



Proceedings of the
12th International Workshop on Graph Transformation
and Visual Modeling Techniques
(GTVMT 2013)

Test Case Generation Using Visual Contracts

Olga Runge, Tamim Ahmed Khan and Reiko Heckel

12 pages

Test Case Generation Using Visual Contracts

Olga Runge¹, Tamim Ahmed Khan² and Reiko Heckel³

¹ o.runge@mailbox.tu-berlin.de

Department of Software Engineering and Theoretical Computer Science, TU-Berlin, Germany

² tamim@bui.edu.pk

Department of Computer and Software Engineering, Bahria University, Pakistan

³ reiko@mcs.le.ac.uk

Department of Computer Science, Leicester University, UK

Abstract: Visual contracts provide a diagrammatic notation for pre- and post-conditions as alternative to the Object-Constraint Language (OCL) or code-level contract languages. Using visual contracts for testing, we benefit from their executability and formal background in graph transformation to provide model-based test oracles and coverage criteria. Based on a static analysis of their dependencies and conflicts, in this paper we use visual contracts to generate test cases according to these coverage criteria.

Together with previous work, this adds up to a comprehensive approach aiming to automate the three major challenges of testing through the use of models.

Keywords: graph transformation, services, visual contracts, test case generation

1 Introduction

Testing involves a variety of activities, including test case generation to create a test suite, coverage analysis to assess its quality, and oracles to predict expected results. Like black-box methods in general, model-based approaches do not require access to source code and are therefore suitable for interface-based testing of components or services.

Visual contracts were developed for interface specification in [HHL05] and have been used for model-based testing in [LMH07, GMWE09]. Using a formal interpretation in terms of typed graph transformation, they are executable and hence suitable for the generation of test oracles [KRH12b]. Theory and tools of graph transformation also provide support for the definition and evaluation of coverage criteria [KRH12a], but the generation of test cases based on these criteria remains an open problem.

In this paper we focus on dependency cover in order to derive test cases, defined on the dependency graph DG extracted from a set of visual contracts. Coverage is achieved if all dependencies in DG are observed at runtime [KRH12a]. By generating sequences of rules likely to exercise these dependencies and validating their executability on the model, we can produce test cases that are guaranteed to achieve full coverage if they are fully executable on the system under test (SUT).

Given a set of visual contracts and their dependency graph, the generation is an iteration of three steps. First, acyclic paths from a suitable start rule in the dependency graph are generated, augmented by additional rules to cater for cases with multiple dependencies. Second, these sequences are validated over the model by executing the rules. If they are not executable, the sequences are dropped. Information from the dependency analysis about overlaps of matches and co-matches is used to determine partial matches, which are completed randomly. In this way we are sure to obtain sequences that exercise dependencies. As part of the validation, executable invocation sequences and coverage are recorded. If coverage is complete, the generation terminates, otherwise there is another iteration. We also terminate when no further progress is made.

After introducing some background on visual contracts in Section 2 and defining dependency cover in Section 3, we present the algorithm in more detail in Section 4 and evaluate its quality and scalability in Section 5. The paper concludes with a discussion of related work in Section 6 and of limitations and possible extensions in Section 7.

2 Visual Contracts

A visual contract represents pre- and post-conditions of an operation [HHL05] as a pair of object diagrams. We formalise visual contracts as rules in a typed attributed graph transformation system with rule signatures (TAGTS), as shown in Figure 1. As a running example we consider a service for managing hotel guests. A registered guest can book a room subject to availability. There are no booking charges and the bill starts to accumulate once the room is occupied. Since payment details are already with the hotel, the bill is automatically deducted when the guest announces their intention to leave. They can check out successfully only when the bill is paid. The type graph for this system is shown in Fig. 2(a) using AGG [AGG07] notation.

Formally, such a model is represented by a typed attributed graph transformation system with rule signatures, consisting of an attributed type graph, rule names with parameter declarations, and for each name a set of rules representing the different outcomes of the operation [HKM11].

Definition 1 (TAGTS with rule signatures) A typed attributed graph transformation system with rule signatures is a tuple $\mathcal{G} = (TG, P, X, \pi, \sigma)$ where TG is an attributed type graph, P is a countable set of rule names, X is a set of variables, π and σ assign to each rule name p a finite set of rules $\pi(p)$ over TG with local attribute variables in X and a list of formal input and output parameters $\sigma(p) = \bar{x} = (q_1x_1 : s_1, \dots, q_nx_n : s_n)$ where $q_i \in \{\varepsilon, out\}$ and $x_i \in X_{s_i}$ for $1 \leq i \leq n$. We write p 's rule signature as $p(\bar{x})$.

That means, we associate a rule signature with each visual contract [KRH12b], consisting of the name of the contract and formal input and output parameters that refer to variables in attribute expressions. Consider Figure 1, where the signature $bookRoom(r:int, n:String)$ has parameters r and n , also used in contract $bookRoom$ to represent the possible attribute values for *room* and *name*. This allows us to relate visual contracts to operations in the system under test, represent invocations of operations and their actual parameters and results at the model level by observations on transformations, their matches and co-matches [KRH12a]. For example, given the transformation sequence in Figure 3(a), its observation is shown in Figure 3(b), with actual

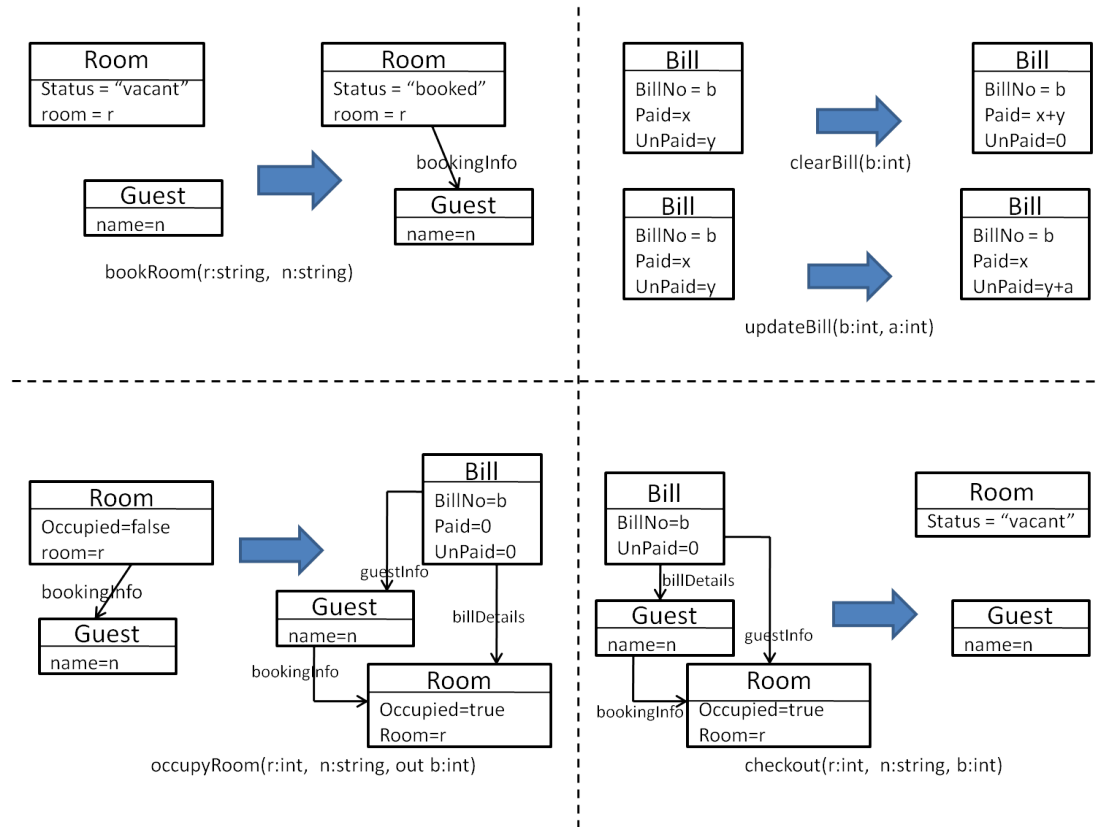


Figure 1: Visual contracts for the Hotel example

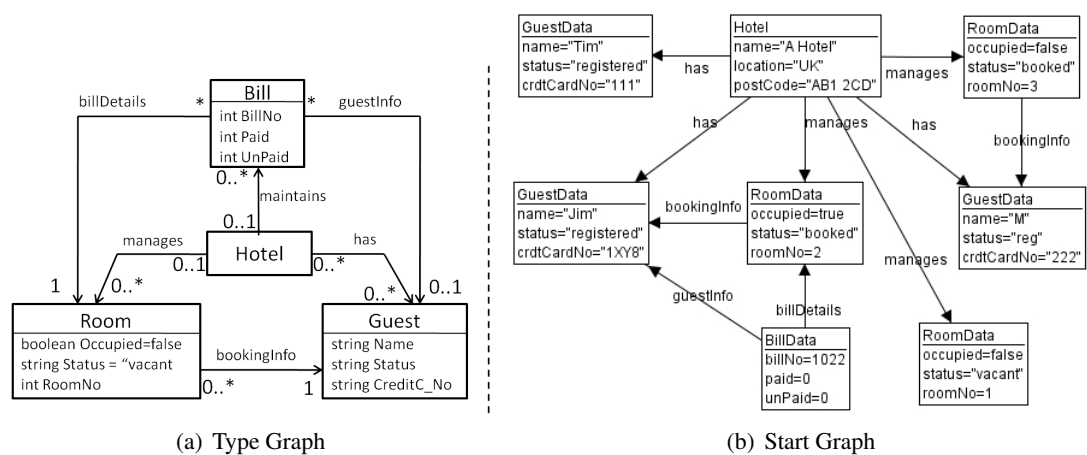


Figure 2: Type graph (a) and start graph (b)

parameters $n = \text{"Tim"}$ and $r = 1$.

Instance graphs, typed over the type graph, represent sample states of a hotel with only one

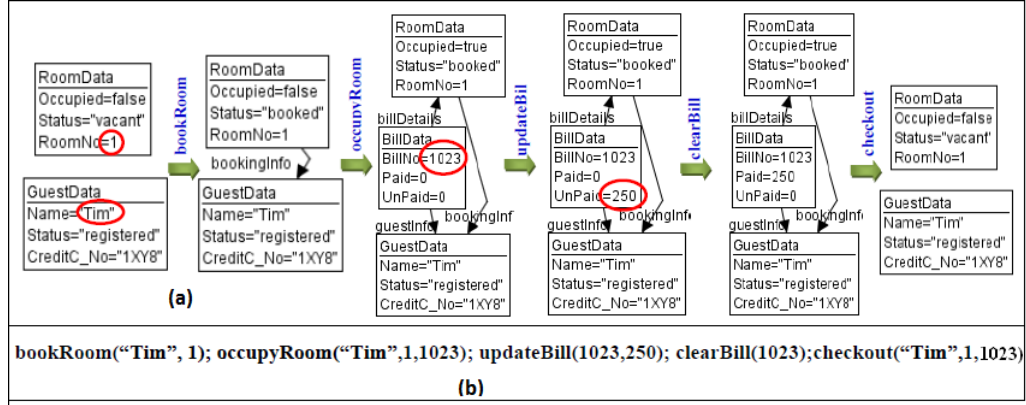


Figure 3: An example transformation sequence and its observation

room and one registered guest. A test case combines a graph defining its initial state with a sequence of invocations, i.e., rule names with input parameters instantiated either by constant values or output parameters of earlier invocations.

Example 1 (test case) Considering the transformation sequence in Figure 3 (a), a test case $t = (G_0, s)$ is given by the graph G_0 in Figure 2(b) and sequence s below.

$$s = \text{bookRoom}(\text{"Tim"}, 1); \text{occupyRoom}(\text{"Tim"}, 1, bNo); \text{updateBill}(bNo, 250); \\ \text{clearBill}(bNo); \text{checkout}(\text{"Tim"}, 1, bNo)$$

A test case, once executed, becomes an observation sequence as shown in Figure 3 (b) by instantiating invocations. For sequence s above and $bNo = 1023$ we obtain observation sequence

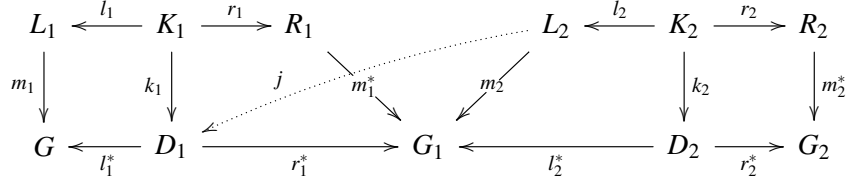
$$\text{bookRoom}(\text{"Tim"}, 1); \text{occupyRoom}(\text{"Tim"}, 1, 1023); \text{updateBill}(1023, 250); \\ \text{clearBill}(1023); \text{checkout}(\text{"Tim"}, 1, 1023)$$

3 Dependencies and Coverage

In this section, we show how to extract a dependency graph for a system under test (SUT) from the available interface specification based on visual contracts. A dependency graph (DG) provides us with a visual representation of dependencies allowing us to study coverage criteria at the interface level. The nodes represent rules while edges indicate the dependencies between them. The edges also bear annotations at the start and the end of the edge to show if the data was created, read, updated or deleted by these rule applications.

Definition 2 (asymmetric dependencies) Given two rules p_1, p_2 , We say that p_1 may enable p_2 , written $p_1 \prec p_2$, if there are steps $G_0 \xrightarrow{p_1, m_1} G_1 \xrightarrow{p_2, m_2} G_2$ without $j : L_2 \rightarrow D_1$ such that $m_2 = r_1^* \circ j$.

In this case, we say that the second step *requires* the first.



For example we find a dependency $bookRoom \prec occupyRoom$. Using this definition, we define a dependency graph as follows.

Definition 3 (dependency graph) A dependency graph $DG = \langle G, OP, op, lab \rangle$ is a structure where

- $G = \langle V, E, src, tar \rangle$ is a graph,
- OP is a set of (names of) operations,
- $op : V \rightarrow OP$ maps vertices to operation names,
- $lab : E \rightarrow \{c, r, d\} \times \{c, r, d\}$ is a labeling function distinguishing source and target types create, read, delete

As anticipated, rules are represented by nodes labeled by rule names, while edges represent dependencies between them.

Definition 4 (dependency graph of TAGTS with rule signatures) Given a TAGTS with rule signatures $\mathcal{G} = (TG, P, X, \pi, \sigma)$, its dependency graph $DG(\mathcal{G}) = \langle G, OP, op, lab \rangle$ with $G = (V, E, src, tar)$ is defined by

- $V = \bigcup_{p \in P} (\{p\} \times \pi(p))$ as the set of all rules tagged by their names. If $s_1 \in \pi(p)$ we write $p_1 : s_1 \in V$.
- $E \subseteq V \times V$ such that:
 - $e = (p_1 : s_1, p_2 : s_2) \in E$ if $p_1 : s_1$ may enable $p_2 : s_2$, i.e., there are steps $G \xrightarrow{p_1:s_1,m_1} H_1 \xrightarrow{p_2:s_2,m_2} H_2$ such that the second step requires the first. The role labels are defined as follows, where the second case takes precedence over the first.
 1. If an element created by the first step is read by the second, $lab(e) = \langle c, r \rangle$.
 2. If an element created by the first step is deleted by the second, $lab(e) = \langle c, d \rangle$
- $OP = P$ is the set of rule names.
- $op : V \rightarrow OP$ is defined by $op(p : s) = p$

Example 2 (dependency graph) Using the example in Fig. 1 we can draw a dependency graph as shown in Fig. 4. Consider the edge between nodes *bookRoom* and *occupyRoom* labelled $\langle c, r \rangle$. That means, an object created during the first operation *bookRoom* is read by the second operation *occupyRoom*. The *cr* edge between *updateBill* and *checkout* is due to attribute *unpaid*. The first rule sets the value 0 and the second reads it.

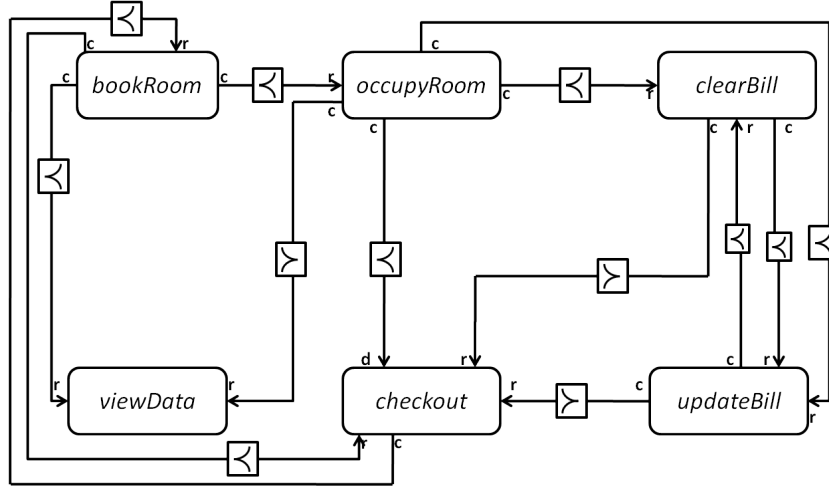


Figure 4: Dependency Graph of TAGTS representing hotel web service

When annotating edges, we denote *create* by c , *read* by r and *delete* by d . If an operation updates an attribute value, this is captured by a c labelling since the link between the attribute and its previous value is deleted and a link to a new value is created. As stated above, we drop the cr label if the same edge also has a cd label, because deletion implies read access. This allows us to restrict labels to $\{c\} \times \{r, d\}$ for the purpose of this paper (as opposed to the more general set $\{c, u, r, d\} \times \{c, u, r, d\}$ discussed in [HKM11]).

4 Test Case Generation

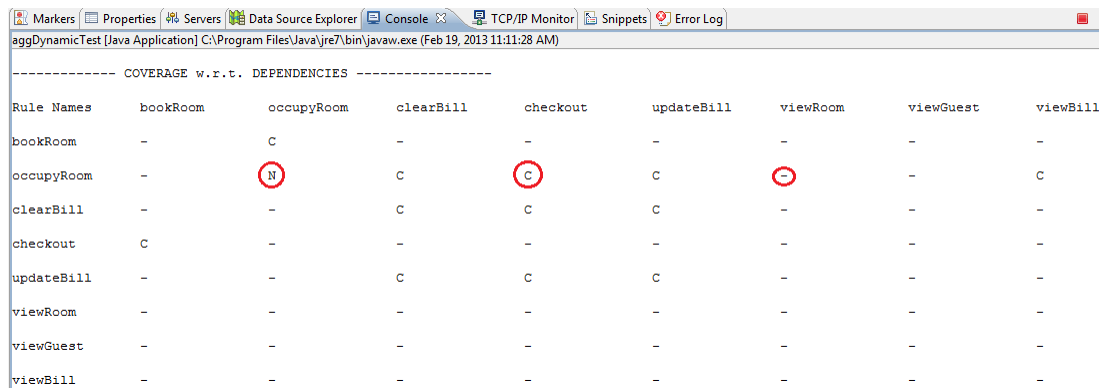
In this section, we introduce our approach to test case generation and describe the algorithm to generate sequences exercising the dependencies between rules and to record coverage. Given the dependency graph DG and initial graph G_0 , first we find out which of the rules are applicable to the start graph. Choosing one of them for the first step, we compute all paths through DG which apply each rule at most once, starting with the chosen rule. This provides us with a set S of rule sequences.

We enrich sequences in S to cater for rules with multiple dependencies, i.e., $p \prec r$ and $q \prec r$ may first lead to a sequence $\dots p; r \dots$ which is then augmented to $\dots p; q; r \dots$. Finally, we remove redundant sequences, i.e., sequences s that are contained in larger sequences as in $s_1; s; s_2$.

Example 3 (rule sequences) For the example in Figure 1, based on the start graph in Figure 2(b) our tool reports `bookRoom` as the only applicable rule. We use this rule to generate a set of possible sequences, say $\{s_1 = \text{bookRoom}; \text{occupyRoom}, s_2 = \text{bookRoom}; \text{checkout}\}$. Considering s_2 , we need `occupyRoom` to be included since `occupyRoom` \prec `checkout`. Therefore, we extend s_2 to $s'_2 = \text{bookRoom}; \text{occupyRoom}; \text{checkout}$. Then we find that s'_2 subsumes s_1 , so running s_1 would not improve coverage. Hence we drop s_1 from the set.

Next, we try to run the sequence to see if it is executable and compute the number of dependency edges it covers to find out if its execution results in additional coverage. To execute our rule sequences we have to determine the rule's matches. A match for the first rule is given by the assumption that we start with a rule that is applicable to the start graph. We then propagate matching information from rule to rule based on the potential dependencies behind the edges in the dependency graph. Applying a rule p_i in a sequence $\dots p_i; p_{i+1} \dots$ we obtain its co-match m_i^* . A potential dependency for (p_i, p_{i+1}) is based on a graph X into which both the right-hand side of p_i and the left-hand side of p_{i+1} are embedded. The overlap of the two graphs in X is used to determine a partial match for p_{i+1} from the co-match of p_i . This partial match is completed randomly, if possible. If no completion exists, another potential dependency for the pair (p_i, p_{i+1}) is chosen until, as a last resort, any match for p_{i+1} is accepted. If no such match exists, the step is dropped from the sequence and we continue with the next rule.

In order to derive the actual test cases, rule parameters have to be instantiated. This is done using the relationship between transformation sequences and their observations. In essence, each match contains an assignment of all the rule's parameters, from which we can select the input parameters to define an invocation. In an invocation sequence, some input parameters will be instantiated by output parameters of previous steps. This is realised by extracting from each (co-)match the substitution on output parameters and applying it to the rest of the invocations in the sequence. In this way, an executable sequence of invocations is generated as a result of the validation of the rule sequence. For successful sequences, coverage is determined and the resulting invocation sequence is added to the test suite.



Rule Names	bookRoom	occupyRoom	clearBill	checkout	updateBill	viewRoom	viewGuest	viewBill
bookRoom	-	C	-	-	-	-	-	-
occupyRoom	-	N	C	C	C	N	-	C
clearBill	-	-	C	C	C	-	-	-
checkout	C	-	-	-	-	-	-	-
updateBill	-	-	C	C	C	-	-	-
viewRoom	-	-	-	-	-	-	-	-
viewGuest	-	-	-	-	-	-	-	-
viewBill	-	-	-	-	-	-	-	-

Figure 5: coverage table

We iterate through the steps above as long as we observe an improvement of coverage, or until coverage is complete. At the end of each run, we report the status by displaying which sequences have failed and what was the resulting coverage. In our prototype, we also ask the user if they would like to continue for one more iteration. In each new iteration we only select those additional sequences that improve coverage. We report the resulting test cases in a text file. For example, in the case of the rule sequence $s'_2 = \text{bookRoom} ; \text{occupyRoom} ; \text{checkout}$ discussed in Example 3, the resulting test case is $\text{bookRoom}(\text{"Jim"}, 1); \text{occupyRoom}(\text{"Jim"}, 1, 100); \text{checkout}(\text{"Jim"}, 1, 100)$.

In order to compute test coverage, we execute the resulting test cases on the model using AGG as discussed in detail in [KRH12a]. We update coverage information and check if all edges are covered. The coverage table shown in Figure 5 presents the dependency information, indicating the outcome of a single run. Dependencies covered are annotated with “C” and uncovered ones are shown as “N”, while the absence of a dependency relation between two rules is represented with a “-”. We are able to cover cyclic dependencies as well, such as in our Hotel example where *bookRoom*(“Jim”, 1) is dependent upon *checkout*(“Jim”, 1, 100) and vice versa.

Example 4 (coverage of dependencies) Executing test case $t = \text{bookRoom}(\text{“Jim”}, 1); \text{occupyRoom}(\text{“Jim”}, 1, 100); \text{checkout}(\text{“Jim”}, 1, 100)$, we observe an overlap between the co-match of the first and the match of the second step exercising the cr dependency between them. In particular, the *bookingInfo* edge created by *bookRoom* is read by *occupyRoom*.

5 Evaluation

For evaluating our approach, we ask the following questions:

1. What is the time needed to generate test cases, validating them and assessing coverage?
2. How complete is the resulting test suite?
3. Does the approach scale to larger examples?

In order to answer the first question, we report on the time to generate the dependency graph and for the execution of the tests on the model to assess model-level coverage. To address the second question, we execute the resulting test cases on the implementation and use the NCover¹ tool to calculate code-based coverage for a test set that provides full dependency cover. We use a second, larger case study of a Bug Tracking service to evaluate scalability.

We report the results for the Hotel application in the first row of Table 1. The time taken by our tool to generate test cases, validate their executability and measure coverage is reported under *test generation* in the third column of Table 1. This time does not include dependency analysis, which is only conducted once when generating the dependency graph and reused until there is an evolution in the model. The time taken for dependency analysis for the Hotel example was 19.029 seconds, while 228.822 seconds were spent on the BTSys case study. In both cases, full dependency coverage was achieved after two iterations. The figures also do not include the time required for running the tests on the SUT, which is outside the scope of test case generation.

We achieve a code-based coverage of 82% for sequence points and 87% for function points (jointly comparable to statement cover in the classical terminology of white-box testing for imperative languages). A more detailed analysis reveals that the shortfall is due to additional, IDE-generated code, exception handling, and dead code, but also due to the insufficiency of dependency cover as the only model-based criterion. In fact, in [KRH12b] we also considered coverage of conflicts as well as rules that are not in conflict or dependency with any other one. Incorporating these criteria is a topic for future work.

¹ available at <http://www.ncover.com/>

S/N	Application	# of rules	# of test cases	test generation in seconds	SUT	
					Sequence points Coverage	Functions Coverage
1.	Hotel Service	9	18	3.123	82%	87%
2.	BTSys	31	135	20.202	84%	89%

Table 1: Label combinations indicating conflicts and dependencies

In order to address the third question, we have derived a Web service from the open source Bug Tracking application BTSys², replacing its GUI by a service interface. The service is implemented in C# and provides operations to manage projects and users, report faults and issues. etc. Development teams can access fault reports and update their status.

We run our test case generation for BTSys and report the results in the second row of Table 1. As might be expected, the growth of the number of test cases per number of rules is well above linear, but the time taken to generate them is 0.17 seconds per test case for the Hotel example vs. 0.15 seconds per test case for the Bug Tracker. The coverage figures are also comparable, with a marginal gain due to the fact that there was no independent rule (i.e., outside the scope of dependency cover) in BTSys.

While results are encouraging, there are some obvious weaknesses in the evaluation. With 9 and 31 rules, both systems are relatively small when compared to industry-size applications. The difficulty is that, while larger benchmark applications are available for software testing, they do not come with visual contracts. The creation of these for a large applications is a significant effort in itself. This points to another potential omission in our analysis, the cost of creating the contracts in the first place, which has to be born out by the benefit of using them for automating the relevant testing activities.

6 Related Work

Model-based software testing as a means to automate testing activities has been investigated by [NFTJ06, GHV, SAV⁺06], among others. Visual contracts have been used for testing, e.g., in [LSE05, KRH12a], and more specifically for the generation of test cases in [GMWE09, KA09, SG10]. The approach proposed in [GMWE09] uses visual contracts as system specifications and translates them to the Java modeling language (JML) to create test oracles. In order to translate logical into executable test cases, they derive concrete pre-states of the system from model-level representations and automate the checking the post-states against post conditions.

The derivation of test cases for service-oriented systems is investigated in [GHV] based on a platform metamodel for SOA and graph transformation rules describing the platform behaviour. Rule sequences are generated as counter examples using model checking based on Groove and LTSA. Our approach does not use model checking but is based on AGG for dependency analysis and validation of rule sequences generated by traversing the dependency graph.

The work in [SG10] uses visual contracts, translating them into a PDDL (Planning Domain

² available at <http://btsys.sourceforge.net/>

Definition Language), so that planning tools can be used for generating tests. Sequences of rules are computed based on an encoding of the initial system state in order to reach a state satisfying a given requirement. The authors compare their approach to the alternative (mentioned above) of using a model checker to generate the state space and find test sequences. They observe that the use of heuristics allows them to do without the generation of the full state space, so avoiding part of the state space explosion problem of model checking. Our approach is not state based at all, but focusses on the dependencies between rules instead. Starting out from a notion of coverage, a well-defined quality of the test suite is guaranteed and we evaluate how close this quality comes to traditional code-based criteria.

Many approaches to model-based testing use state machines, sequence diagrams, or the Object Constraint Language (OCL). In [OA99] software cost reduction (SCR) specifications provide state machines for test case generation against transition coverage, full predicate coverage, transition-pair coverage, and complete sequence coverage. Use cases augmented with contracts in OCL are considered in [NFTJ06] for the same purpose. Use cases are simulated and a transition system is derived, which is used for test case generation in analogy to the model checking approaches discussed earlier. The authors also analyse the implementation of their case study to identify dead code, functional code, and code to validate user and environmental inputs and show that their test cases provide complete statement cover on most categories. The generation of test cases for non-functional code from functional specifications is an interesting question for future work.

The C# extension Spec# is considered in [KA09] to support test case generation from code-based contracts. The approach uses mutation of contracts to create test cases distinguishing the mutated and original contracts and highlights cases where the implementation is found adhering to mutated instead of actually specified contracts.

Test case generation using dependency analysis is proposed in [PL11]. The authors consider information from user sessions and construct a request dependence graph for web applications. Source code analysis is conducted to construct this dependence graph where test cases are developed to test the transition relations between web pages. The approach discussed in [BL02b, BL02a] considers UML artefacts for test case generation and uses activity diagrams to represent system level dependencies between use cases. In our approach, analysis is based on dependencies extracted from graph transformation rules.

The work presented in [NMS09] discusses coverage analysis for object-oriented systems considering dependency information. The authors propose a call-based system dependence graph using the control or data dependencies between statements and calls in a method. Our approach uses dependency-based coverage at the model rather than the implementation level.

We have proposed an approach to use data dependencies as a means to derive test cases based on model-based coverage criteria proposed in [KRH12a]. Our approach differs from [GMWE09] and [SG10] in the use of dependency analysis as opposed to state-space generation or search. With respect to state machine or sequence diagram models, we focus on changes to data rather than communication behaviour. Other data-oriented approaches such as [OA99] and [NFTJ06] use textual logics such as OCL or Spec# while we start out from visual contracts that are more in line with diagrammatic modelling and more easily accessible for practitioners.

7 Conclusion and Outlook

We specify operations by visual contracts, defining pre- and post-conditions formally as rules in a graph transformation system. This provides us with an executable model where we can analyse dependencies in order to construct a dependency graph which we use to define coverage and to generate test cases. We evaluate our approach in terms of both time taken and code-based coverage achieved.

In addition to considering dependency cover, we plan to extend our approach by including coverage of conflicts, intuitively corresponding to branches in the code, as well as of isolated rules. To increase expressiveness, we plan to extend the approach to rules with negative application conditions and multi objects. Finally, after having developed a coherent set of separate components, we are investigating alternative ways of interleaving test case generation, execution, oracles and coverage analysis, for example in order to generate test cases at runtime, depending on the response of the SUT to test cases generated previously.

To address weaknesses in the evaluation, we plan to work on larger and more realistic cases studies.

Bibliography

- [AGG07] AGG - Attributed Graph Grammar System Environment. <http://www.tfs.tu-berlin.de/agg>, 2007.
- [BL02a] L. C. Briand, Y. Labiche. A UML-based approach to system testing. *Software and Systems Modeling* 1(1), 2002.
- [BL02b] L. Briand, Y. Labiche. A UML-Based Approach to System Testing, Carleton University TR SCE-01-01- Version 4. Revised June 2002.
- [GHV] L. Gönczy, R. Heckel, D. Varró. Model-Based Testing of Service Infrastructure Components. In *Testing of Software and Communicating Systems, 19th IFIP TC6/WG6.1 International Conference, TestCom 2007, 7th International Workshop, FATES 2007, Tallinn, Estonia, June 26-29, 2007, Proceedings*. Lecture Notes in Computer Science 4581, pp. 155–170. Springer.
- [GMWE09] B. Güldali, M. Mlynarski, A. Wübbeke, G. Engels. Model-Based System Testing Using Visual Contracts. In *35th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2009, Patras, Greece, August 27-29, 2009, Proceedings*. Pp. 121–124. IEEE Computer Society, 2009.
- [HHL05] J. H. Hausmann, R. Heckel, M. Lohmann. Model-Based Development of Web Services Descriptions Enabling a Precise Matching Concept. *Int. J. Web Service Res.* 2(2):67–84, 2005.
- [HKM11] R. Heckel, T. A. Khan, R. Machado. Towards Test Coverage Criteria for Visual Contracts. In *Proceedings of Graph Transformation and Visual Modeling Techniques, GTVMT 11, Electronic Communications of the EASST* 41, 2011.

- [KA09] W. Krenn, B. K. Aichernig. Test Case Generation by Contract Mutation in Spec#. *Electr. Notes Theor. Comput. Sci.* 253(2):71–86, 2009.
- [KRH12a] T. A. Khan, O. Runge, R. Heckel. Testing Against Visual Contracts: Model-based Coverage. In *6th International Conference, ICGT 2012, University of Bremen, Germany 24 - 29 September, 2012*. Lecture Notes in Computer Science 7562, pp. 155–170. Springer, LNCS, 2012.
- [KRH12b] T. A. Khan, O. Runge, R. Heckel. Visual Contracts as Test Oracle in AGG 2.0. In *Proceedings of Graph Transformation and Visual Modeling Techniques, GTVMT 12, Electronic Communications of the EASST* 47, 2012.
- [LMH07] M. Lohmann, L. Mariani, R. Heckel. A Model-Driven Approach to Discovery, Testing and Monitoring of Web Services. In *Test and Analysis of Web Services*. Pp. 173–204. Springer, 2007.
- [LSE05] M. Lohmann, S. Sauer, G. Engels. Executable Visual Contracts. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Pp. 63–70. IEEE Computer Society, Washington, DC, USA, 2005.
- [NFTJ06] C. Nebut, F. Fleurey, Y. L. Traon, J.-M. Jézéquel. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Software Eng.* 32(3):140–155, 2006.
- [NMS09] E. Najumudheen, R. Mall, D. Samanta. A Dependence Graph-Based Test Coverage Analysis Technique for Object-Oriented Programs. In *Sixth International Conference on Information Technology: New Generations, 2009. ITNG '09*. Pp. 763–768. IEEE, April 2009.
- [OA99] J. Offutt, A. Abdurazik. Generating tests from UML specifications. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*. UML'99 1723, pp. 416–429. Springer-Verlag, Berlin, Heidelberg, 1999.
- [PL11] X. Peng, L. Lu. A new approach for session-based test case generation by GA. In *3rd International Conference on Communication Software and Networks (ICCSN), 2011*. Pp. 91–96. IEEE, May 2011.
- [SAV⁺06] V. Santiago, A. S. M. do Amaral, N. L. Vijaykumar, M. d. F. Mattiello-Francisco, E. Martins, O. C. Lopes. A Practical Approach for Automated Test Case Generation using Statecharts. In *Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 02. COMPSAC '06*, pp. 183–188. IEEE Computer Society, Washington, DC, USA, 2006.
- [SG10] M. Schnelte, B. Güldali. Test Case Generation for Visual Contracts Using AI Planning. In Fähnrich and Franczyk (eds.), *Informatik 2010: Service Science - Neue Perspektiven für die Informatik, Beiträge der 40. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Band 2, 27.09. - 1.10.2010, Leipzig, GI Jahrestagung (2)*. LNI 176, pp. 369–374. GI, 2010.