Electronic Communications of the EASST
Volume 41 (2011)

Proceedings of the
Tenth International Workshop on
Graph Transformation and
Visual Modeling Techniques
(GTVMT 2011)

Towards Test Coverage Criteria for Visual Contracts

Reiko Heckel, Tamim Ahmed Khan and Rodrigo Machado

15 pages

Guest Editors: Fabio Gadducci, Leonardo Mariani
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Towards Test Coverage Criteria for Visual Contracts

Reiko Heckel[1], Tamim Ahmed Khan[2] and Rodrigo Machado[3]

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

[2] tak12@mcs.le.ac.uk
Department of Computer Sciences, Leicester University, UK

[3] rma@inf.ufrgs.br
Univ. Federal do Rio Grande do Sul, Porto Alegre, Brazil

Abstract: When testing component-based or service-oriented applications we cannot always rely on coverage criteria based on source code. Instead, we have to express our requirements for testing at the interface level. Specifying interfaces by graph transformation rules, so-called visual contracts, we define model-based coverage criteria exploiting the well-known relations of causal dependency and conflict on transformation rules.

To this end we establish an observational semantics for graph transformation systems with rule signatures formalising a notion of test execution, and define dependency graphs to provide a structure on which coverage can be analysed.

Keywords: graph transformation, services, visual contracts, observable behaviour, test coverage, causal dependencies and conflicts

1 Introduction

A user's view of a web service is through provided interfaces, which abstract from implementation details and prevent us from using traditional testing methods based on source code [CP06]. Testing of web services carries an additional overhead if it involves invoking external services, possibly remotely, causing delays, network traffic and cost [PBE$^+$07]. In order to ensure that tests carried out have sufficiently exercised the system we normally rely on coverage criteria or deploy fault-seeding to assess test sets. Coverage provides a metric for completeness with respect to a given test requirement [PJ08]. Fault-based approaches seed random faults and attempt to find them by means of the test suite to be assessed [Pfl01]. Testing web services, we cannot rely on either approach because both require access to the source code.

We propose to replace code-based by model-based coverage criteria using semantic service descriptions at the interface level. Specifying the provided operations by visual contracts, formally typed attributed graph transformation rules, we analyse their potential conflicts and dependencies [EEPT06]. We generate a dependency graph whose nodes represent rules while its edges indicate potential conflicts or dependencies between them. They also carry labels showing the nature of the relation, allowing us to record where data was defined, used, updated, or deleted. Our coverage criteria will make use of this information. Apart from formalising the basic notions and defining the criteria and their satisfaction, in this paper we illustrate their relevance by an example and discuss the limitations of the present approach.

The paper is organised as follows. Section 2 covers the relevant concepts of graph transformation while Section 3 introduces the observational semantics that form the theoretical basis for dependency graphs and model-based coverage criteria. These are then defined in Section 4 and Section 5, respectively. Section 6 is devoted to related work before we conclude the paper in Section 7.

2 Typed Attributed Graph Transformation

This section provides the basic notions on typed attributed graph transformation, following the algebraic approach [EEPT06]. A graph is a tuple (V,E,src,tgt) where V is a set of nodes (or vertices), E is a set of edges and $src,tgt : E \to V$ associate, respectively, a source and target node for each edge in E. Given graphs G_1 and G_2, a graph morphism is a pair (f_V, f_E) of total functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that source and targets of edges are preserved.

An E-graph is a graph equipped with an additional set V_D of data nodes (or values) and special sets of edges E_{EA} (edge attributes) and E_{NA} (node attributes) connecting, respectively, edges in E and nodes in V to values in V_D. An attributed graph is a tuple (EG,D) where EG is an E-graph and D is an algebra with signature $\Sigma = (S,OP)$ such that $\biguplus_{s\in S} D_s = V_D$, the set of data values available for attribution. A morphisms $f : (EG,D) \to (EG',D')$ of attributed graphs is a pair of an E-graph morphism $f_{EG} : EG \to EG'$ and a compatible algebra homomorphism $f_D : D \to D'$. Fixing as type graph an attributed graph ATG (usually over a final Σ-algebra), we define the category $\mathbf{AGraph_{ATG}}$ of ATG-typed attributed graphs [EEPT06]. Objects are pairs (G,t) of attributed graphs G with typing homomorphisms $t : G \to ATG$. Morphisms $f : G \to H$ are attributed graph morphisms compatible with the typing.

Let us denote by $X = (X_s)_{s\in S}$ a family of countable sets of variables, indexed by sorts $s \in S$, and write $x : s \in X$ for $x \in X_s$. An ATG-typed graph transformation rule (or production) over X is a span $L \xleftarrow{l} K \xrightarrow{r} R$ where l,r are monomorphisms, the algebra component of L,K,R is $T_\Sigma(X)$, the term algebra of Σ with variables in X, which the rule morphisms preserve, i.e., $l_D = r_D = id_{T_\Sigma(X)}$. That means, variables are preserved across the rule. The class of all rules over ATG with variables in X is denoted $Rules(ATG,X)$.

A typed attributed graph transformation system (TAGTS) is a tuple (ATG,P,π) where ATG is an attributed type graph, P is a set of rule names and $\pi : P \to Rules(ATG,X)$ maps rule names to ATG-typed graph transformation rules.

Example 1 (hotel service) *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 credit card details are already with the hotel management, the bill is automatically deducted when the guest announces their intention to check out. The guest can check out successfully only when the bill is paid. The type graph and rules modelling this service are shown in Fig. 1 using AGG [AGG07] notation.*

In Fig. 1, underlined attribute declarations represent key attributes used to identify nodes. For example, int roomNo *in node type RoomData is used to identify RoomData nodes. Key attribute* *values are required to be unique, not just within each graph but across the entire transformation sequence.*

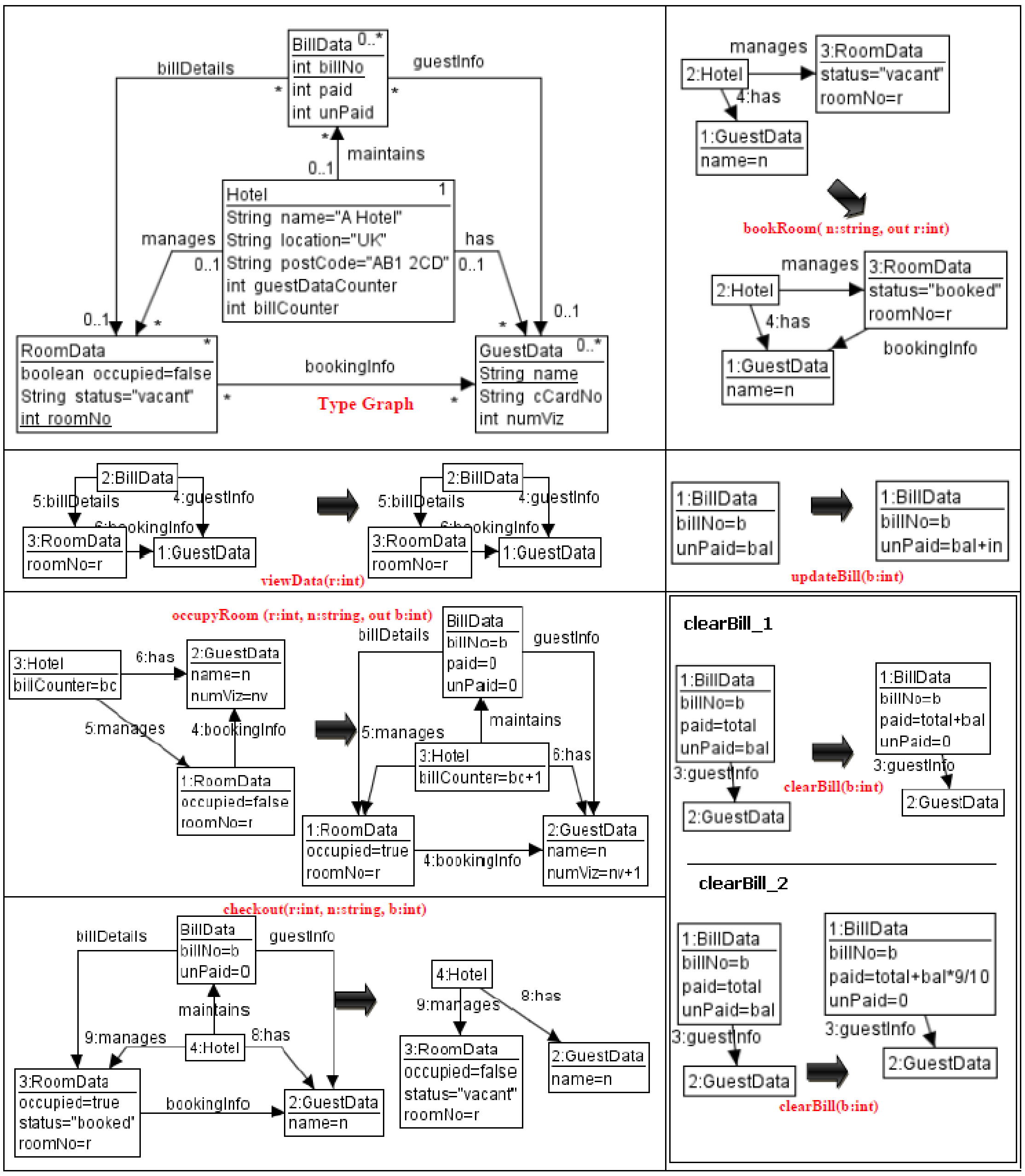


Figure 1: Type graph and rules for the hotel service

The operational semantics of rules is defined by the double-pushout construction. Given an ATG-typed graph G and graph production $L \xleftarrow{l} K \xrightarrow{r} R$ together with a match (an ATG-typed graph morphism) $m : L \to G$, a *direct derivation* $G \overset{p,m}{\Longrightarrow} H$ exists if and only if the diagram below can be constructed, where both squares are pushouts in $\mathbf{AGraph_{ATG}}$ such that G, C, H share the same algebra D and the algebra components l^*_D, r^*_D of morphisms l^*, r^* are identities on D. This ensures that data elements are preserved across derivation sequences, which allows their use as actual parameters with global namespace. We also write $G \overset{p,d}{\Longrightarrow} H$ for $d = (m = d_L, d_K, m^* = d_R)$ if we want to refer to the entire DPO diagram. A derivation is a sequence $G_0 \overset{p_1,\, m_1}{\Longrightarrow} G_1 \overset{p_2,\, m_2}{\Longrightarrow} \ldots \overset{p_n,\, m_n}{\Longrightarrow} G_n$ of direct derivations. The class of all derivations for a given *TAGTS* $\mathcal{G}$ is denoted $Der(\mathcal{G})$.

$$\begin{tikzcd}
L \arrow[d, "m=d_L"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d_K"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*=d_R"] \\
G & C \arrow[l, "l^*"] \arrow[r, "r^*"'] & H
\end{tikzcd}$$

3 Observational Semantics

In order to define a notion of observation on rule applications we provide rule names with formal parameters to be instantiated by the mach and comatch. Intuitively, these *rule signatures* provide the interface declaration of the system which is implemented by the rules. In order to distinguish different outcomes of the same operation, we allow several rules to implement the same signature.

Definition 1 (TAGTS with rule signatures) A typed attributed graph transformation system with rule signatures is a tuple $\mathcal{G} = (ATG, P, X, \pi, \sigma)$ where

- ATG is an attributed type graph with set of node attributes E_{NA};
- P is a countable set of rule names,
- X is an S-indexed family $(X_s)_{s \in S}$ of sets of variables,
- $\pi : P \longrightarrow \mathcal{P}_{fin}(Rules(ATG, X))$ assigns each rule name a finite set of rules $L \xleftarrow{l} K \xrightarrow{r} R$ over ATG, X,
- $\sigma : P \to (\{\varepsilon, out\} \times X)^*$ assigns to each rule name $p \in P$ a list of formal input and output parameters $\sigma(p) = \bar{x} = (q_1 x_1 : s_1, \ldots, q_n x_n : s_n)$ were $q_i \in \{\varepsilon, out\}$ and $x_i \in X_{s_i}$ for $1 \leq i \leq n$. We write p's *rule signature* $p(\bar{x})$ and refer to the set of all rule signatures as *signature of* $\mathcal{G}$.

Note that we distinguish normal and output parameters, the latter being indicated by *out* in front of the declaration. Normal parameters are both input and output, that is they are given in advance of the application, restrict the matching and are still valid after the rule has been applied. Output parameters are only assigned values during the matching.

Since $L \xleftarrow{l} K \xrightarrow{r} R$ is attributed over $T_\Sigma(X)$, rule parameters $x_i \in X_{s_i}$ are from the set of variables used in attribute expressions. Hence, actual parameters will not refer to nodes or edges, but to attribute values in the graph. Since the algebra part of attributed graphs is preserved, actual parameters have a global name space across transformations. Using global satisfaction of key constraints we can ensure that these attribute values are globally unique.

Example 2 (TAGTS with rule signatures) *For the system in Fig. 1, the rule signatures shown below are based on data sorts $S = \{int, boolean, string\}$ with the usual operations. Output parameters are indicated by the prefix* out *in the declaration.*

- *bookRoom(n:string, out r:int)*
- *occupyRoom(r:int, n:string, out b:int)*
- *clearBill(b:int)*
- *checkout(r:int, n:string, b:int)*
- *updateBill(b:int)*
- *viewData(r:int)*

We can associate several rules to the same signature to represent alternative actions inside the same operation, chosen by different input values and the system's internal state. In our example, the hotel provides a 10% discount to its guests on every tenth visit. In order to describe this, two rules are required: One is applicable if the current visit of a particular guest is the tenth one while the second is applicable otherwise. We associate the same rule signature with both of these rules, as shown in Fig. 1.

The purpose of signatures is to allow observations on transformations including information about rules and their matches. Below we define the label alphabet, then the observations associated with direct derivations.

Definition 2 (labels) Given a rule $p : L \leftarrow K \rightarrow R$ with signature $p(q_1x_1 : s_1, \dots, q_nx_n : s_n)$ and a Σ-algebra D, we denote by $p(D)$ the set of all rule labels $p(a_1, \dots, a_n)$ with $a_i \in D_{s_i}$. The label alphabet $L_{\mathcal{G},D}$ for a system $\mathcal{G}$ is defined as the union over all rule labels $\bigcup_{p\in P} p(D)$. If D and/or $\mathcal{G}$ are understood from the context, we write $L_{\mathcal{G}}$ or just L.

The (usually infinite) alphabet of labels L consists of all possible instances of rule signatures, replacing their formal parameters by values from the algebra D. Labels in L may be interpreted as observations of direct derivations, where the instantiation is given by the algebra component of the matches. Let L^* denote the Kleene closure over the label alphabet, providing the set of all finite sequences of labels. The following definition describes the observational semantics of TAGTS via sequences of labels produced by its derivations.

Definition 3 (observations from derivations) Let $G \overset{p,m}{\Longrightarrow} H$ be a direct derivation of a TAGTS $\mathcal{G}$ with algebra component D. The observation function obs : $Der(\mathcal{G}) \rightarrow L^*_{\mathcal{G}}$ is defined on direct derivations by $\text{obs}(G \overset{p,\, m}{\Longrightarrow} H) = p(a_1, \dots, a_n)$ if p's signature is $p(q_1x_1 : s_1, \dots, q_nx_n : s_n)$ with $a_i = m(x_i)$. The observation function freely extends to finite sequences of derivations, yielding sequences of labels.

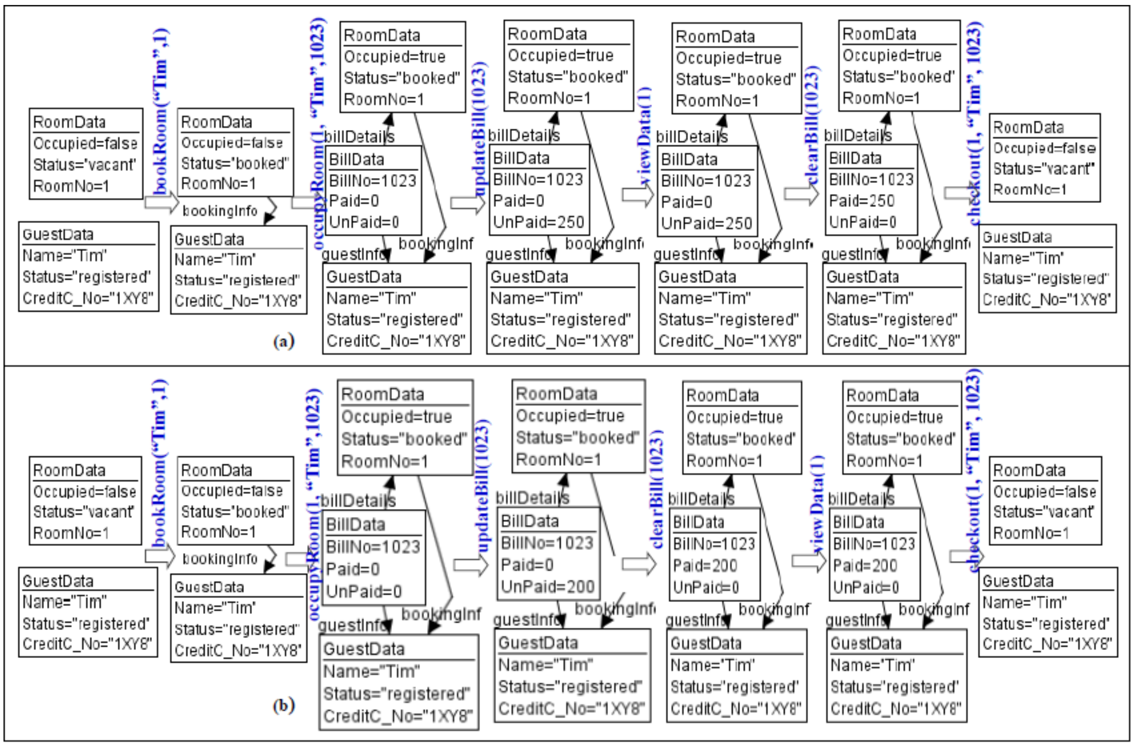


Figure 2: Two example derivations

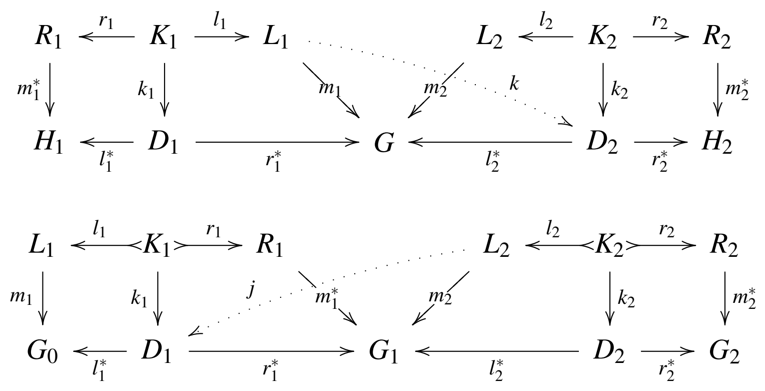
Referring to the system in Fig. 1, consider the sequences in Fig. 2(a) and Fig. 2(b). Rules names are instantiated with parameter values to represent the corresponding observation sequence. The applications of *viewData* and *clearBill* are independent and can be swapped. The graphs shown represent sample states of a hotel with only one room and one registered guest.

The following definition lifts weak dependencies and conflicts to the level of labels. The relations are essentially those of asymmetric event structures [BCM01]. The asymmetry arises from the interplay of deletion and preservation, which is specific to rewriting approaches with explicit read access to resources, such as graph transformation or contextual Petri nets.

Definition 4 (asymmetric dependencies and conflicts) Two labels l_1 and l_2 *are in direct (asymmetric) conflict*, written $l_1 \nearrow l_2$, iff there exist transformations $t_1 = (G \overset{p_1,m_1}{\Longrightarrow} H_1)$ and $t_2 = (G \overset{p_2,m_2}{\Longrightarrow} H_2)$ such that $l_i = \mathrm{obs}(t_i)$ and t_2 *disables* t_1, i.e., in the upper diagram in Fig. 3 there exist no $k : L_1 \to D_2$ such that $m_1 = l_2^* \circ k$.

Two labels l_1 and l_2 *are in (asymmetric) dependency*, $l_1 \prec l_2$, iff there exist transformations $t_1 = (G_0 \overset{p_1,m_1}{\Longrightarrow} G_1)$ and $t_2 = (G_1 \overset{p_2,m_2}{\Longrightarrow} G_2)$ such that $l_i = \mathrm{obs}(t_i)$ and t_2 *requires* t_1, i.e., in the lower diagram in Fig. 3 there exist no $j : L_2 \to D_1$ such that $m_2 = r_1^* \circ j$. Labels l_1 and l_2 are independent, $l_1 \mid l_2$, iff they are unrelated by $\nearrow$ and $\prec$.

Example 3 (asymmetric conflicts and dependencies of $\mathscr{G}$) *Using the rule signatures in Exam-*



$$\begin{tikzcd}
R_1 \arrow[d, "m_1^*"'] & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"'] & L_1 \arrow[dr, "m_1"] \arrow[drrr, dotted, "k"] & & L_2 \arrow[dl, "m_2"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] \arrow[d, "k_2"] & R_2 \arrow[d, "m_2^*"] \\
H_1 & D_1 \arrow[l, "l_1^*"] \arrow[rr, "r_1^*"'] & & G & & D_2 \arrow[ll, "l_2^*"] \arrow[r, "r_2^*"'] & H_2
\end{tikzcd}$$

$$\begin{tikzcd}
L_1 \arrow[d, "m_1"'] & K_1 \arrow[l, tail, "l_1"'] \arrow[r, tail, "r_1"] \arrow[d, "k_1"'] & R_1 \arrow[dr, "m_1^*"] & & L_2 \arrow[dl, "m_2"'] \arrow[dlll, dotted, "j"'] & K_2 \arrow[l, tail, "l_2"'] \arrow[r, tail, "r_2"] \arrow[d, "k_2"] & R_2 \arrow[d, "m_2^*"] \\
G_0 & D_1 \arrow[l, "l_1^*"] \arrow[rr, "r_1^*"'] & & G_1 & & D_2 \arrow[ll, "l_2^*"] \arrow[r, "r_2^*"'] & G_2
\end{tikzcd}$$

Figure 3: Asymmetric conflicts and dependencies

ple 2 we obtain labels by instantiating formal parameters by possible data values. As the set of all labels is usually infinite due to infinite data types, in this example we limit ourselves to a small subset sufficient to label the transformations in Fig. 2. For example, $clearBill(bill\_no : int)$ *is instantiated by* $clearBill(1023)$*, replacing the variable* $bill\_no : int$ *by the value* 1023*.*

Let us analyse more closely the weak conflicts and dependencies in the derivations of Fig. 2. We have represented conflicts and dependencies between these labels in Table 1. For example we find a dependency

$$bookRoom(1,\ Tim) \prec occupyRoom(1,\ Tim,\ 1023).$$

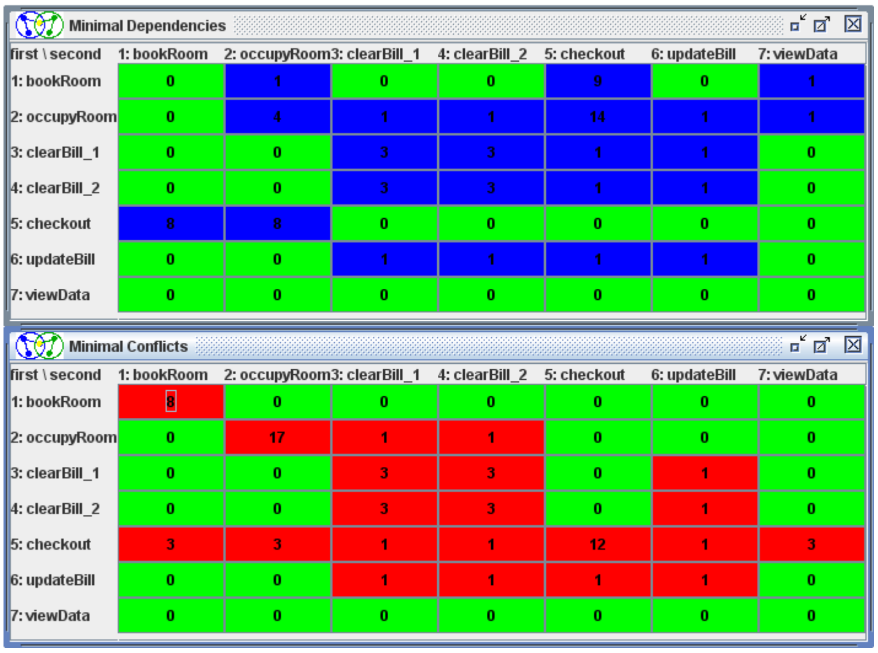
Notice the relation between the parameters for room number and client name, which determines the overlap of the transformations denoted by these labels. Similarly, there exists a conflict

$$updateBill(1023) \nearrow checkout(1, Tim, 1023),$$

i.e., $updateBill$ *reads the* $BillData$ *object, changing the* $unpaid$ *amount, while* $checkout$ *deletes the object.*

First/Second ($\downarrow$) **/** ($\rightarrow$)	$bookRoom$ $(1,"Tim")$	$occupyRoom$ $(1,"Tim",1023)$	$clearBill$ (1023)	$checkout$ $(1,"Tim",1023)$	$updateBill$ (1023)	$viewData$ (1)
$bookRoom(1,"Tim")$	$\nearrow$	$\prec$		$\prec$		$\prec$
$occupyRoom(1,"Tim",1023)$		$\nearrow \prec$	$\prec$	$\prec$	$\prec$	$\prec$
$clearBill(1023)$			$\nearrow \prec$	$\nearrow \prec$	$\nearrow \prec$	$\vert$
$checkout(1,"Tim",1023)$	$\prec$	$\prec$		$\nearrow$	$\nearrow$	
$updateBill(1023)$			$\nearrow \prec$	$\nearrow \prec$	$\nearrow \prec$	$\vert$
$viewData(1)$			$\vert$	$\nearrow$	$\vert$	$\vert$

Table 1: Conflicts $\nearrow$ and dependencies $\prec$ between labels



Minimal Dependencies

first \ second	1: bookRoom	2: occupyRoom	3: clearBill_1	4: clearBill_2	5: checkout	6: updateBill	7: viewData
1: bookRoom	0	1	0	0	9	0	1
2: occupyRoom	0	4	1	1	14	1	1
3: clearBill_1	0	0	3	3	1	1	0
4: clearBill_2	0	0	3	3	1	1	0
5: checkout	8	8	0	0	0	0	0
6: updateBill	0	0	1	1	1	1	0
7: viewData	0	0	0	0	0	0	0

Minimal Conflicts

first \ second	1: bookRoom	2: occupyRoom	3: clearBill_1	4: clearBill_2	5: checkout	6: updateBill	7: viewData
1: bookRoom	8	0	0	0	0	0	0
2: occupyRoom	0	17	1	1	0	0	0
3: clearBill_1	0	0	3	3	0	1	0
4: clearBill_2	0	0	3	3	0	1	0
5: checkout	3	3	1	1	12	1	3
6: updateBill	0	0	1	1	1	1	0
7: viewData	0	0	0	0	0	0	0

Figure 4: Critical pairs and dependencies

4 Dependency Graphs

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 conflicts and dependencies allowing us to study coverage criteria at the interface level.

Definition 5 (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.
- $op : V \to OP$ maps vertices to operation names.
- $lab : E \to \{c,u,r,d\} \times \{\prec, \nearrow\} \times \{c,u,r,d\}$ is a labelling function distinguishing source and target types *create, update, read, delete* and dependency types $\prec, \nearrow$.

We use the visual contracts specifying the interface to extract a dependency graph, where rules are represented by nodes labeled by operation names while edges represent dependencies and conflicts between them. Edge labels tell us whether an edge represents a dependency ($\prec$) or a conflict ($\nearrow$) and what roles are played by the source and target nodes.

Definition 6 (dependency graph of TAGTS with rule signatures) Given a TAGTS with rule signatures $\mathcal{G} = (ATG, P, X, \pi, \sigma)$, 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 rule spans 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 there are steps $G \overset{p_1:s_1,m_1}{\Longrightarrow} H_1 \overset{p_2:s_2,m_2}{\Longrightarrow} H_2$ such that the second step requires the first. The role labels are defined as follows.
    1. If an element created by the first step is deleted or read by the second, $lab(e) = \langle c, \prec, d \rangle$ or $lab(e) = \langle c, \prec, r \rangle$, respectively. If both apply, label d takes precedence over r.
    2. If an attribute updated by the first step is updated or read by the second, $lab(e) = \langle u, \prec, u \rangle$ or $lab(e) = \langle u, \prec, r \rangle$, respectively. If both apply, label u takes precedence over r.
    3. If an object created by the first step has an attribute updated by the second, $lab(e) = \langle c, \prec, u \rangle$
  - $e = (p_1 : s_1, p_2 : s_2) \in E$ if there are steps $H_1 \overset{p_1:r_1,m_1}{\Longleftarrow} G \overset{p_2:r_2,m_2}{\Longrightarrow} H_2$ such that the second disables the first. The role labels are defined as follows.
    1. If an element deleted or read by the first step is also deleted by the second, $lab(e) = \langle d, \nearrow, d \rangle$ or $lab(e) = \langle r, \nearrow, d \rangle$, respectively. If both apply, label d takes precedence over r.
    2. If an attribute updated or read by the first step is updated by the second, $lab(e) = \langle u, \prec, u \rangle$ or $lab(e) = \langle r, \prec, u \rangle$, respectively. If both apply, label u takes precedence over r.
    3. If an object's attribute is updated by the first step and deleted by the second, $lab(e) = \langle u, \prec, d \rangle$
- $OP = P$ is the set of rule names.
- $op : V \to OP$ is defined by $op(p : s) = p$

Example 4 (dependency graph) *Using the example in Fig. 1 we can draw a dependency graph as shown in Fig. 5. Consider an edge between nodes $bookRoom(\ldots)$ and $occupyRoom(\ldots)$ where the labeling is $\langle c, \prec, r \rangle$. That means, an object created during the first operation $bookRoom(\ldots)$ is read by the second operation $occupyRoom(\ldots)$ with $\prec$ representing the dependency relation. Similarly, consider an edge between $clearBill(\ldots)$ and $checkout(\ldots)$ where the labelling is $\langle r, \nearrow, d \rangle$. It means, $clearBill(\ldots)$ reads an object which is deleted by $checkout(\ldots)$. An examination of the rules reveals that $clearBill(\ldots)$ operates on a $BillData$ object which is deleted by $checkout(\ldots)$.*

5 Coverage Criteria

Dataflow graphs, in code-based approaches, are generated considering the control-flow of the system with additional annotations on the nodes. These annotations are used to mark the places

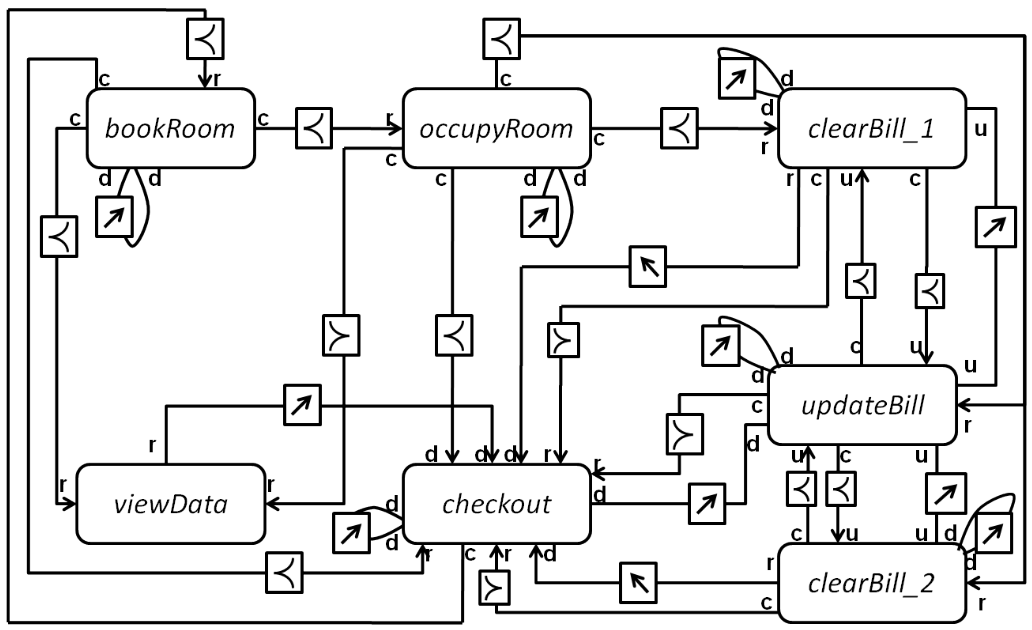


Fig. 5: dependency graph of *TAGTS* representing hotel web service

where data is defined and used in a program [PJ08], [CPRZ89], [TSP01]. The locations where a variable is defined are annotated by *def*, use is indicated by *use*, and deallocated by *kill*. Paths are identified through the system such that they exercise particular coverage criteria. For example, *def*−*use* coverage requires to find test cases such that all edges in the dataflow graph from nodes annotated with *def* to nodes annotated with *use* are exercised at least once.

Our dependency graph carries information about dependencies between operations at the interface level. We have annotated sources and targets of edges with *c* (create), *r* (read), *u* (update) an *d* (delete), analogousy to *def*, *use* and *kill* annotations for traditional versions of dataflow graphs. Using these labels on edges rather than nodes, we can focus on the type of access that gives rise to the particular dependency or conflict represented by the edge. The possibilities are summarised in Table. 2.

Label Combination	**Conflict**	**Dependency**
cr	×	√
cd	×	√
cu	×	√
uu	√	√
rd	√	×
dd	√	×
ud	√	×

Table 2: Label combinations indicating conflicts and dependencies

Using the DG, we can devise different coverage criteria such as cr, cd, ud, etc. The question is, which of these pairs to include into our criteria. If we demand all cr (create-read), cu (create-update), cd (create-delete), and uu (update-update) edges in $DG(\mathcal{G})$, we exercise all dependencies based on data being defined and used subsequently. If we add ud (update-delete) and rd (read-delete), we also cover situations of asymmetric conflict. The case of dd (delete-delete) represents an anomaly because we should not be able to observe a sequence where two steps are in symmetric conflict. It might still be interesting to create a test case to check that it is non-executable, but to guarantee the conflict we have to enforce a certain overlap of parameters, requiring a more detailed version of the dependency graph.

In order to see if a set of test cases T provides the required coverage, we record all the nodes and edges that T is exercising.

Definition 7 (sub-graph covered by test set) Given a TAGTS $\mathcal{G} = (ATG, P, X, \pi, \sigma)$ with dependency graph $DG(\mathcal{G}) = \langle G, OP, op, lab \rangle$ and let T be a set of derivations in $\mathcal{G}$. The graph $cov(DG, T) = \langle G_T, OP_T, op_T, lab_T \rangle$ is the subgraph of $DG(\mathcal{G})$ with $G_T = \langle V_T, E_T, src, tgt \rangle$ such that:

- $v \in V_T$ iff for $p = op(v)$ and a derivation $s \in T$ there is a step in s labelled by $p(\bar{a})$.
- $e \in E_T$ iff for $op(src(e)) = p$ and $op(tgt(e)) = q$ and a derivation $s \in T$ there are steps in s labelled $p(\bar{a}), q(\bar{b})$ in asymmetric conflict or dependency as specified by $lab(e)$.
- $OP_T = OP\ |_{V_T}$
- $op_T = op\ |_{V_T}$
- $lab_T = lab\ |_{E_T}$

A coverage criterion C is defined by any subgraph of $DG(\mathcal{G})$. Test set T provides the coverage required by C if $C \subseteq cov(DG, T)$.

Example 5 (coverage of test cases) *Consider criterion $cr + cd$ and let T be the set of test cases shown in Table. 3.*

test cases set-I
$bookRoom(1, \text{“}J\text{”}); occupyRoom(1, \text{“}J\text{”}, 3)$
$occupyRoom(1, \text{“}J\text{”}, 3); viewData(1)$
$occupyRoom(1, \text{“}J\text{”}, 3); checkout(1, \text{“}J\text{”}, 3)$
$updateBill(3); clearBill(3)$
$clearBill(3); updateBill(3)$

Table 3: Test cases providing node coverage

In the graph in Fig. 6, dotted red and the blue lines of the show the edges not covered by the test cases in Table. 3, while the solid black rest of the graph is covered by T. This includes all the nodes, but we need to add test cases in order to get the required coverage $cr + cd$. In order to achieve this we add the test cases in Table 4 and analyse the resulting coverage.

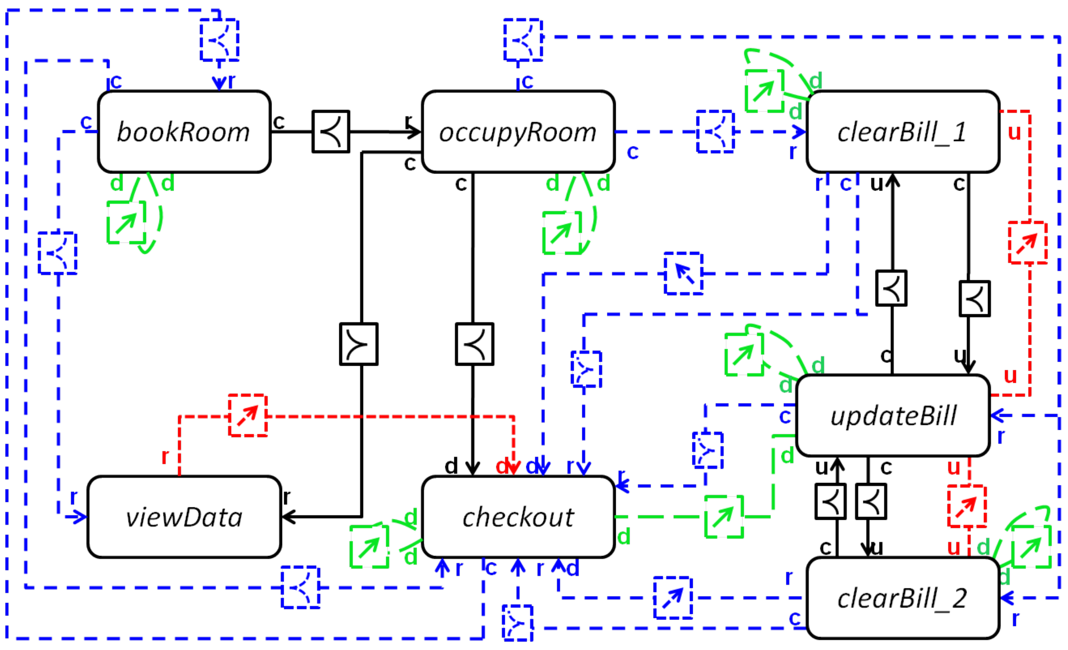


Figure 6: $Cov(DG,\ T)$

test cases set-II
$bookRoom(1, \text{“}J\text{”}); occupyRoom(1, \text{“}J\text{”}, 3); viewData(1)$
$bookRoom(1, \text{“}J\text{”}); occupyRoom(1, \text{“}J\text{”}, 3); checkout(1, \text{“}J\text{”}, 3)$
$checkout(1, \text{“}J\text{”}, 3); bookRoom(1, \text{“}J\text{”})$
$occupyRoom(1, \text{“}J\text{”}, 3); clearBill(3);$
$updateBill(3); clearBill(3); checkout(1, \text{“}J\text{”}, 3)$

Table 4: Additional test cases to cover $cr + cd$

The first two test cases in Table 4 are required to cover the cr edges between $bookRoom(\dots)$ and $viewData(\dots)$ and between $bookRoom(\dots)$ and $checkout(\dots)$ and the inclusion of $occupyRoom(1, \text{“}J\text{”}, 3)$ is required. The third and forth test cases are required to cover the edges between $checkout(\dots)$ and $bookRoom(\dots)$ and between $occupyRoom(\dots)$ and $clearBill(\dots)$ where the fifth test case exercises edges between $updateBill(\dots)$, $clearBill(\dots)$ and $checkout(\dots)$. Notice that the dotted red edge between $viewData(\dots)$ and $checkout(\dots)$ is still not covered by any of the test cases, just as the edges shown in dotted green. We need stronger criteria to include test cases covering these edges, too.

6 Related Work

For similar motivations as our own, several approaches have developed coverage criteria based on control flow graphs and state machines. In [LCG08] they have been derived from semantic service descriptions in RDF, [SP06] uses WSDL-S while [ROL+07] constructs a global flow graph from information for each party in a collaboration and defines conditions for regression testing.

Approaches to test services based on dataflow include [BBMP08, SP06, HXX+08]. For testing Web services compositions [BBMP08] uses BPEL specifications where dependencies between calls are established by means of input and output relationship. Paths through such a graph are extracted and criteria are defined to cover the combination of events in these test paths. In a later approach, the BPEL process is visualised as a directed graph where nodes represent activities and edges represent (control and data) flows. Annotations for def and *use* are used at inputs and outputs of service calls and coverage based is defined. BPEL is also considered in [HXX+08] and a dependency analysis for variables acquired from WSDL is used to arrive at possible paths through the process. Testing approaches for object-oriented systems are introduced in [CLN05, BLL10].

We have made use of web service specifications by means of visual contracts for deriving a dependency graph to define coverage. Dependencies and conflicts extracted by critical pair analysis provide a simple representation of the system at the interface level, abstracting from detailed control flow specifications.

7 Conclusion

In this paper we have explored the use of graph transformation systems specifying service interfaces for the derivation of model-based coverage criteria. While we were able to demonstrate the potential usefulness of the approach, two areas are left to be explored.

- We would like to extend our dependency graphs with data flow information to describe in more detail the conflicts and dependencies encountered. This will allow to define sequences of labels as forbidden traces, such that tests can not only check the existence of required behaviours, but also the absence of forbidden ones.

- An approach to test coverage based on executable models may be naturally combined with the use of these models as oracles, to define when a test case conforms to the specification. This is conceptually simple, but requires the integration of a testing tool with a graph transformation engine executing the specification.

Bibliography

[AGG07] AGG. AGG - Attributed Graph Grammar System Environment. http://tfs.cs.tu-berlin.de/agg, 2007.

[BBMP08] C. Bartolini, A. Bertolino, E. Marchetti, I. Parissis. Architecting Dependable Systems V. In Lemos et al. (eds.). Chapter Data Flow-Based Validation of Web Services Compositions: Perspectives and Examples, pp. 298–325. Springer-Verlag, Berlin, Heidelberg, 2008.

[BCM01] P. Baldan, A. Corradini, U. Montanari. Contextual Petri Nets, Asymmetric Event Structures, and Processes. *Information and Computation* 171(1):1 – 49, 2001.

[BLL10] L. Briand, Y. Labiche, Q. Lin. Improving the Coverage Criteria of UML State Machines Using Data Flow Analysis. *Software Testing, Validation, and Reliability (Wiley)* 20(3), 2010.

[CLN05] Y. Chen, S. Liu, F. Nagoya. An Approach to Integration Testing Based on Data Flow Specifications. In Liu and Araki (eds.), *Theoretical Aspects of Computing - ICTAC 2004*. Lecture Notes in Computer Science 3407, pp. 235–249. Springer Berlin / Heidelberg, 2005.

[CP06] G. Canfora, M. D. Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional* 8:10–17, 2006.

[CPRZ89] L. A. Clarke, A. Podgurski, D. J. Richardson, S. J. Zeil. A Formal Evaluation of Data Flow Path Selection Criteria. *IEEE Trans. Softw. Eng.* 15(11):1318–1332, 1989.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

[HXX$^+$08] J. Hou, B. Xu, L. Xu, D. Wang, J. Xu. A testing method for Web services composition based on data-flow. *Wuhan University Journal of Natural Sciences* 13:455–460, 2008.

[LCG08] L. Li, W. Chou, W. Guo. Control Flow Analysis and Coverage Driven Testing for Web Services. In *Web Services, 2008. ICWS '08. IEEE International Conference on*. Pp. 473 –480. 2008.

[PBE$^+$07] M. Penta, M. Bruno, G. Esposito, V. Mazza, G. Canfora. Web Services Regression Testing. Pp. 205–234. 2007.
doi:10.1007/978-3-540-72912-9_8
http://dx.doi.org/10.1007/978-3-540-72912-9_8

[Pfl01] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[PJ08] A. Paul, O. Jeff. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[ROL$^+$07] M. Ruth, S. Oh, A. Loup, B. Horton, O. Gallet, M. Mata, S. Tu. Towards Automatic Regression Test Selection for Web Services. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*. Pp. 729–736. IEEE Computer Society, Washington, DC, USA, 2007.

[SP06] A. Sinha, A. Paradkar. Model-based functional conformance testing of web services operating on persistent data. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*. TAV-WEB '06, pp. 17–22. ACM, New York, NY, USA, 2006.

[TSP01] B.-Y. Tsai, S. Stobart, N. Parrington. Employing data flow testing on object-oriented classes. *Software, IEE Proceedings* - 148(2):56 –64, Apr. 2001.