, and S. Sauer (2006). "Model-driven Monitoring: Generating Assertions from Visual Contracts". In: *Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International Conference on*, pages 355–356. DOI: 10.1109/ASE.2006.52 (cited on pages 41, 43).

Lohmann, M., S. Sauer, and G. Engels (2005). "Executable visual contracts". In: *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 63–70. DOI: 10.1109/VLHCC.2005.35 (cited on pages 30–32).

Lohmann, Marc, Leonardo Mariani, and Reiko Heckel (2007). "A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services". In: *Test and Analysis of Web Services*. Edited by Luciano Baresi and ElisabettaDi Nitto. Springer Berlin Heidelberg, pages 173–204. ISBN: 978-3-540-72911-2. DOI: 10.1007/978-3-540-72912-9_7. URL: http://dx.doi.org/10.1007/978-3-540-72912-9_7 (cited on page 32).

Mens, Tom and Pieter Van Gorp (2006). "A Taxonomy of Model Transformation". In: *Electron. Notes Theor. Comput. Sci.* 152, pages 125–142. ISSN: 1571-0661 (cited on pages 133, 134).

Meyer, Bertrand (1992). "Applying "Design by Contract"". In: *Computer* 25.10, pages 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279. URL: http://dx.doi.org/10.1109/2.161279 (cited on pages 30, 31, 130).

Porres, Ivan and Irum Rauf (2010). "Generating Class Contracts from Deterministic UML Protocol Statemachines". English. In: *Models in Software Engineering*. Edited by Sudipto Ghosh. Volume 6002. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pages 172–185. ISBN: 978-3-642-12260-6. DOI: 10.1007/978-3-642-12261-3_17. URL: http://dx.doi.org/10.1007/978-3-642-12261-3_17 (cited on page 130).

Rindt, Michaela, Timo Kehrer, and Udo Kelter (2014). "Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools". In: *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014), Valencia, Spain, October 1st and 2nd, 2014*. URL: http://ceur-ws.org/Vol-1255/paper7.pdf (cited on page 147).

Runge, Olga, Tamim Ahmed Khan, and Reiko Heckel (2013). "Test Case Generation Using Visual Contracts". In: *ECEASST* 58 (cited on pages 4, 44).

The Fujaba Tool Suite: From UML to Java and Back Again (2016). URL: http://www.fujaba.de/ (visited on 03/15/2017) (cited on page 41).

Thüm, Thomas et al. (2012). "Applying Design by Contract to Feature-oriented Programming". In: *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*. FASE'12. Tallinn, Estonia: Springer-Verlag, pages 255–269. ISBN: 978-3-642-28871-5. DOI: 10.1007/978-3-642-28872-2_18. URL: http://dx.doi.org/10.1007/978-3-642-28872-2_18 (cited on page 42).

Varró, Gergely, Katalin Friedl, and Dániel Varró (2006). "Implementing a Graph Transformation Engine in Relational Databases". English. In: *International Journal*

- *Software and Systems Modeling* 5.3 (3), pages 313–341. ISSN: 1619-1366 (cited on page 69).

Varró, Dániel (2006). "Model Transformation By Example". In: *In Proceedings of the ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML)*. Springer, pages 410–424 (cited on pages 135–137).

You, Chang hun, Lawrence B. Holder, and Diane J. Cook (2009). "Learning patterns in the dynamics of biological networks". In: *Proceedings of the 15th ACM SIGKDD International conference on Knowledge discovery and data mining*. KDD '09. Paris, France: ACM, pages 977–986. ISBN: 978-1-60558-495-9 (cited on pages 131, 133).

Specification Learning and Mining

Alnusair, Awny and Tian Zhao (2009). "Towards a model-driven approach for reverse engineering design patterns". In: *Proceedings of the 2nd International Workshop on Transforming and Weaving Ontologies in MDE (TWOMDE 2009). Denver, Colorado, USA*. Volume 531 (cited on page 130).

Alshanqiti, Abdullah M, Reiko Heckel, and Tamim Khan (2013). "Learning Minimal and Maximal Rules from Observations of Graph Transformations". In: *Electronic Communications of the EASST* 58 (cited on pages 15, 19, 20, 65, 140, 142).

Artzi, Shay et al. (2006). "Finding the needles in the haystack: Generating legal test inputs for object-oriented programs". In: *M-TOOS 2006: 1st Workshop on Model-Based Testing and Object-Oriented Systems*. Portland, OR, USA (cited on pages 138, 139).

Brito, Hugo et al. (2012). "On-the-fly extraction of hierarchical object graphs". English. In: *Journal of the Brazilian Computer Society*, pages 1–13. ISSN: 0104-6500 (cited on pages 37, 126–129).

Cai, Kai-Yuan et al. (2005). "Adaptive Testing of Software Components". In: *Proceedings of the 2005 ACM Symposium on Applied Computing*. SAC '05. Santa Fe, New Mexico: ACM, pages 1463–1469. ISBN: 1-58113-964-0. DOI: 10.1145/1066677.1067011. URL: http://doi.acm.org/10.1145/1066677.1067011 (cited on page 148).

Conte, D. et al. (2004). "Thirty years of graph matching in pattern recognition". In: *International journal of pattern recognition and artificial intelligence* 18.03, pages 265–298 (cited on pages 17, 41, 138).

Cook, Jonathan E. and Alexander L. Wolf (1998). "Discovering Models of Software Processes from Event-based Data". In: *ACM Trans. Softw. Eng. Methodol.* 7.3, pages 215–249. ISSN: 1049-331X. DOI: 10.1145/287000.287001. URL: http://doi.acm.org/10.1145/287000.287001 (cited on page 40).

Corbett, J.C. et al. (2000). "Bandera: extracting finite-state models from Java source code". In: *ICSE- Proceedings of the 2000 International Conference on Software Engineering*, pages 439–448. DOI: 10.1109/ICSE.2000.870434 (cited on page 130).

Dahm, Nicholas et al. (2015). "Efficient subgraph matching using topological node feature constraints". In: *Pattern Recognition* 48.2, pages 317 –330. ISSN: 0031-3203 (cited on pages 18, 41, 138).

Ernst, Michael D. et al. (2001). "Dynamically Discovering Likely Program Invariants to Support Program Evolution". In: *IEEE Trans. Softw. Eng.* 27.2, pages 99–123. ISSN: 0098-5589. DOI: 10.1109/32.908957. URL: http://dx.doi.org/10.1109/32.908957 (cited on page 89).

Ernst, Michael D. et al. (2007). "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* 69".1–3. Special issue on Experimental Software and Toolkits, pages 35 –45. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2007.01.015. URL: http://www.sciencedirect.com/science/article/pii/S016764230700161X (cited on pages 16, 85, 88, 91, 94, 138, 140, 143).

Ganter, Bernhard et al. (2004). "Concept-Based Data Mining with Scaled Labeled Graphs". In: *Conceptual Structures at Work: 12th International Conference on Conceptual Structures, ICCS 2004, Huntsville, AL, USA, July 19-23, 2004. Proceedings*, pages 94–108 (cited on page 74).

Jouili, S., I. Mili, and S. Tabbone (2009). "Attributed graph matching using local descriptions". In: *Advanced Concepts for Intelligent Vision Systems - Acivs 2009*. Lecture Notes in Computer Science. SEE. Springer, pages 89–99 (cited on pages 41, 138).

Kull, A. (2012). "Automatic GUI Model Generation: State of the Art". In: *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, pages 207–212. DOI: 10.1109/ISSREW.2012.23 (cited on page 130).

Liesaputra, Veronica, Sira Yongchareon, and Sivadon Chaisiri (2015). "Efficient Process Model Discovery Using Maximal Pattern Mining". English. In: *Business Process Management*. Edited by Hamid Reza Motahari-Nezhad, Jan Recker, and Matthias Weidlich. Volume 9253. Lecture Notes in Computer Science. Springer International Publishing, pages 441–456. ISBN: 978-3-319-23062-7. DOI: 10.1007/978-3-319-

23063-4_29. URL: http://dx.doi.org/10.1007/978-3-319-23063-4_29 (cited on pages 40, 41).

Malpani, A. et al. (2010). "Reverse engineering models from databases to bootstrap application development". In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1177–1180. DOI: 10.1109/ICDE.2010.5447776 (cited on page 130).

Mariani, Leonardo (2004). "Capturing and Synthesizing the Behavior of Component-Based Systems". In: *Technical Report LTA:2004:01, DISCO, University of Milano Bicocca. LTA lab* (cited on page 138).

Qiu, M. et al. (2010a). "A New Approach of Graph Isomorphism Detection Based on Decision Tree". In: *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*. Volume 2. IEEE, pages 32–35 (cited on page 40).

– (2010b). "A New Approach of Graph Isomorphism Detection Based on Decision Tree". In: *Education Technology and Computer Science (ETCS), 2010 Second International Workshop on*. Volume 2. IEEE, pages 32–35 (cited on page 137).

Spivey, J Michael and JR Abrial (1992). *The Z Notation: A Reference Manual*. Oriel College, Oxford, OX1-4EW, England: Second Edition, Prentice Hall International (UK) Ltd (cited on pages 7, 32, 43).

Woodcock, Jim et al. (2009). "Formal Methods: Practice and Experience". In: *ACM Comput. Surv.* 41.4, 19:1–19:36. ISSN: 0360-0300. DOI: 10.1145/1592434.1592436. URL: http://doi.acm.org/10.1145/1592434.1592436 (cited on page 32).

Zhao, Chunying, Jun Kong, and Kang Zhang (2010). "Program Behavior Discovery and Verification: A Graph Grammar Approach". In: *IEEE Transactions on Software Engineering* 36.3, pages 431–448. ISSN: 0098-5589 (cited on pages 37, 40, 126–128, 130).

Model Based Software Engineering

Allilaire, Freddy and Tarik Idrissi (2004). "ADT: Eclipse development tools for ATL". In: *Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2)*. Canterbury, University of Kent, UK, pages 171–178 (cited on page 134).

Amar, Bastien et al. (2010). "Using Aspect-Oriented Programming to Trace Imperative Transformations". In: *Enterprise Distributed Object Computing Conference*

(EDOC), 2010 14th IEEE International. IEEE, pages 143–152 (cited on pages 127–129).

Arendt, Thorsten and Gabriele Taentzer (2013). "A tool environment for quality assurance based on the Eclipse Modeling Framework". In: *Automated Software Engineering* 20.2, pages 141–184 (cited on page 146).

Ashish, A.K. and J. Aghav (2013). "Automated techniques and tools for program analysis: Survey". In: *Computing, Communications and Networking Technologies (ICCCNT),2013 Fourth International Conference on*, pages 1–7 (cited on page 8).

Barr, E. T. et al. (2015). "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5, pages 507–525. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2372785 (cited on pages 138, 139).

Bernstein, Phillip A., Alon Y. Halevy, and Rachel A. Pottinger (2000). "A Vision for Management of Complex Models". In: *SIGMOD Rec.* 29.4, pages 55–63. ISSN: 0163-5808. DOI: 10.1145/369275.369289. URL: http://doi.acm.org/10.1145/369275.369289 (cited on page 23).

Berrisford, Graham (2004). "Why IT veterans are sceptical about MDA". In: *Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2)*. Canterbury, University of Kent, UK, pages 125–135 (cited on page 134).

Binkley, David (2007). "Source Code Analysis: A Road Map". In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, pages 104–119. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.27. URL: http://dx.doi.org/10.1109/FOSE.2007.27 (cited on pages 34, 37, 151).

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer (2012). "Model-driven software engineering in practice". In: *Synthesis Lectures on Software Engineering* 1.1, pages 1–182 (cited on pages 23–25).

Budinsky, Frank (2004). *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional (cited on page 145).

Bürdek, Johannes et al. (2015). "Reasoning about product-line evolution using complex feature model differences". In: *Automated Software Engineering*, pages 1–47 (cited on page 146).

Canfora, Gerardo, Massimiliano Di Penta, and Luigi Cerulo (2011). "Achievements and Challenges in Software Reverse Engineering". In: *Commun. ACM* 54.4, pages 142–151. ISSN: 0001-0782. DOI: 10.1145/1924421.1924451. URL: http://doi.acm.org/10.1145/1924421.1924451 (cited on pages 8, 18, 33–40, 151).

CanforaHarman, Gerardo and Massimiliano Di Penta (2007). "New Frontiers of Reverse Engineering". In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, pages 326–341. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.15. URL: http://dx.doi.org/10.1109/FOSE.2007.15 (cited on pages 33, 36, 37, 40, 151).

Chen, Feng et al. (2006). "A Formal Model Driven Approach to Dependable Software Evolution". In: *Computer Software and Applications Conference, 2006. COMPSAC '06. 30th Annual International*. Volume 1, pages 205–214. DOI: 10.1109/COMPSAC.2006.10 (cited on pages 27, 28, 35, 126, 127, 134).

Chikofsky, E.J. and II Cross J.H. (1990). "Reverse engineering and design recovery: a taxonomy". In: *Software, IEEE* 7.1, pages 13–17. ISSN: 0740-7459. DOI: 10.1109/52.43044 (cited on pages 5, 33, 40).

Cipolla-Ficarra, Francisco Vicente (2010). *Quality and Communicability for Interactive Hypermedia Systems: Concepts and Practices for Design: Concepts and Practices for Design*. IGI Global (cited on page 38).

Claudia, P., M. Liliana, and F. Liliana (2011). "Recovering Use Case Diagrams from Object Oriented Code: An MDA-based Approach". In: *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*. IEEE, pages 737–742 (cited on page 129).

Cornelissen, B. et al. (2009). "A Systematic Survey of Program Comprehension through Dynamic Analysis". In: *Software Engineering, IEEE Transactions on* 35.5, pages 684–702. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.28 (cited on pages 8, 18, 37, 38, 65, 121).

Czarnecki, K. and S. Helsen (2006). "Feature-based Survey of Model Transformation Approaches". In: *IBM Syst. J.* 45.3, pages 621–645. ISSN: 0018-8670. DOI: 10.1147/sj.453.0621. URL: http://dx.doi.org/10.1147/sj.453.0621 (cited on page 134).

Duarte, Lucio Mauro, Jeff Kramer, and Sebastian Uchitel (2006). "Model Extraction Using Context Information". In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. MoDELS'06. Genova, Italy: Springer-Verlag, pages 380–394. ISBN: 3-540-45772-0, 978-3-540-45772-5. DOI: 10.1007/11880240_27. URL: http://dx.doi.org/10.1007/11880240_27 (cited on page 128).

Eilam, Eldad (2011). *Reversing: secrets of reverse engineering*. John Wiley & Sons. ISBN: 9781118079768. URL: https://books.google.co.uk/books?id=_78HnPPRU_oC (cited on page 34).

Elberzhager, Frank, Jürgen Münch, and Vi Tran Ngoc Nha (2012). "A Systematic Mapping Study on the Combination of Static and Dynamic Quality Assurance Techniques". In: *Information and Software Technology* 54.1, pages 1–15. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2011.06.003. URL: http://dx.doi.org/10.1016/j.infsof.2011.06.003 (cited on page 39).

Ernst, Michael D (2003). "Static and dynamic analysis: Synergy and duality". In: *WODA 2003: ICSE Workshop on Dynamic Analysis*. Citeseer. Citeseer, pages 24–27 (cited on pages 34, 36–39).

Favre, Jean-Marie (2004). "Foundations of model (driven)(reverse) engineering: Models". In: *Proceedings of the International Seminar on Language Engineering for Model-Driven Software Development, Dagstuhl Seminar 04101*. available at http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.9106&rep=rep1&type=pdf (cited on pages 23, 27, 28).

Favre, Liliana, Liliana Martinez, and Claudia Pereira (2009). "MDA-Based Reverse Engineering of Object Oriented Code". English. In: *Enterprise, Business-Process and Information Systems Modeling*. Edited by Terry Halpin et al. Volume 29. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, pages 251–263. ISBN: 978-3-642-01861-9. DOI: 10.1007/978-3-642-01862-6_21. URL: http://dx.doi.org/10.1007/978-3-642-01862-6_21 (cited on page 129).

Galvao, Ismenia and Arda Goknil (2007). "Survey of traceability approaches in model-driven engineering". In: *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE, pages 313–313 (cited on page 127).

Hassan, A.E. et al. (2008). "An Industrial Case Study of Customizing Operational Profiles Using Log Compression". In: *Software Engineering, 2008. ICSE '08. ACM / IEEE 30th International Conference on*, pages 713–723. DOI: 10.1145/1368088.1379445 (cited on page 39).

Kagdi, Huzefa, Michael L. Collard, and Jonathan I. Maletic (2007). "A survey and taxonomy of approaches for mining software repositories in the context of software evolution". In: *Journal of Software Maintenance and Evolution: Research and Practice* 19.2, pages 77–131. ISSN: 1532-0618. DOI: 10.1002/smr.344. URL: http://dx.doi.org/10.1002/smr.344 (cited on pages 39, 40).

Kehrer, Timo, Udo Kelter, and Gabriele Taentzer (2011). "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*, pages 163–172 (cited on page 146).

Kehrer, Timo et al. (2013). "Generating Edit Operations for Profiled UML Models". In: *Proc. Intl. Workshop on Models and Evolution*. Volume 1090. CEUR Workshop Proceedings, pages 30–39 (cited on page 147).

Kehrer, Timo et al. (2016). "Automatically Deriving the Specification of Model Editing Operations from Meta-Models". In: *Proc. 9th Intl. Conf. on Model Transformations*. Springer, pages 173–188 (cited on page 147).

Kipyegen, Noela Jemutai, WP Korir, and Kenya Njoro (2013). "Importance of Software Documentation". In: *International Journal of Computer Science Issues, IJCSI* 10.5, pages 223–228 (cited on page 5).

Kramer, R. (1998). "iContract - The Java(tm) Design by Contract(tm) Tool". In: *TOOLS 1998: 26th International Conference on Technology of Object-Oriented Languages and Systems, 3-7 August 1998, Santa Barbara, CA, USA*, pages 295–307. URL: http://dx.doi.org/10.1109/TOOLS.1998.711021 (cited on page 30).

Kühne, Thomas (2006). "Matters of (Meta-)Modeling". In: *Software and System Modeling* 5.4, pages 369–385. DOI: 10.1007/s10270-006-0017-9. URL: http://dx.doi.org/10.1007/s10270-006-0017-9 (cited on page 23).

Lindeman, Ricky T., Lennart C. L. Kats, and Eelco Visser (2011). "Declaratively defining domain-specific language debuggers". In: *Proc. 10th Intl. Conf. on Generative Programming and Component Engineering*, pages 127–136. URL: http://doi.acm.org/10.1145/2047862.2047885 (cited on page 150).

Martinez, L, C Pereira, and L Favre (2011). "Recovering Activity Diagrams from Object Oriented Code: an MDA-based Approach". In: *Proceedings 2011 International Conference on Software Engineering Research and Practice (SERP 2011)*. Volume 1, pages 58–64 (cited on page 129).

Mellor, S.J., A.N. Clark, and T. Futagami (2003). "Model-driven development - Guest editor's introduction". In: *Software, IEEE* 20.5, pages 14–18. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231145 (cited on page 26).

Muhairat, Mohammad I. and Rafa E. Al-Qutaish (2009). "An Approach to Derive the Use Case Diagrams from an Event Table". In: *Proceedings of the 8th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*. SEPADS'09. Cambridge, UK: World Scientific, Engineering Academy, and Society (WSEAS), pages 33–38. ISBN: 978-960-474-052-9. URL: http://dl.acm.org/citation.cfm?id=1553890.1553897 (cited on page 129).

Müller, Hausi A et al. (2000). "Reverse engineering: A roadmap". In: *Proceedings of the Conference on the Future of Software Engineering, ACM*. ACM, pages 47–60 (cited on page 151).

Ossher, Harold and Peri Tarr (2000). "Hyper/J: Multi-dimensional Separation of Concerns for Java". In: *Proceedings of the 22Nd International Conference on Software Engineering*. ICSE '00. Limerick, Ireland: ACM, pages 734–737. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337618. URL: http://doi.acm.org/10.1145/337180.337618 (cited on page 150).

Pavletic, Domenik et al. (2015). "Extensible debugger framework for extensible languages". In: *Reliable Software Technologies–Ada-Europe 2015*. Springer, pages 33–49 (cited on page 150).

Pender, Tom (2003). *UML Bible*. 1st edition. New York, NY, USA: John Wiley & Sons, Inc. ISBN: 0764526049, 9780764526046 (cited on pages 23, 24).

Pistoia, M. and O. Tripp (2014). *Eliminating false-positive reports resulting from static analysis of computer software*. US Patent 8,745,578. URL: https://www.google.com/patents/US8745578 (cited on page 8).

Pradel, Michael and Thomas R. Gross (2009). "Automatic Generation of Object Usage Specifications from Large Method Traces". In: *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 371–382. DOI: 10.1109/ASE.2009.60. URL: http://dx.doi.org/10.1109/ASE.2009.60 (cited on page 130).

Priestly, Mark (2005). "The Logic of Correctness in Software Engineering". In: *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings of the CAiSE'05 Workshops, Vol. 2*, pages 463–473 (cited on page 98).

Ramos, A.L., J.V. Ferreira, and J. Barcelo (2012). "Model-Based Systems Engineering: An Emerging Approach for Modern Systems". In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 42.1, pages 101–111. ISSN: 1094-6977. DOI: 10.1109/TSMCC.2011.2106495 (cited on pages 2, 22, 23, 25).

Rountev, Atanas, Olga Volgin, and Miriam Reddoch (2005). "Static Control-flow Analysis for Reverse Engineering of UML Sequence Diagrams". In: *SIGSOFT Softw. Eng. Notes* 31.1, pages 96–102. ISSN: 0163-5948 (cited on pages 8, 36, 126).

Sarkar, Mrinal Kanti, Trijit Chatterjee, and Dipta Mukherjee (2013). "Reverse Engineering: An Analysis of Static Behaviors of Object Oriented Programs by Extracting UML Class Diagram". In: *International Journal of Advanced Computer Research* 3.3. ISSN: 2249-7277 (cited on pages 36, 126).

Schmidt, D.C. (2006). "Guest Editor's Introduction: Model-Driven Engineering". In: *Computer* 39.2, pages 25–31. ISSN: 0018-9162 (cited on pages 25, 26).

Seidewitz, Ed (2003). "What Models Mean". In: *IEEE Softw.* 20.5, pages 26–32. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231147. URL: http://dx.doi.org/10.1109/MS.2003.1231147 (cited on pages 23, 24, 28).

Siegel, Jon M. (2014). *Object Management Group - Model Driven Architecture (MDA) Guide rev. 2.0.* This final draft of the revised MDA Guide edited in Boston on 18 June 2014 reflects all AB edits requested at the Reston and Boston AB meetings. URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01 (visited on 03/15/2017) (cited on pages 2, 23, 26–29, 42, 126, 133, 134).

Sommerville, Ian (2011). *Software Engineering Ninth Edition.* Addison-Wesley (cited on pages 2, 3, 25–27).

Stahl, Thomas, Markus Voelter, and Krzysztof Czarnecki (2006). *Model-Driven Software Development: Technology, Engineering, Management.* John Wiley & Sons. ISBN: 0470025700 (cited on pages 25, 26, 151).

The Architecture-Driven Modernization Task Force (ADMTF) (2016). URL: http://adm.omg.org/ (visited on 03/15/2017) (cited on page 127).

The Java Modeling Language (JML) (2016). URL: http://www.eecs.ucf.edu/~leavens/JML//index.shtml (visited on 03/15/2017) (cited on pages 29, 41).

The Unified Modeling Language (UML) (2016). URL: http://www.uml.org/ (visited on 03/15/2017) (cited on page 29).

Tonella, P. and A. Potrich (2003). "Reverse engineering of the interaction diagrams from C++ code". In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 159–168 (cited on pages 36, 37, 126).

Venkatasubramanyam, Radhika D. and Sowmya G. R. (2014). "Why is Dynamic Analysis Not Used As Extensively As Static Analysis: An Industrial Study". In: *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices.* SER&IPs 2014. Hyderabad, India: ACM, pages 24–33. ISBN: 978-1-4503-2859-3. DOI: 10.1145/2593850.2593855. URL: http://doi.acm.org/10.1145/2593850.2593855 (cited on pages 36–38).

Wasylkowski, Andrzej, Andreas Zeller, and Christian Lindig (2007). "Detecting object usage anomalies". In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 35–44. DOI: 10.1145/1287624.1287632. URL: http://doi.acm.org/10.1145/1287624.1287632 (cited on pages 128, 129).

Whittle, J., J. Hutchinson, and M. Rouncefield (2014). "The State of Practice in Model-Driven Engineering". In: *Software, IEEE* 31.3, pages 79–85. ISSN: 0740-7459. DOI: 10.1109/MS.2013.65 (cited on pages 2, 25, 26, 28).

Zeller, Andreas (2000). "Debugging with DDD". In: *User's Guide and Reference Manual, Version* 3 (cited on page 149).

Ziadi, Tewfik et al. (2011). "A fully dynamic approach to the reverse engineering of UML sequence diagrams". In: *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, pages 107–116 (cited on pages 37, 126, 128, 129).

Miscellaneous

AspectJ (2016). URL: http://eclipse.org/aspectj/ (visited on 03/15/2017) (cited on pages 47, 48, 56, 139, 144, 150).

BTrace (2017). URL: https://kenai.com/projects/btrace (visited on 03/15/2017) (cited on page 48).

Byteman Trace (2017). URL: http://byteman.jboss.org/ (visited on 03/15/2017) (cited on page 48).

InTrace (2017). URL: https://mchr3k.github.io/org.intrace/ (visited on 03/15/2017) (cited on page 48).

JHotDraw as Open-Source Project by Java (2016). URL: http://www.jhotdraw.org/ (visited on 03/15/2017) (cited on pages 19, 114, 143).

Laddad, Ramnivas (2009). *AspectJ in Action: Enterprise AOP with Spring Applications*. 2nd. Greenwich, CT, USA: Manning Publications Co. ISBN: 1933988053, 9781933988054 (cited on page 56).

Marin, M. (2004). "Refactoring JHotDraw's Undo concern to AspectJ". In: *Proceedings Workshop on Aspect Reverse Engineering (WARE) at WCRE* (cited on page 123).

Method Tracing (2017). URL: http://blog.rejeev.com/2009/04/method-tracing.html (visited on 03/15/2017) (cited on page 48).

NanoXML - a small non-validating XML parser for Java (2016). URL: http://nanoxml.sourceforge.net/orig/ (visited on 03/15/2017) (cited on pages 19, 60, 114, 143, 146).