Use Case Evolution Analysis based on Graph Transformation with Negative Application Conditions [1]

Leila Ribeiro, Lucio Duarte, Rodrigo Machado, Andrei Costa, Érika Cota, Jonas Bezerra

PPGC/INF - Federal University of Rio Grande do Sul (UFRGS) – Porto Alegre, Brazil
Email: {leila, lmduarte, rma, acosta, erika, jsbezerra}@inf.ufrgs.br

Abstract

Use Case (UC) quality impacts the overall quality and defect rate of a system, as they specify the expected behavior of an implementation. In a previous work, we have defined an approach for a step-by-step translation from UCs written in natural language to a formal description in terms of Graph Transformation (GT), where each step of the UC was translated to a transformation rule. This UC formalisation enables the detection of several specification problems even before an actual implementation is produced, thus reducing development costs. In this paper, we extend our approach to handle UC evolution by defining *evolution rules*, which are described as higher-order rules, simultaneously changing the behaviour of a set of transformation rules. We also support the use of *negative application conditions (NAC)* associated both to the transformation and evolution rules. Analysis of the interplay between the evolution rules and the rules describing UC steps shows the effects of an evolution and serves to identify potential impacts, even before the changes are actually carried out. Besides defining the theoretical foundations of UC evolution with NACs, we have implemented the evolution analysis technique in the Verigraph tool and used it to verify impacts in 3 different case studies. The results demonstrate the applicability and usefulness of our approach to help developers in the evolution process based on UCs.

Keywords: Use Case, Graph Transformation, Software evolution, Higher-order transformations, NACs

1. Introduction

Use Cases (UC) descriptions are typically informally documented, using natural language [1]. Being informal descriptions, UCs might be ambiguous and imprecise, resulting in a number of problems that can propagate to later development phases and jeopardize the overall system quality [2]. Effective manual verification of these artifacts may be expensive, which makes it desirable to have a way of automating this task. UC correctness is specially vulnerable during system evolution. As a system evolves, specification documents must change accordingly, describing new functionalities and/or including changes in existing features. Depending on the number of UCs and their complexity, identifying the impacts of an evolution can be extremely hard to do by human inspection. Ensuring that the UCs have been correctly updated to reflect the necessary modifications can be a

costly and time-consuming task. Hence, this process should be supported by techniques that could guide the developer, indicating how to apply each of the required changes to the UCs and how these changes affect local (intra-UC) and global (inter-UC) behaviour. Thus, the translation of UCs to a formal model can help improve UC reliability and correctness by enabling a variety of analyses using existing tool support. This allows the detection of several specification problems before an actual implementation is produced, thus reducing development costs.

In [3], we presented an approach for the translation of UCs to a formal model called *Graph Transformation (GT)*. Elements of a system are described as nodes of a graph and edges between them represent their relationships (e.g., data dependency). System behavior is specified by a set of transformation rules, creating/preserving/deleting nodes and/or edges. Each rule specifies application pre-conditions (i.e., the necessary set of nodes and their specific connections through edges that must exist as a subgraph of the current system graph) and post-conditions (i.e., the resulting subgraph). Besides being a *visual* and *intuitive notation*, GT is *data-driven*, allowing a very abstract description of a system, without imposing any control not strictly enforced by data. Moreover, GT analysis is supported by tools capable of pinpointing possible problems, which are easily traced back to the original UC description.

The proposed translation method, although not yet fully supported by tools, can be followed by developers with very basic knowledge of the formalism, as we provide step-by-step guidelines. The subsequent analyses are automatically performed by available tools (AGG [4] and Verigraph[2]). Types and severity of different classes of problems that can be detected have been defined, as well as hints on the causes and possible solutions. The UCs can then be modified until the analyses show that they present the desired behavior. As a result, the GT-based UC verification approach increases the correctness of the UCs, while keeping the informality and flexibility of their description in natural language. UC analysis based on GTs has been evaluated on real UCs and revealed several problems not detected during manual inspection [3]. From these problems, 75% were classified as actual errors by the developers who had created the UCs. Hence, the approach has proved to be useful and effective for UC analysis and enhancement.

In this paper, *we extend our approach to support the description and analysis of system evolution*. By "evolution" we mean any modification in the system, which may or may not affect the system behaviour. We expand the formal framework to define evolution in terms of higher-order GTs, introduced in [5]. The main idea is to describe evolution as a second-order rule, called *evolution rules*, which are rules that change other (first-order) rules. Because evolution is also represented as a GT, the developer works within the same formalism used to specify the system. We present the theoretical foundations for reliable UC change in a system evolution scenario, offering not only a precise description of evolution, but also the possibility of analyzing evolution effects even before modifying the original rules, thus helping the decision-making process. The analysis technique is based on critical pairs, and was implemented in the *Verigraph* tool, which is also able to apply the changes defined by an evolution rule. We also extend the framework of higher-order graph transformation to support evolution of rules with *negative application conditions (NACs)*, since NACs are required in the GT-based representation of use cases. Moreover,

---

[2]`http://github.com/verites/verigraph`

we provide guidelines to interpret the analysis results and demonstrate the application of our evolution framework in 3 case studies.

This paper contains the following structure: Section 2 presents the background on GTs, the UCs formalization strategy proposed in [3], and critical pair analysis in the context of UCs; Section 3 extends the notion of second-order transformation towards rules with negative application conditions (NACs). Section 4 presents our approach to describe and analyze UC evolution; in Sect. 5 the application of the proposed approach on 3 case studies is presented; Section 6 describes some related work; and Section 7 presents conclusions and possible future work.

2. Background

2.1. Graph Transformations

In this section, we present the main concepts of Graph Transformations (GT). The algebraic approaches for graph transformation use categorical operations in order to perform the transformations defined by the rules [6, 7]. The approach we follow uses two *pushouts* as gluing operations, therefore it is called Double-PushOut approach (DPO). Examples are presented in the next section.

Graphs are structures that consist of a set of nodes and a set of edges. Each edge connects two nodes of the graph, one representing a source and another representing a target. A **(total) homomorphism** between graphs is a mapping of nodes and edges that is compatible with sources and targets of edges. Intuitively, a homomorphism from a graph *G1* to a graph *G2* means that all items (nodes and edges) of *G1* can be found in *G2* (but distinct nodes/edges of *G1* are not necessarily distinct in *G2*).

Definition 1 (graph, graph morphism). *A **graph** $G = (V, E, s, t)$ consists of a set V of nodes, a set E of edges, and two functions, $s, t : E \to V$, the source and target functions. Given two graphs, $G_1 = (V_1, E_1, s_1, t_1)$ and $G_2 = (V_2, E_2, s_2, t_2)$, a **graph morphism** $f : G_1 \to G_2$ is composed by two total functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$ such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. A graph morphism is injective/surjective/iso if both components are injective/surjective/iso. The category of graphs and graph morphisms is called* **Graph**.

For practical applications, it is very convenient to distinguish different types of vertices and edges in a graph. In the DPO approach, this can be achieved by the notion of typed graph. Let TG be a graph that represents all possible (graphical) types that are needed to describe a system, a homomorphism h from any graph G to TG associates a (graphical) type to each item of G. The triple $\langle G, h, TG \rangle$ is called *typed graph*, where TG is the *type graph* (nodes of TG denote all possible types of nodes of a system, edges of TG denote possible relationships between the nodes).

Definition 2 (typed graph, typed graph morphism). *A **typed graph** is a triple $(G1, type_{G1}, TG)$, denoted by $G1^{TG}$, where $G1$ and TG are graphs and $type : G1 \to TG$ is a graph morphism. Given two typed graphs over the same type graph, G_1^{TG} and G_2^{TG}, and their respective typing morphisms $type_{G1} : G_1 \to TG$ and $type_{G2} : G_2 \to TG$, a **typed graph morphism** is a pair (f, id_{TG}), where $f : G_1 \to G_2$ is a graph morphism and id_{TG} is the identity morphism of TG, such that: $type_{G2} \circ f = type_{G1}$. A typed graph morphism is injective/surjective/iso if f is injective/surjective/iso.*

$$G_1 \xrightarrow{\;\;f\;\;} G_2$$

$$\underset{type_1}{\searrow} \quad = \quad \underset{type_2}{\swarrow}$$

$$TG$$

The category of graphs typed over TG as objects and typed graph morphisms as morphisms is called $\mathbf{Graph}_{TG}$. This category is the comma category $(\mathbf{Graph} \downarrow \mathbf{TG})$.

A **Graph Rule** describes how a system may change. It consists of: a **left-hand side (LHS)**, denoting items that must be present for this rule to be applied; a **preserved part (also called gluing part K)**, describing items that will be preserved when the rule is applied; a **right-hand side (RHS)**, describing items that will be present after the application of the rule; **mappings from K to LHS and RHS**, which mark in LHS and RHS the items that will be preserved by the application of the rule; and a **negative application condition (NAC)**, that is actually a collection of conditions representing situations that prevent the rule from being applied (NACs are described by mappings from LHS to the graph representing the forbidden context). The mappings from K to LHS and RHS (as well as from LHS to the NACs) must be compatible with the structure of the graphs (graph homomorphisms). Items that are in LHS and are not in K are **deleted**, whereas items that are in the RHS and are not in K are **created**. We assume that rules do not merge items (rules are injective). The structure $LHS \leftarrow K \rightarrow RHS$ of a rule is called **rule span**.

Definition 3 (rule, NAC, rule with NACs, NAC satisfiability)**.** *A (typed graph) rule p consists of two typed graph morphisms l and r with the same typed graph as source, $p = L \xleftarrow{l} K \xrightarrow{r} R$, where l and r are monomorphisms (in the category $\mathbf{Graph}_{TG}$, monomorphisms are injective mappings).*

*Given a rule $p = L \xleftarrow{l} K \xrightarrow{r} R$, a **negative application condition** $nac(n)$ for p is an arbitrary typed graph morphism $n : L \rightarrow N$. A NAC $n : L \rightarrow N$ is satisfied with respect to a match $m : L \rightarrow G$ if and only if $\nexists q : N \rightarrow G$ such that q is injective and $q \circ n = g$.*

A rule with NACs (p, nac_p) is composed by a rule p and a set of NACs for p (nac_p).

A match $m : L \rightarrow G$ satisfies nac_p if and only if it satisfies all single NACs in nac_p, we denote as $nac_p \vDash m\,(p, G)$.

$$N_i$$
$$q \,\nearrow \qquad \uparrow n_i$$
$$G \xleftarrow{\;\;m\;\;} L$$

A **GT System** consists of a type graph, specifying the (graphical) types of the system, and a set of rules over this type graph that define the system behavior.

Definition 4 ((first order) graph transformation system)**.** *A **(first order) Graph Transformation System** consists of a set of (typed graph) rules with NACs and a typed graph, called initial or start graph. The rules of a first-order graph transformation system are called first-order rules.*

The *application of a rule* (NAC, rs) where $rs : L \leftarrow K \rightarrow R$ to a graph G is possible if (i) an image of L is found in G (that is, there is a total typed-graph morphism from

the LHS of rs to G); (ii) for each $nac : L \to X$ in NAC it is not possible to find an image for X in G (i.e., the forbidden items are not in G); and (iii) the **gluing condition** is satisfied, this condition assures that the result of removing from G all items that should be deleted by the rule yields a unique result that is a well-formed graph (for example, it is not possible to delete a node from G if there are edges connected to it that are not also marked for deletion in L). The result of a rule application deletes from G all items that are not in the gluing graph K (step (1) below) and adds the ones created in R (step 2 below). Formally, these steps are constructed by pushouts in a suitable category.

$$X \xleftarrow{\ n_i\ } L \xleftarrow{\ l\ } K \xrightarrow{\ r\ } R$$

$$
\begin{array}{ccccccc}
X & \xleftarrow{n_i} & L & \xleftarrow{\ l\ } & K & \xrightarrow{\ r\ } & R \\
{\scriptstyle e}\Big\downarrow & & {\scriptstyle m}\Big\downarrow & (1) & \Big\downarrow & (2) & \Big\downarrow \\
 & \dashrightarrow & G & \longleftarrow & D & \longrightarrow & H
\end{array}
$$

Definition 5 (match, typed graph transformation)**.** *Consider a rule $p = L \xleftarrow{l} K \xrightarrow{r} R$ and a typed graph G, as in the diagram below. A **match** is an arbitrary typed graph morphism from L to G.*

*A match m satisfies the **gluing condition** iff both conditions below are satisfied:*

(dangling condition) *All edges of G that are connected to nodes that are in the image of m are also in the image of m;*

(identification condition) *If an element e of L is deleted (not in the image of l), no other element of L may be mapped to $m(e)$ in G.*

A match $m : L \to G$ satisfies nac_p if and only if it satisfies all single NACs in nac_p, we denote as $nac_p \models m\,(p, G)$.

$$
\begin{array}{ccc}
 & & N_i \\
 & {\scriptstyle q}\nearrow & \Big\uparrow {\scriptstyle n_i} \\
G & \xleftarrow{\ m\ } & L
\end{array}
$$

*Given a rule p and a match m, a **(typed) graph transformation** $G \overset{p,m}{\Longrightarrow} H$ from G to H is defined as the diagram below, where (1) and (2) are pushouts in the category* $\mathbf{Graph}_{TG}$.

$$
\begin{array}{ccccc}
L & \xleftarrow{\ l\ } K & \xrightarrow{\ r\ } & R \\
{\scriptstyle m}\Big\downarrow & (1) \quad \Big\downarrow & (2) & \Big\downarrow {\scriptstyle m'} \\
G & \xleftarrow{\ l'\ } D & \xrightarrow{\ r'\ } & H
\end{array}
$$

2.2. UC Formalization

The main purposes of a UC description are the documentation of the expected system behavior and the communication between stakeholders - often including non-technical people - about required system functionalities. For this reason, the most usual UC description is textual, using natural language. Figure 1 depicts a UC of a bank system describing the login operation, executed by a client.

We use a general UC description format, containing a primary actor and a set of sequential steps describing interactions between the primary actor and the system towards a certain goal. A sequence of alternative steps is often included to represent exception flows (*Extensions* in Figure 1), when the primary goal is not achieved, or to describe alternative ways to achieve it. Pre- and post-conditions are also listed to indicate, respectively, conditions that must hold before and after the UC execution.

Name	Log into System via ATM	
Preconditions	User has bank card and registered password. System is running and system asks for card	
Postconditions	User receives menu of available ATM operations	
Primary Actor(s)	Bank Customer	
Main Scenario	**Step**	**Action**
	1	User inserts card
	2	System asks for password
	3	User enters password
	4	System validates user's card and password and display menu of options
Extensions	**Step**	**Branching Action**
	4a_1	System notifies user that identification is invalid
	4a_2	System exits option, releases the card and returns to initial state.

Figure 1: Login Use Case description.

In [3], we proposed a UC formalization and verification approach based on the GT formalism. Given a UC, we build a *type graph* containing all relevant types for this UC and a set of rules describing its steps, one rule for each step, which makes it easy to understand the generated model and to trace it back to the UC. Figure 2 presents the type graph for our bank example: a card can be either in possession of a user or inside an ATM; a user can be logged to an ATM and has a password for access purposes, which is provided to the ATM via an input; the ATM can generate outputs to a user via a textual message.
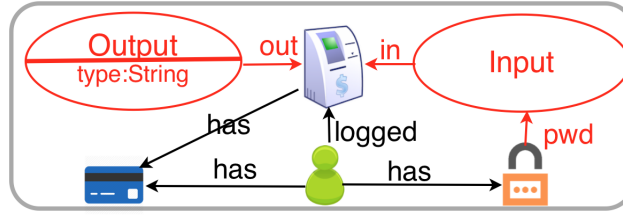

Figure 2: Type graph.

Figure 3 presents the graph transformation rules derived from the *Login Use Case*, shown in Figure 1. Each rule provides a representation of a step of the UC (see labels inside black boxes). For instance, the act of a user inserting a card into an ATM is represented by the creation of an edge between the ATM and the card (rule InsertCard). The negative application condition (NAC) of rule InsertCard is used to disable the insertion of a card if there is already a card inside the ATM. The Output node represents a message on the ATM screen, which is used to interact with a user. Its type indicates the type of the displayed message. The gluing component of the rule (K) is not explicitly
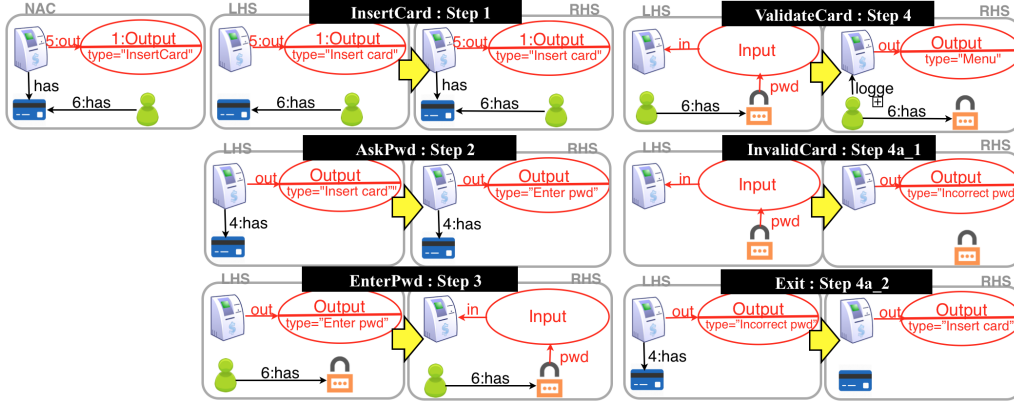
Figure 3: Graph rules obtained from the individual steps of the example Use Case (Figure 1).

represented; rather it is composed by items that are both in the LHS and the RHS of the rule (in the edges and Input/Output nodes, numbers are used to make the mapping more explicit). Similarly, the Input node is used to represent data input. For example, rule AskPwd changes the output node type in the presence of a card. Then, rule EnterPwd represents the act of a user entering a password. From this point on, there are two possibilities: either the user enters the right password, and the system creates a user session (rule ValidateCard), or the password is invalid and the system presents a message (rule InvalidCard), and then ejects the card (rule Exit).

After obtaining a first version of the GT from an UC, a series of automatic verifications can be performed using the AGG tool [4]. All detected issues are annotated as *open issues (OIs)* along with possible solutions (when applicable). Hence, any design decision made over an OI can be documented and tracked back to the original UC. One important point is that, during the formalization process, clarifications and decisions about the intended semantics of the textual description must be made. Annotated OIs force stakeholders to be more precise and explicit about tacit knowledge and unexpressed assumptions about system invariants and desired behavior. The result of applying the approach proposed in [3] is an improved UC, as well as a GT representing the UC described behavior. This GT is the starting point of our contribution presented here: we propose a formal definition of evolution and an analysis method that shows the impact of this evolution on UCs (without actually performing the evolution). The analysis method is based on finding critical pairs between evolution rules and rules describing UC steps. We now review the main concepts of critical pair analysis in the context of GT.

2.3. Use Case Analysis based on Critical Pairs

In GT, conflicts and dependencies between rules can be detected by computing *critical pairs*, which are pairs of rules such that the application of one rule may have an impact on the application of the other (enabling or disabling it). For example, if a rule r_1 deletes an item of type X and a rule r_2 needs this item to be applied, then applying r_1 may prevent the application of r_2. Notice that this is a *potential* conflict: if there are multiple items of type X in the state graph, then the two rules may be applied independently, using different instances of X. There are essentially two types of conflict that may arise in GT systems with NACs [8]:

Use-delete (ud): one rule uses (deletes or preserves) an element that is deleted by another rule;

Produce-forbid (pf): one rule produces an element that triggers some NAC of another rule.

A pair of rule applications may have a conflict in just one direction (r_1 is in conflict with r_2, but r_2 is not in conflict with r_1), or in both directions, i.e., each rule deletes items that the other rule uses, or creates items that trigger a NAC of the other rule. We refer to the latter cases, respectively, as **delete-delete (dd)** conflicts and **double-produce-forbid (ff)** conflicts. Note that, since rules may delete/create/preserve many items, two rules may be simultaneously in more than one type of conflict. Although there may be infinite conflicting situations involving two rules (because conflicts depend on the actual state in which rules are applied and there is usually an infinite set of states), there are only a finite number of critical pairs, which makes them useful for static analysis techniques. Computing critical pairs involve comparing types of items created/preserved/deleted/forbidden by two rules. Since the comparison is based on types, *critical pair analysis (CPA)* detects potential conflicts between rules. For the formal definition of critical pairs see [8].

Dependencies between two rules r_1 and r_2 arise when rule r_1 creates some node or edge which is required for the application of rule r_2, or when r_1 deletes something that is forbidden by a NAC of r_2. They can be defined analogously to conflict critical pairs.

CPA is essentially a brute force calculation of all critical pairs for all possible pairs of rules. From this, it is possible to present an enumeration of all potential conflicts/dependencies between rule applications or, alternatively, to display this information visually as in Figure 4. This is called a *CPA graph*: nodes represent the rules and edges represent possible conflicts (solid lines) and dependencies (dotted lines).

Example: *Figure 4 shows the conflicts and dependencies between rules from Figure 3. It is possible to see that all rules are in conflict with themselves. This represents the fact that, after a match is found and the rule is applied, it cannot be applied again in the same part of the graph. In a UC, this means that each step of the UC is supposed to be performed only once. Rules* **ValidateCard** *and* **InvalidCard** *are related by an undirected conflict line: they are in* **dd** *conflict. In this case, this is an expected conflict, as these rules represent a decision point of the UC. Rule* **InsertCard** *is in* **ff** *conflict with itself, which is also expected as one should not insert a card after it has already been inserted. This graph was automatically generated from the rules by the AGG tool, which can also inform which type of conflict/dependency each arrow represents and show the elements that caused the conflict/dependency.*
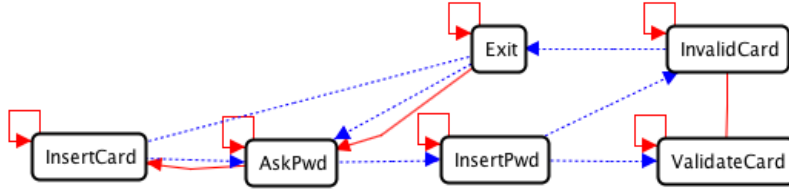

Figure 4: Conflicts and Dependencies from Figure 2.

Each unexpected conflict/dependency in the CPA graph represents a possible error either in the UC text or in its corresponding model. Thus, by checking each edge of the CPA graph, an analyst performs a guided verification of both the UC original text and its model. Problems can then be immediately corrected in either (or both) artifact(s). In [3], we provided a list of OIs that may arise from the analysis of this graph. When all issues are solved, the UC must be manually updated according to the changes made to the rules so as to improve it and keep it up-to-date.

2.4. Second-order Graph Transformation

In this section we review the notion of rule-based modification of typed graph rules with and without NACs presented in [9, 5]. This rewriting of rules is based on the DPO approach, thus the rule format remains : $L \leftarrow K \rightarrow R$. However, instead of rewriting graphs, rules will be used to rewrite other rules. This new rule scheme is called second-order rule, or 2-rule for simplicity, and it requires the definition of morphisms between rules.

Definition 6 (span, span morphism). *A (typed graph)* span *is a diagram with shape $G \xleftarrow{l} G' \xrightarrow{r} G''$ in the category of T-typed graphs. For convenience, we refer to spans as the pair of morphisms (l, r) with common source. A **span morphism** $f : s \rightarrow s'$ between spans $s = (l, r)$ and $s' = (l', r')$ is a triple (f_L, f_K, f_R) of typed graph morphisms between the objects of the spans such that the diagram below commutes.*

$$
\begin{array}{ccccc}
L & \xleftarrow{\ \ l\ \ } & K & \xrightarrow{\ \ r\ \ } & R \\
f_L \downarrow & = & \downarrow f_K & = & \downarrow f_R \\
L' & \xleftarrow{\ \ l'\ \ } & K' & \xrightarrow{\ \ r'\ \ } & R'
\end{array}
$$

*A rule morphism is mono/epi/isomorphic if all three morphisms are also mono/epi/isomorphic. Spans and span morphisms constitute a category, named **Span**.*

Definition 7 (rule, rule morphism). *A (typed graph) rule r is a span $L \xleftarrow{l} K \xrightarrow{r} R$ such that l and r are* monomorphisms, *i.e. injective typed graph morphisms. A* rule morphism $f : r \rightarrow r'$ *is a span morphism (f_L, f_K, f_R) between rules. Notice that f_L, f_K and f_R need not be injective.*
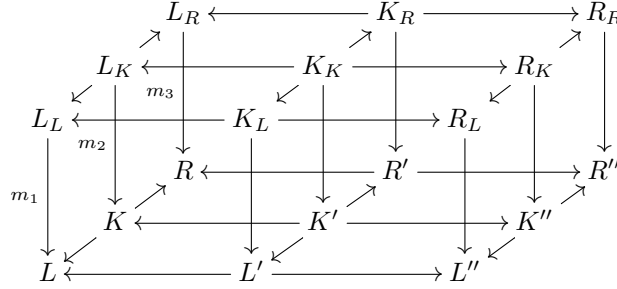
Definition 8 (second-order rule (2-rule)). *A second-order rule is span of monomorphic rule morphisms. A second-order rule with NACs is a pair (s, NAC_s) composed by a second-order rule s and a set of (second-order) NACs for s, where a second-order NAC is defined as a rule morphism with source on the left hand side of s.*

Second-order rules are analogous to first-order rules, the difference is that, instead of defining transformations of graphs, they define transformations of rules (that transform graphs). Examples of second-order rules will be given in Section 4 defining evolutions the use case illustrated in Section 2.2.

The transformation of a rule by means of a 2-rule is defined by means of a DPO diagram in the category **Span**, as in the case of graphs. One caveat exists, however: the resulting span may not be a valid rule because injectivity is not necessarily preserved by

the rewriting. To mitigate this, it is possible to build (for each individual 2-rule) a set of *structure preserving* NACs that forbids any match that would result in a ill-formed rule. In the following, we omit this set of structure-preserving NACs because they only affect the selection of 2-rule matches, not the second-order rewriting itself. The construction of this set of NACs is detailed in [9, 5].

Definition 9 (second-order transformation, rule evolution). *Consider a second-order rule $\alpha = (L_{\{L,K,R\}} \leftarrow K_{\{L,K,R\}} \rightarrow R_{\{L,K,R\}})$, and a first-order rule $p = (L \leftarrow K \rightarrow R)$, as in the diagram below. The upper part of this diagram is a 2-rule. A second-order match is a rule morphism from $L_{\{L,K,R\}}$ to p. Let $l = (L_L \leftarrow K_L \rightarrow R_L)$, $k = (L_K \leftarrow K_K \rightarrow R_K)$ and $r = (L_R \leftarrow K_R \rightarrow R_R)$. A second-order transformation $p \xRightarrow{\alpha, m_{\{1,2,3\}}} p''(L'' \leftarrow K'' \rightarrow R'')$ is defined by the diagram below, where $L \xRightarrow{l,m_1} L''$, $K \xRightarrow{k,m_2} K''$ and $R \xRightarrow{r,m_3} R''$ are typed graph transformations and $(L' \leftarrow K' \rightarrow R')$ and $(L'' \leftarrow K'' \rightarrow R'')$ are valid typed graph rules.*

The span of rules $p \leftarrow p' \rightarrow p''$ (the floor of the diagram above) is called rule evolution.

3. Second-order transformations and NACs

Negative Application Conditions (NACs) are widely used in practice, being very important in the modelling of real systems. In particular, conditional constructs within a use case are mapped in a very direct way to rules with NACs.

The fact that second-order rewriting as defined in [9] does not provide support for modifying rules containing NACs is an important limitation for its applicability. To be able to employ second-order principles to describe and analyse evolution of use cases, our formal setting needs to support the transformation of rules with NACs. Formally, we need a solution for the following problem:

Evolution of first-order NACs. Given a rule with NACs (p, NAC_p) and a second-order transformation $p \Rightarrow p''$ transforming $p = L \leftarrow K \rightarrow R$ into $p'' = L'' \leftarrow K'' \rightarrow R''$, the question is how to obtain a set $NAC_{p''}$ to build a rule with NACs $(p'', NAC_{p''})$ maintaining the same semantics of the set NAC_p with respect to p.

For this analysis in particular, notice that the full second-order DPO diagram is not needed: we only require the respective rule evolution. Therefore, given a rule evolution transforming p into p'', and a set of NACs for p, we intend to obtain a set of NACs over
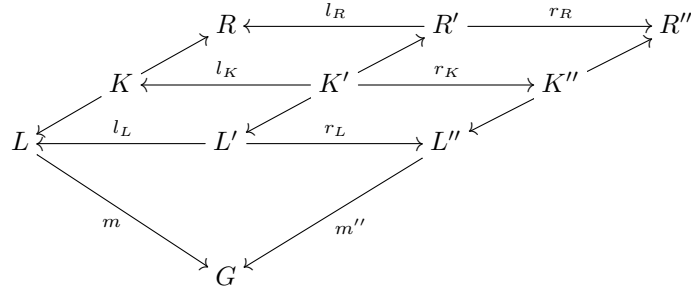
p'' that, ideally should forbid and allow exactly the same corresponding matches: i.e., the semantics of the NACs is preserved by the transformation. As the preconditions (the left-hand side) of the rules can be modified by a second-order rule, the set of NACs must be altered in a compatible way.

A solution for this problem has been found recently in the context of a master thesis [10]. In this (rather technical) section we describe this solution, which serves as basis for our implementation. We show that it is possible to generate a new set of NACs provided that we restrict our matches to be injective.

We start formalizing the notion of semantic preservation, and characterize the situations in which such a semantics preservation occur. For a fixed graph G, we start by defining when matches rules p and p'' in G are related.

Definition 10 (Related matches). *Given a rule evolution $p \xleftarrow{l} p' \xrightarrow{r} p''$ (where $p = L \leftarrow K \to R$, $p' = L' \leftarrow K' \to R'$ and $p'' = L'' \leftarrow K'' \to R''$), a typed graph morphism m: $L \to G$ (representing a match for p in G) and a typed graph morphism m'': $L'' \to G$ (representing a match for p'' in G). We say that m and m'' are related matches (by means of the rule evolution) if and only if*

- *m satisfies DPO gluing conditions for p*

- *m'' satisfies DPO gluing conditions for p''*

- *the equation $m \circ l_L = m'' \circ r_L$ holds, i.e. the following diagram commutes*

Intuitively, matches of a rule and its evolution are related when they are essentially the same considering the part of the left-hand side of the rule that was preserved by the rule evolution. The following definition describes the meaning of semantics preservation of NACs in an evolution.

Definition 11 (Preservation of NAC-behavior). *Let $p \leftarrow p' \to p''$ be a rule evolution, NAC_p be a set of NACs for p and $NAC_{p''}$ be a set of NACs for p''.*
We say that

- *$NAC_{p''}$ preserves the NAC-blocking behavior of NAC_p when, for every $m : L \to G$ and related $m'' : L'' \to G$, we have*

$$NAC_p \not\models m \Rightarrow NAC_{p''} \not\models m''$$

- $NAC_{p''}$ *preserves the NAC-allowing behavior of* NAC_p *when, for every* $m : L \to G$ *and related* $m'' : L'' \to G$, *we have*

$$NAC_p \vDash m \Rightarrow NAC_{p''} \vDash m''$$

- $NAC_{p''}$ *preserves the NAC-behavior of* NAC_p *whenever* $NAC_{p''}$ *preserves the NAC-blocking behavior and NAC-allowing behavior of* NAC_p.

The purpose of NACs is to enable or disable a match m of rule r in graph G. Roughly speaking, in DPO, a match is forbidden if (at least) one of the following situations occur:

- G has some element which does not appear in (the image of) L but appear in (the image of) a NAC;

- m identifies elements preserved by r, but these elements are not identified in a NAC;

- m does not identify elements preserved by r, but these elements are identified in a NAC.

These effects can be observed in the Figure 5, which shows a rule $L \leftarrow K \to R$ with set of NACs $\{n_1, n_2\}$ together with three graphs G_1, G_2 and G_3, together with one match for each graph: $m_1 : L \to G$, $m_2 : L \to G_2$, and $m_3 : L \to G_3$. The mappings are provided by the numbers within nodes. Match m_1 is disabled by NAC n_1 due to the presence of a star in G_1. Match m_2 is disabled by NAC n_2 due to the identification of the circles. Match m_3 is not disabled by n_2 because n_2 does not identify the squares, even considering that m_3 identifies the circles as specified by n_2.

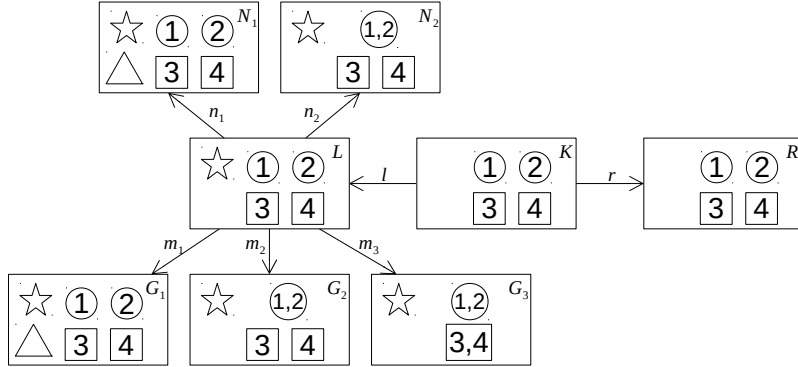

Figure 5: NACs enabling and disabling matches.

Given a rule evolution $p \leftarrow p' \to p''$ and a set NAC_p of negative application conditions for p, we now define how to build a set $NAC_{p''}$ of NACs for p''. We will show that $NAC_{p''}$ preserves the NAC-blocking behavior of NAC_p. We also present a counter-example that shows that a NAC-allowing behavior preservation is not possible in general, considering arbitrary evolutions, and propose a restriction to guarantee NAC-allowing behavior preservation.

In [11], Lambers defined a way to construct, from a graph A with NACs, a set of equivalent NACs for a graph B related to A via a morphism $t : A \to B$. This construction is called *NAC-shift*.

Definition 12 (NAC-shift (over a morphism)). *Given a NAC $n : A \to N$, a morphism $A \to B$ and the diagram below, (1) is a NAC-shift if:*
 (i) (1) commutes.
 (ii) t' and n' are jointly epi.
 (iii) t' is mono.

$$
\begin{array}{ccc}
N & \xrightarrow{\;t'\;} & N' \\
\big\uparrow{\scriptstyle n} & (1) & \big\uparrow{\scriptstyle n'} \\
A & \xrightarrow{\;t\;} & B
\end{array}
$$

Definition 13 (Shift of a NAC along a morphism). *Given a NAC $n : A \to N$ and a monomorphism $t : A \to B$:*
 $D_t(n) = \{n' \mid n' : B \to N', t' : N \to N'\}$ *where (1) is a NAC-shift.*

Definition 14 (Shift of a set of NACs). $D_t(NAC) = \bigcup\limits_{n \in NAC} D_t(n)$

Using NAC-shift it was possible to describe the NACs evolution process as a transformation of two steps: first a pushout complement (as in DPO transformations), and then the shift of NACs over a morphism. Note that in this process each NAC can be evolved to zero or many NACs, which is not an issue since our aim is only to preserve the behavior of the NAC, and not the NACs themselves. This construction was shown in [11] to preserve the NAC-blocking behavior.

Definition 15 (Evolution of a set of NACs). *Let $p \leftarrow p' \to p''$ be a rule evolution, where $p = L \leftarrow K \to R$, $p' = L' \leftarrow K' \to R'$ and $p'' = L'' \leftarrow K'' \to R''$. Let NAC_p be a set of NACs for p. We define the* evolved set of NACs $NAC_{p''}$ *as*

$$NAC_{p''} = D_t(NAC_p)$$

Theorem 1 (Preservation of NAC-blocking behavior). *Let $p \leftarrow p' \to p''$ be a rule evolution, where $p = L \leftarrow K \to R$, $p' = L' \leftarrow K' \to R'$ and $p'' = L'' \leftarrow K'' \to R''$. Let $m : L \to G$ and $m'' : L'' \to G$ be related matches. Let NAC_p be a set of NACs for p, and $NAC_{p''} = D_t(NAC_p)$.*

$$\text{If } NAC_p \not\models m(p,G) \text{ then } NAC_{p''} \not\models m''(p'',G).$$

Proof. See [11]. □

However, the preservation of NAC-allowing behavior is not as straightforward as the preservation of NAC-blocking behavior. We start presenting a counterexample involving non-injective matches.

Example 1 (Invalidation of NAC-allowing behavior). *Figure 6 depicts an example of evolution of a rule with NACs. The intermediate rule of the evolution is isomorphic to the right-hand side rule and it is omitted from the diagram. The original rule deletes a*
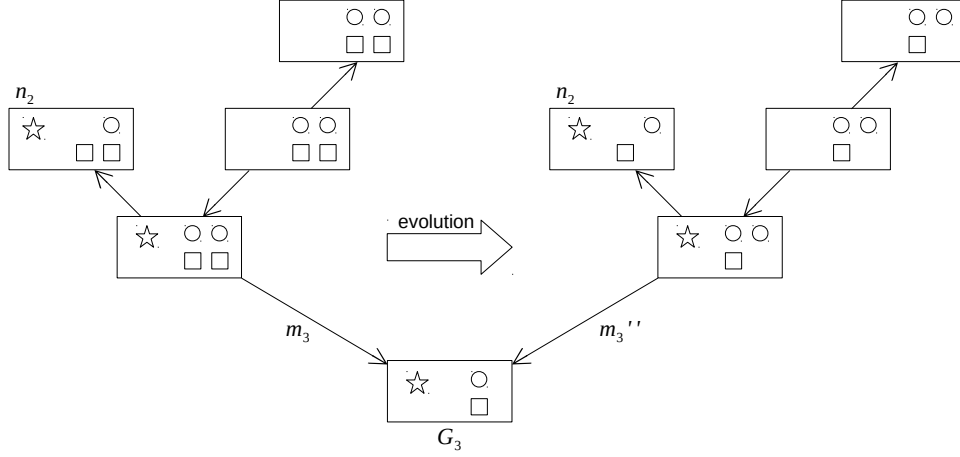
Figure 6: Evolution where NAC-allowing behavior is not preserved.

star in the presence of two circles and two squares. Its only NAC $n_2 : L \to N_2$ forbids the application whenever the preserved circles are identified. Notice that $m_3 : L \to G_3$ is not disabled by n_2, since there is only one square, and therefore there is no injective morphism $e : N_2 \rightarrowtail G_3$. The evolution consists in removing one of the preserved squares from the rule, generating a rule that deletes a star in the presence of two circles and one square. The NAC evolution generates a single NAC $n_2'' : L'' \to N_2''$, without the deleted square. Notice, however, that what prevented NAC n_2 from disabling m_3 was the impossibility of identifying the squares in a monomorphism. Since now there is only one square, there is actually a possible monomorphism $e'' : N_2'' \rightarrowtail G_3$ for n_2'', and therefore n_2'' disables the match m_3'', which is related to m_3 by the evolution.

This situation occurs because some NACs forbid more than one (preserved) elements and if an evolution deletes some of these preserved elements, the distinction between matches that would be allowed or disabled by the NAC may disappear. If, however, we restrict second-order transformations to injective matches, these problematic situations are impossible, and it is possible to obtain preservation of NAC-allowing behavior, as we show in the following theorem.

Theorem 2 (Preservation of NAC-allowing behavior for injective matches)**.** *Let $p \leftarrow p' \to p''$ be a rule evolution, where $p = L \leftarrow K \to R$, $p' = L' \leftarrow K' \to R'$ and $p'' = L'' \leftarrow K'' \to R''$. Let $m : L \rightarrowtail G$ and $m'' : L'' \rightarrowtail G$ be related injective matches. Let NAC_p be a set of NACs for p, and $NAC_{p''} = D_t(NAC_p)$.*

$$\text{If } NAC_p \vDash m\,(p, G) \text{ then } NAC_{p''} \vDash m''\,(p'', G).$$

Proof.

14

- (by definition)
  If for all $n : L \to N \in NAC_p$ there is no monomorphism $e : N \to G$ such that $e \circ n = m$, then for all $n'' : L'' \to N'' \in NAC_{p''}$ there is no monomorphism $e'' : N'' \to G$ such that $e'' \circ n'' = m''$.

- (contrapositive)
  If there exists $n'' : L'' \to N'' \in NAC_{p''}$ and monomorphism $e'' : N'' \to G$ such that $e'' \circ n'' = m''$, then there exists $n : L \to N \in NAC_p$ and monomorphism $e : N \to G$ such that $e \circ n = m$.

- (existence of monomorphism e) consider the diagram below:

  - assume monomorphisms $m : L \rightarrowtail G$, $m'' : L'' \rightarrowtail G$ and $e'' : N'' \rightarrowtail G$ in the diagram above. Notice that each $n'' \in NAC_{p''}$ was created from some $n \in NAC_p$ by means of a pushout complement and a NAC shift commutative square, as shown in the diagram;
  - $n'' : L'' \to N''$ is mono because $m'' = e'' \circ n''$ and m'' is mono. By a similar argument, note that n' and n are also mono.
  - let $e' : N' \to G$ be the composition of monos $e'' \circ r_N$;
  - let (u, v) be the pushout of (e', l_N). Because the category of graphs is adhesive, pushouts preserve monomorphisms and, therefore, both $u : N \to H$ and $v : G \to H$ are mono;
  - let $e : N \to G$ be the unique arrow from pushout square (n', l_L, L_N, n) towards the cospan (m, e');
  - $e : N \to G$ is mono because $u = v \circ e$, and u is mono.

  $\square$

As Theorems 1 and 2 show, NACs preservation is constrained by the kind of morphism allowed as a rule match:

- with general matches, only preservation of blocking behavior is possible.

- with injective matches, preservation of allowing and blocking behavior is possible.

Considering this scenario, we use Definition 15 as appropriated algorithm for evolving a set of NACs along a rule evolution, and we must employ only injective matches in our graph grammars. This is the basis of our implementation of a second-order transformation model for first-order rules with NACs.

4. Use Case Evolution

As software is always evolving, it is necessary to create ways to specify this evolution and analyze the impact of changes. In order to evolve use cases, a very simple idea would be to manually modify the UCs. However, modifying directly the UCs would mean to work on an inherently ambiguous and imprecise language rather than using the corresponding formal description (GT). Moreover, modifying the UC would make the GT model of the system useless, as consistency would not be maintained, unless a new translation was executed. Another aspect is that in large systems the effect of modifying one UC on the whole system is not always evident, since changes may propagate to other UCs due to dependencies. On the other hand, by modifying the GT model directly, one can precisely define evolution (enabling automation) as well as get a detailed impact analysis, whilst keeping all specification artifacts coherent with the desired evolution.

We propose to describe and analyze evolution using GT, updating the corresponding UCs only at the end of the process. Hence, we only go back to the UCs after we are certain that the evolution has the desired effect on the system. In this way, evolution leads to an updated formal model (GT) and corresponding updated UCs.

In order to represent evolution in GT, we use as formal foundation *second-order rules* [5], which are rules that modify other rules (the first-order rules). In this work, we call second-order rules *evolution rules*, describing intended modifications to be applied to an existing GT. The main idea is to construct a rule that defines a modification pattern. This pattern denotes the items that must be changed and how they shall change for a given evolution. Then, this pattern can be compared to all rules of a GT model, called *step rules* (rules describing steps of a UC). In case of a match, the rule is modified accordingly (an idea very similar to the use of aspects in programming [12]). This way of formalizing the definition of an evolution guarantees a precise description, using a language similar to the one used to describe the system itself. It also enables the possibility of analysis of changes due to the evolution process (side-effects).

The proposed method consists of 3 steps:

Step 1 – Definition of evolution: The developer defines any behavior that has to be modified, added to, or suppressed from the system in terms of GT rules. Behaviors to be modified are described as evolution rules (second-order rules), whereas new behavior is described as step rules (first-order rules). Rules to be deleted may be specified explicitly or be selected in Step 3.2.

Step 2 – Analysis of evolution: Using evolution rules, analysis based on critical pairs (CP) can be performed:

Step 2.1 – CPs between evolution rules: give hints on consistency and termination of the evolution process;

Step 2.2 – CPs between evolution and execution: describe how system's step rules (and corresponding UC steps) will be impacted by evolution rules. By analyzing these CPs, the developer has to decide whether the evolution should be performed as is or modified to prevent undesired behavior (the latter case means going back to Step 1);

Step 3 – Execution of evolution: Perform the intended evolution:

Step 3.1 – Generate a new GT: Execute the modifications defined in Step 1 on the original GT;

Step 3.2 – Analyze the new GT: Construct the CPA graph for the new GT and identify rules that shall be deleted (became useless), perform deletion, and redo analysis until the GT exhibits the desired behavior;

Step 3.3 – Modify UCs: UCs whose step rules were affected by evolution should be rewritten accordingly.

Before detailing how evolution is defined and performed, we illustrate the proposed method with a simple example of evolution applied to our running. This example considers only one UC, but more complex case studies are reported in Section 5.

Figure 7: Evolution Rule

Example: *(Evolution 1: Insert fingerprint login.) As an example of evolution, suppose the bank now decides to change the way in which a client logs in via an ATM: besides the original card-and-password method, the bank also wants to allow identification via fingerprint. With this change, any step of a UC leading back to the start state of the ATM, showing the* **Insert card** *message, would have to be modified to show a* **Choose id** *message, indicating that the user now has to select the identification method to be used.*

Step 1: *Figure 7 describes an evolution rule modeling this modification. To stress the orthogonal nature of evolution with respect to the behaviour of the system, evolution rules will be depicted vertically (contrasting to the horizontal description of step rules). The LHS of the evolution rule (shown above the downward arrow) is the pattern to be found in a step rule of the specification to trigger a modification on that rule. In this example, it matches all step rules where an ATM is preserved and an* **Output** *node of type "***Insert Card***" is generated. The effect, as specified by the evolution rule's RHS (rule pattern below the downward arrow), is to delete the* **Output** *node of type "***Insert card***" and replace it by an* **Output** *node of type "***Choose id***". The NAC of the evolution rule (shown on the top) is used to ensure that this transformation is applied only to step rules that create an* **Output** *node "***Insert Card***", not to rules that preserve it. Besides changing existing step rules, new rules are added to model the new behavior (see Figure 8): now the system starts by asking the user to choose between card or fingerprint identification (rules* **ChooseId-card** *and* **ChooseId-fprt***), and rules that treat insertion and validation of fingerprints are added.*

Step 2: *This evolution consists of only one rule, which is in conflict with itself because, once applied, it will change the step rule in a way that the evolution rule can not be applied again (to this step rule). Thus, the evolution process is consistent and terminates. By*
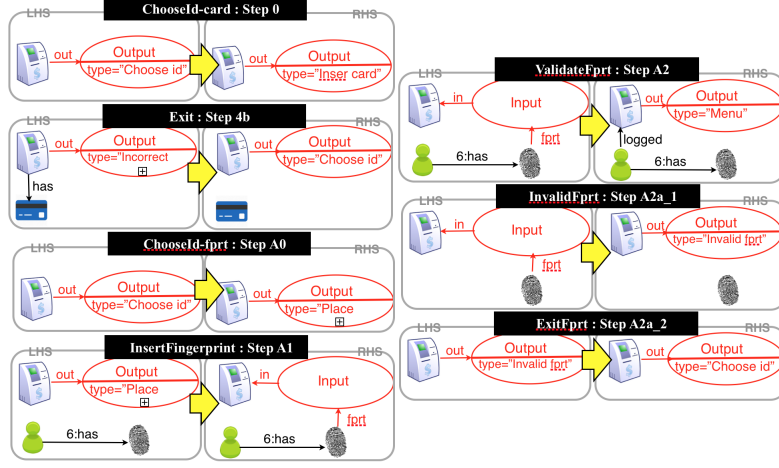
Figure 8: UC Rules after application of the Evolution Rule

analyzing the matches and conflicts of the evolution rule with the execution ones, it is possible to detect all step rules that should be modified and, consequently, all UC steps that should be changed to apply the evolution. In our example, the only step rule that would be modified is rule **Exit** *(corresponds to Step 4a_2), that will now generate a message* **Choose id** *instead of* **Insert card**. *This is exactly what was expected, hence the evolution can be performed.*

Step 3: *We can now perform the evolution (Step 3.1) and generate the CPA graph (Step 3.2 – see Figure 9). In this example no step rule shall be deleted (the next example will illustrate this case) and we can thus update the corresponding UC: steps of the original UC corresponding to unchanged step rules remain unchanged; steps that correspond to changed rules must be updated; and steps corresponding to new rules must be added. This UC is shown in Figure 10.*
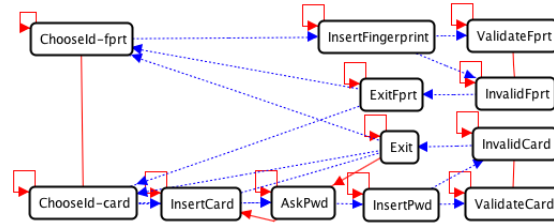

Figure 9: CPA graph - evolution 1.

4.1. Formalization of Evolution

This section presents the formalization of the evolution steps previously discussed and applied to to our ATM example.

Step 1: We now define how to represent the evolution of a system.

Name	Log into System via *ATM*
Preconditions	User has *bank card* and registered *password (or fingerprint), system is running and system asks for login option*
Postconditions	User receives *menu* of available *ATM* operations
Primary Actor(s)	Bank *Customer*

Main Scenario	Step	Action
	0	User chooses to log in using card
	1	User inserts card
	2	System asks for password
	3	User enters password
	4	System validates user's card and password and displays menu of options
Extensions	**Step**	**Alternative Actions**
	A0	User chooses to log in using fingerprint
	A1	User places finger in the reader
	A2	System validates fingerprint and displays menu of options
Extensions	**Step**	**Exception Actions**
	4a_1	System notifies user that password is invalid
	4a_2	System exits option, releases card, and returns to initial state
	A2a_1	System notifies user that fingerprint is invalid
	A2a_2	System exits option and returns to initial state

Figure 10: Login Use Case description - evolution 1.

When modifying a graph transformation system, one may alter the type graph (inserting or deleting types of elements), add new rules, delete deprecated rules and update rules by means of second-order rules (which in this context will also be referred as evolution rules). All these modifications are codified by the structure we call *evolution structure*.

Definition 16 (Evolution structure). *Let $\mathcal{G} = (T, P)$ be a GT system. An **evolution structure for** $\mathcal{G}$ is a tuple $ES = (ET, EP, Del, New)$ where*

1. $ET = T \xleftarrow{l} T' \xrightarrow{r} T''$ *is a span of graph morphisms such that l and r are injective. This describes how the type graph is modified by the evolution.*

2. *EP is a set of second-order rules (evolution rules) that describe how to update the rules which are preserved during the evolution; the LHS of each evolution rule must be typed over T and the RHS must be typed over T'';*

3. *$Del \subseteq P$ is a set of GT rules to be deleted;*

4. *New is a set of GT rules (T''-typed) to be created.*

Example: *(Evolution 2: Remove card/password login) Suppose now that the bank decides to remove the old card-and-password method and make the fingerprint recognition the only available login method. In this case, all step rules that contain any reference to card or password would have to be modified. Figure 12 shows examples of two evolution*
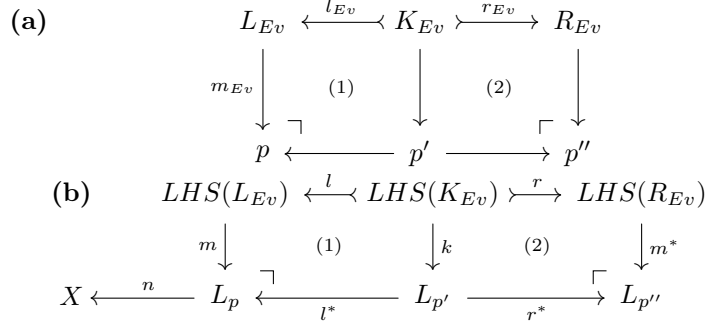
19

$$\textbf{(a)} \qquad L_{Ev} \xleftarrow{\;l_{Ev}\;} K_{Ev} \xrightarrow{\;r_{Ev}\;} R_{Ev}$$

$$m_{Ev} \downarrow \quad (1) \qquad \downarrow \qquad (2) \qquad \downarrow$$

$$p \xleftarrow{\quad} p' \xrightarrow{\quad} p''$$

$$\textbf{(b)} \qquad LHS(L_{Ev}) \xleftarrow{\;l\;} LHS(K_{Ev}) \xrightarrow{\;r\;} LHS(R_{Ev})$$

$$m \downarrow \quad (1) \qquad \downarrow k \qquad (2) \qquad \downarrow m^*$$

$$X \xleftarrow{\;n\;} L_p \xleftarrow{\;l^*\;} L_{p'} \xrightarrow{\;r^*\;} L_{p''}$$

Figure 11: (a) Span rewriting (in **Span**) (b) Rewriting of a rule's LHS (in T-**Graph**)

rules that may be used to accomplish this task. Rules rule-Evol1 and rule-Evol2 state that step rules that preserve/produce bank card should have bank cards removed. Similar rules can describe that references to bank cards, as well as references to Insert card, should be removed. We will call these rules rule-Evol3 (rules that create references to cards should be updated), rule-Evol4 (rules that preserve Insert card should be updated), rule-Evol5 (rules that create Insert card should be updated), rule-Evol6 (rules that delete Insert card should be updated). We also have rule-Evol7, analogous to the rule shown in Figure 7, that rewrites step rules that create Choose id to rules that create Place finger (because now there is no choice of identification method). Finally, rule-Evol8 removes references to Choose id from the LHSs of step rules. The tuple $(T \leftarrow T' \rightarrow T', \{\text{rule-Evo1}, \dots, \text{rule-Evo8}\}, \{\}, \{\})$ is an evolution structure, where T' is obtained from T by removing the node types card, Insert card and associated edge types.
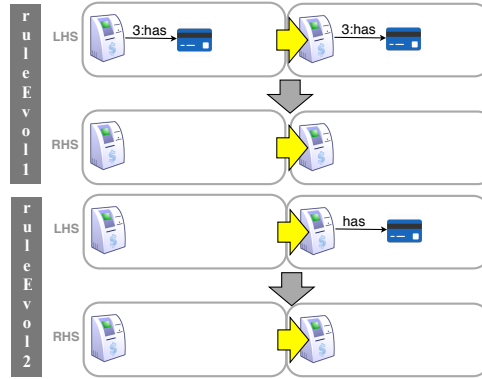

Figure 12: Evolution of rules: Removing card

Step 2.1: In [5], critical pairs between higher-order rules were defined. Here we can apply this notion to evolution rules obtaining a CPA graph describing the conflicts and dependencies. We require that every evolution rule be in conflict with itself and do not depend on itself. Moreover, we require that there are no conflicts nor dependencies between two different evolution rules. This guarantees that every rule can be applied at most once at each match and, therefore, the evolution process will eventually terminate

(all rule sets and graphs are finite) and that the evolution process is deterministic.

Step 2.2: There are different ways in which an evolution rule may impact a step rule. Intuitively, the **gluing problem** occurs when the evolution rule cannot be applied because the (second-order) gluing condition is not satisfied. This means that the application of this rule would lead to side effects; for example, when we are trying to remove some node of the step rule that is referenced by some edge not specified for deletion in the evolution rule. The **NAC-gluing problem** represents a situation in which applying an evolution rule implies the deletion of a NAC of the step rule: if the elements that were forbidden by the NAC are removed, the NAC becomes useless. These situations actually prevent the application of the evolution rule to this step rule since they would lead to side effects that were not foreseen. When this occurs, the developer has to decide whether the side effect is desired (in which case a new rule explicitly stating this not as a side effect but as the effect of the rule must be added) or not (in this case, nothing must be done, since the evolution rule is not applicable to the step rule in which side effects would occur). Other situations to consider are when the evolution rule increases or restricts the step rule applicability (the LHS of the step rule is changed), and when the evolution rule changes the effect of the step rule (the RHS of the step rule is changed). These situations are characterized below.

Definition 17 (Interplay between Evolution and Execution rules)**.** *Given an evolution rule with NACs* $EvR = (EvNAC, \alpha)$ *that is not an isomorphism, where* $\alpha = L_{Ev} \hookleftarrow K_{Ev} \rightarrowtail R_{Ev}$, *a step rule with NACs* $ExR = (exNAC, p)$ *and a morphism* $m : L_{Ev} \to p$, *the interplay between the evolution rule* EvR *and the step rule* ExR *can be classified in 4 groups:*

1. **Situations that prevent evolution:**

   Gluing problem (gp): *there is no unique p' such that diagram (1) of Figure 11(a) is a pushout [3];*

   NAC-gluing problem (np): *there is a NAC $n : L_p \to X \in N$ such that $n \circ l^*$ in Figure 11(b) is an isomorphism;*

2. **Situations that increase rule applicability:**

   Delete^2-delete (d^{2}d): *an item deleted by ExR is deleted from p by EvR;*

   Delete^2-preserve (d^{2}p): *an item preserved by ExR is deleted from p by EvR;*

3. **Situations that restrict rule applicability:**

   Create^2-delete (c^{2}d): *EvR creates an item to be deleted by p (i.e., inserts an item in p's LHS);*

   Create^2-preserve (c^{2}p): *EvR creates an item to be preserved by p (i.e., inserts an item in p's LHS);*

---

[3]The name *gluing* is standard in the graph transformation community, and denotes a situation in which the gluing rule p' can not be determined. This may occur due to a conflict between deletion and preservation of items during rule application, or due to trying to delete items that are connected to others by edges that are not specified for deletion by the rule.

4. Situations that modify rule effect:

Create^2-create (c^{2}c): *EvR creates an item to be created by p (i.e., inserts an items in p's RHS);*

Delete^2-create (d^{2}c): *EvR deletes an item created by p.*

The open issues that may be raised by this analysis step are listed in Table 1.

Open issue	Verification	Problem	Severity level	Possible action
OI.Ev1	Evolution rule not in conflict with itself	The rule may be applied multiple times	🛑 **Red**	Review the evolution rule (adding a NAC or an item to be deleted).
OI.Ev2	Evolution rule dependent on any evolution rule	Evolution may not terminate	🛑 **Red**	Review the rules.
OI.Ev3	Evolution rule in conflict with other evolution rule	May lead to non-deterministic evolution	🛑 **Red**	Check LHSs of conflicting evolution rules to make sure that they are not applicable to the same step rule
OI.Ev4	Gluing problem occured	An evolution is not applicable to some step rule due to extra context	❓ **Orange**	Check whether the evolution should be applicable to this step rule. If not, nothing has to be done. If yes, create a new evolution rule with the extra context, if such rule does not exist.
OI.Ev5	NAC-gluing problem occured	An evolution would make the NAC of a step rule useless	❓ **Orange**	Check the NAC: if it makes sense that it become useless, delete this NAC. Otherwise, review the evolution rule.
OI.Ev6	Undesired increase of (step) rule applicability	Delete^2-delete or Delete^2-preserve situation occurred	❓ **Orange**	Check the items that will be deleted from the step rule by the evolution rule (items deleted in LHS(L_{Ev}).
OI.Ev7	Undesired restriction of (step) rule applicability	Create^2-delete or Create^2-preserve situation occurred	❓ **Orange**	Check the items that will be inserted in the LHS of the step rule by the evolution rule (items created in LHS(R_{Ev}) .
OI.Ev8	Undesired modification of (step) rule behavior	Create^2-create or Delete^2-create situation occurred	❓ **Orange**	Check the items that are created in/deleted from the RHS of the step rule by the evolution rule (items created in/deleted from RHS(R_{Ev}).

Table 1: Open Issues concerning Evolution

Example: *Many step rules in the grammar of Figure 8 will be affected by evolution 2 (removal of card and password authentication). Due to space limitations, it is not possible to show all cases, but only examples of situations that arise from this evolution:*

Gluing problem: *There is a gluing problem between rule **ruleEvol2** and rule **insertCard**: when we try to delete the card from the step rule **insertCard**, the gluing problem warns that there is a connection of this card to a User and thus, by removing the card, we would have to remove this connection as a side effect. If this is the desired effect, either this evolution rule should be updated to include the card, or another evolution rule including the card should be added (in case the two methods – with an without card – are needed). Another possibility would be to mark the step rule to be deleted by the evolution (in this example, this would be the desired effect since without the card the rule do nothing, as shown in Figure 13).*

NAC-gluing problem: *Once again, the application of rule **ruleEvol2** would impact in rule **insertCard**. The NAC would become equal to the LHS of the rule, meaning that it would not forbid anything, becoming useless.*

Delete^2-delete: *rule **AskPwd** is in $d^2 d$-situation with **rule-Evol6** because both delete the node **Output:InsertCard**.*

delete^2-preserve: *rule **rule-Evol4** is in $d^2 p$-situation with rule **InsertCard**.*

Figure 13: Evolution of rule insertCard

The existence of such conflicts means that the evolution step will change the behavior of the affected rules and, therefore, the developer has to check whether the outcome of the evolution captures the desired behavior. If, despite the conflicts, evolution according to the given rules is performed, this means the developer judged that these conflicts do not represent undesired UC behavior.

Step 3.1: Let $\alpha = L_{Ev} \leftarrow K_{Ev} \to R_{Ev}$ be an evolution rule, let p be a graph transformation rule without NACs, and let $m_{Ev} : L_{Ev} \to p$ be a span morphism. The evolution of p to p'' via the application of (α, m_{Env}), denoted by $p \xoverset{\alpha, m_{Env}}{\Longrightarrow} p''$, was defined in [5]. We now extend this notion of evolution to step rules with NACs. If $p \xoverset{\alpha, m_{Env}}{\Longrightarrow} p''$, then $(N, p) \xovertset{\alpha, m_{Env}}{\Longrightarrow} (N', p'')$, where N' is obtained from N as follows: for each NAC $n : L_p \to X \in N$, we can obtain a corresponding NAC $n' : L_{p''} \to X'$ as the universal pushout morphism from pushout (2) in Figure 11(b), where X' is the pushout (object) from $n \circ l^* \circ k$ and r. With this, we can define how an evolution structure evolves a GT system as a whole.

Definition 18 (Evolution). *Let $\mathcal{G}$ be a graph transformation system, and $ES = (T \leftarrow T' \to T'', EP, Del, New)$ be an evolution structure over $\mathcal{G}$. An **evolution** of $\mathcal{G}$ induced by ES is the GTS $G'' = (T'', P'')$ where P'' is obtained by*

1. *$P_1 = P \setminus Del$, i.e., remove all step rules marked for deletion;*

2. *Keep applying evolution rules in EP to step rules in P_1, until no evolution rule is applicable. This gives rise to the set of rules P_2;*

3. $P'' = P_2 \cup New$, i.e., add new rules to the specification.

Step 3.2: This step is a garbage collection step that deletes the step rules that are no longer useful in the system. It is performed by constructing the CPA graph of the result of the evolution and comparing it to the CPA graph of the system before evolution. Rules that are disconnected in this graph are the first candidates for deletion (since they do not use/create any item that is needed by other rules and thus represent actions that are totally independent from others, which is not common in UCs). The second verification concerns branching points: if a branching point (bi-directional conflict) disappeared, probably all rules depending on one of the branches should be removed (because UCs do not have usually two independent execution threads). Third, if a dependency disappears, it should be checked whether the rule is still necessary. Note that selecting the rules to be deleted is a manual task, since it involves choosing which behavior should be kept in the UC. However, the CPA graph helps finding the rules to be deleted. Besides using the CPA graph, we also look for rules that turned into isomorphisms due to evolution (and thus, became useless) and also for rules that are duplicated.

Step 3.3: This step is done by removing from the original UC all steps that correspond to deleted rules, modifying the steps that correspond to changed rules (by comparing the old and new rules, it is usually obvious how the text should be rewritten), and add steps corresponding to new rules. The order of steps must be compatible with the dependency order of the CPA graph of the evolved system.

Example: *(Evolution 2: Remove card/password login – Step 3) By performing an evolution using evolution rules **rule-Evol1** to **rule-Evol8** and deleting step rule **InsertCard**, we obtain the modified step rules shown in Figure 14 (rules corresponding to other steps remain unchanged). After the application of evolution, the analyst may review the specification in order to check whether the remaining rules make sense in the new context.*
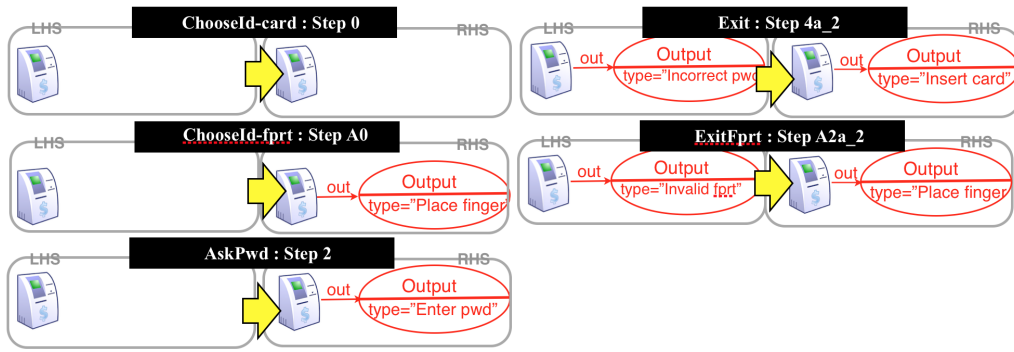

Figure 14: Evolution of rules of UC1

*The CPA graph of the resulting GT is shown in Figure 15 (we included **InsertCard** here to show how it would appear, if it had not been explicitly deleted by the evolution step). Rules **InsertCard** and **ChooseId-card** should be removed from the set of resulting step rules because they have no effect (they do not delete/create anything, and are thus disconnected in the CPA graph). Rule **ChooseId-fprt** is not in conflict with itself and thus can occur*

indefinitely many times: since this choice is actually no longer needed, it may be removed. The starting point of the corresponding UC should be InsertFingerprint. Rules AskPwd, InsertPwd, ValidateCard, InvalidCard and Exit are not reachable from InsertFingerprint. Since these actions are not to be performed in parallel with others (if this would be the case, being independent from InsertFingerprint would be correct), all these rules should also be removed. The resulting UC is shown in Figure 16.
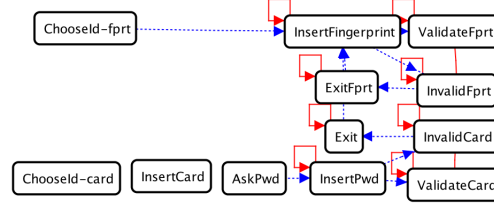

Figure 15: Conflict-dependency graph - evolution 2.

Name	Log into System via *ATM*
Preconditions	User has *bank card* and registered *fingerprint and system is running*
Postconditions	User receives *menu* of available *ATM* operations
Primary Actor(s)	Bank *Customer*

Main Scenario	Step	Action
	A1	User places finger in the reader
	A2	System validates fingerprint and displays menu of options
Extensions	**Step**	**Exception Actions**
	A2a_1	System notifies user that fingerprint is invalid
	A2a_2	System exits option and returns to initial state

Figure 16: Login Use Case description - evolution 2.

5. Case Studies

We present 3 case studies to evaluate the use of our approach. For each case study, we have previously constructed the original GT using the methodology described in [3]. Hence, here we only describe the evolution process of these systems, as well as quantify the impact of the modifications. All original UCs and GTs, the evolution rules, and the modified GTs can be found at `http://www.ufrgs.br/verites`. We used the Verigraph tool[4] to perform analyses on each UC. Verigraph is implemented in the Haskell programming language and offers (amongst other functionalities) critical-pair analysis and evolution of graph transformation systems based on second-order rules. It is currently under development and, although freely available, provides at the moment only a command-line interface. We used the AGG tool [4] to visualise analysis results and to model first- and second-order graph transformation rules.

---

[4]Available at `http://github.com/verites/verigraph`

The first case study involved 3 UCs describing a medical procedure at a Brazilian research institute. The aim was to describe how to apply a saline solution to skin areas with lesions. The quantity of solution and the adequate procedure to inject it on a patient depend on the location, appearance, and size of the lesions. The specification describes how to deal with each possible case. After the translation from the UCs to GT, 32 rules were created. In this case, the evolution consisted in considering two types of lesions (small and large), removing different types of needles, and treating anesthetic buttons in an unified way. This evolution led to the merging of UCs (many rules became isomorphisms and were deleted, 2 UCs became equivalent – i.e., they had repeated rules).

The bank case study contained 5 UCs describing how a customer accesses his account in a given bank through an ATM and how a bank employee can certify that a deposit is correct and, thus, can be completed. The shortest UC contained 7 steps, considering the main scenario and extensions, whereas the longest contained a total of 17 steps. The formalization resulted in a total of 48 rules. From this original GT, we applied evolution 1, described in the running example in Section 4, with an additional evolution rule to store the type of login performed. Based on this evolution, a non-predicted second evolution was necessary. The problem occurred because the login by card and password required a card to be inserted in the ATM, which must be dispensed when the costumer logs out. However, when the login was done by fingerprint, there was no card to be dispensed. Hence, the first evolution allowed a costumer to login by fingerprint but required a card to be dispensed at logout. The new second-order rule modified rules including a marker in situations where the login was by fingerprint. This rule induced inter-level conflicts, since the original rules were applicable to graphs without the marker, unlike the modified rules. This ensured the card must be dispensed at logout only if it was inserted at login, thus allowing the correct logout operation for each login type.

The third case study considered the specification of an e-commerce application[5]. For this case study, we have formalized 5 UCs, considering the UC that corresponds to the login operation and, therefore, is required by all other UCs, and 4 other UCs related to basic operations of the system, such as browsing/searching the catalogue of items and dealing with account information. Each UC contained between 6 and 20 steps. The proposed evolution was to preserve the data of a user account when it is cancelled, rather than simply deleting the account. Hence, when cancelling an account, this account would just be marked as inactive and could be reactivated later on by a system manager. This forced other three necessary modifications: (i) all rules mentioning a user have to consider whether this user has an active account; (ii) when a new user account is created, it must be initialized as deactivated until the user confirms its activation; and (iii) all operations in the system must be restricted only to users with active accounts.

5.1. Results

Table 2 shows the quantitative results of our case studies. It presents, for each case study: the number of UCs of each system; the total number of first-order rules present in the original GT; the number of evolution rules created based on the proposed evolution; number of inter-level conflicts (i.e., when a second-order rule creates a conflict with a

---

[5]Available at `http://www.utdallas.edu/~chung/RE/Presentations07S/Team_3/UseCaseDocument.doc`. Last accessed on July 2, 2019.

first-order rule); the number of rules affected by the evolution; the number of rules deleted from the original GT because they became either obsolete (i.e., inapplicable) or useless (i.e., without any practical effect) after an evolution; and the number of UCs modified as a consequence of the evolution.

Table 2: Summary

Case	Medical	Bank	Marvel
# of UCs	3	5	5
# of 1st-order rules	32	48	32
# of 2nd-order rules	8	2	4
# of inter-level conflicts	3	4	2
# of affected rules	11	7	5
# of deleted rules	7	0	0
# of affected UCs	3	3	3

The number of inter-level conflicts indicates the points where special care has to be taken and decisions concerning the effects of an evolution have to be made. The sum of affected and deleted rules indicates the amount of work that would be necessary to apply the corresponding evolution. This means, for example, that applying the evolution of the Medical case study would require dealing with a total of 18 rules. However, we were able to automatically apply these changes by specifying 8 evolution rules. Considering that this specific case study contained 32 first-order rules, applying the evolution manually by identifying and modifying each rule could be difficult and error-prone. Instead, by specifying evolution rules, we can automatically detect all step rules that have to be changed, as well as which changes should be applied.

We only considered 3 case studies with a few UCs but, even with this reduced number of cases, it was already possible to see the benefits our approach brings in contrast with modifications carried out by hand. In particular, the experiments showed how an evolution may trigger other changes, and keeping track of all these side-effects could be a hard task. Therefore, the experiments demonstrate our approach is useful, scalable, and applicable in practice.

Despite the need of some expertise to formally define the evolution, we noticed that often evolution rules are not complex to specify. Moreover, considering the real gain in having automatic analyses of the impact of system changes, it seems worth the effort. The interpretation of the analysis results also require some knowledge about conflicts and dependencies, but these concepts can be easily associated to similar ideas involving UCs. It is worth to mention that our work provides support for the analysis of the impacts of an intended evolution, but the user has the final decision on implementing it or not. As the user can analyse how the evolution affects the system, they can decide whether the impact is desirable and acceptable or the effects of the proposed evolution are more a problem than a solution.

6. Related Work

CPA has already been used to support evolution in scenarios not connected with UCs, such as described in [13] and in [14]. However, we consider that UC specification is

important as a means to communicate with stakeholders. Although we specify evolution in the GT language, we propose the corresponding changes to be applied to the UCs so as to keep them up-to-date as a documentation of the software for developers and stakeholders.

Considering support for UC evolution, Rui and Butler [15, 16] discussed a metamodel for UCs along with a set of UC refactoring categories based on the metamodel. Refactorings correspond to changes in the syntax or structure of the UCs, such as changing or deleting a UC entity. Although the mechanics of some refactoring operations were discussed, the authors did not evaluate the effect of a UC refactoring in other UCs and offer no support for semantic changes. In [17], an approach for evolution of metamodels described by graphs was presented. They consider how to change the metamodel and guarantee that models constructed from this metamodel are kept consistent. However, neither of the two aforementioned approaches considers a combination of UC evolution and GT to represent and analyze evolution. Although evolution rules work as a sort of metamodel, they are used to describe changes in behavior, rather than only describe restrictions on the connections of elements at the structural level.

In [18], the authors present a tool-supported approach for analysing change impact regarding evolving configurations of product line use case models, which applies an idea similar to ours of impact analysis. They support decision-making about inclusion/exclusion of variant UCs, allowing the analysis of different configurations (i.e., sets and orders of UCs) for a product-line software and the effect of these changes in the specific products derived from this more general UC. A particular version of UC description is used to restrict rules and keywords constraining the use of natural language, thus enabling automatic changes in the product-specific (PS) software once product-line (PL) configuration changes have been approved. In this case, the PL UC plays a similar role as a second-order rules in our work and, due to the restrictions of the language used in the UCs, automated updates are possible. However, they focus on high-level decisions, changing configurations, whereas our approach deals with a fine-grain specification, concentrating on the specific behaviour described by each UC. Moreover, like other strategies used to automate the process of dealing with UCs, they restrict the use of natural language in the UC description. We believe this limits the expression of the requirements in terms of the end-user language and may preclude the expression of unforeseen scenarios. For this reason, we would like to keep the UC description in natural language and in the well-known format, but take advantage of the formal model. Furthermore, we also define an analysis of the impact of evolution on UCs even before their actual application. Therefore, we not only support the evolution of an existing system originated from a set of UCs, but also help the decision-making process related to whether implementing or not a certain evolution.

As far as we know, there is no previous work proposing the use of second-order graph grammars to describe evolution. We advocate that GG is an intuitive way of describing system behaviour and its mapping from UCs creates the possibility of formal analysis on natural-language specification and their modification.

7. Conclusions

We described an approach to support the definition and analysis of impact of use case (UC) evolution, extending previous work on the formalization and analysis of UCs using

graph transformation (GT). In this work, we employed a high-order graph transformation framework as basis for the definitions and analyses. This framework was extended to allow higher-order transformation of rules with NACs, since NACs are required when modelling conditional steps in use cases. Analysis of the impacts of evolution is based on the conflicts between evolution rules and step rules. We detailed the conflicts that may occur and implemented this verification technique in the Verigraph tool. We also provided guidelines to interpret the results of this automatic verification, such that the developer can have a hint on possible sources of evolution errors and on how to correct them (Table 1).

We illustrated the impact of evolution within a single UC as a running example, but our approach may be used to support evolution of systems consisting of several UCs, where the analysis of the impacts of changes is an error-prone and time-consuming task. Results on experiments with larger systems were reported in Sect. 5. We noticed that even small changes usually affect more than one UC in a system, not always in obvious ways. Therefore, techniques to detect the impacts of an evolution in a system that can be automated, like the one proposed in this paper, are highly desirable. Although we used the formal framework to provide a foundation for UC evolution, it is noteworthy that the formal model is generic and could be used as a semantic model for evolution of other artifacts used in the software development process (one would just have to define a translation from this artifact to GT).

As future work, we plan to analyse the impact of evolution rules on the CPA graph of a system, adding new hints on how behavior will be affected by evolution; and to work on test case generation and on improving the automation of evolution by adding detection of isomorphic and duplicate rules.

References

[1] A. Cockburn, Writing Effective Use Cases, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[2] V. Alagar, K. Periyasamy, Specification of Software Systems, Texts in Computer Science, Springer, 2011.

[3] M. A. de Oliveira Junior, L. Ribeiro, E. Cota, L. M. Duarte, I. Nunes, F. Reis, Use case analysis based on formal methods: An empirical study, in: Recent Trends in Algebraic Development Techniques, Vol. 9463 of LNCS, Springer, Sinaia, Romania, 2015, pp. 110–130.

[4] G. Taentzer, AGG: A tool environment for algebraic graph transformation, in: Applications of Graph Transformations with Industrial Relevance, Vol. 1779 of LNCS, Springer Berlin Heidelberg, 2000, pp. 481–488.

[5] R. Machado, L. Ribeiro, R. Heckel, Rule-based transformation of graph rewriting rules: Towards higher-order graph grammars, Theor. Comput. Sci. 594 (2015) 1–23. doi:10.1016/j.tcs.2015.01.034.
URL http://dx.doi.org/10.1016/j.tcs.2015.01.034

[6] H. Ehrig, M. Pfender, H.-J. Schneider, Graph-grammars: An algebraic approach, in: Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on, 1973, pp. 167–180. doi:10.1109/SWAT.1973.11.

[7] H. Ehrig, Introduction to the algebraic theory of graph grammars (a survey), in: International Workshop on Graph Grammars and Their Application to Computer Science, Springer, 1978, pp. 1–69.

[8] L. Lambers, H. Ehrig, F. Orejas, Conflict detection for graph transformation with negative application conditions, in: ICGT'06, Vol. 4178 of LNCS, Springer, Natal, Brazil, 2006, pp. 61–76.

[9] R. Machado, Higher-order graph rewriting systems, Ph.D. thesis, Ph. D. thesis, Instituto de Informática - Universidade Federal do Rio Grande do Sul (2012).

[10] A. Costa, Evolution of negative application conditions on second-order graph rewriting, Master's thesis, MsC thesis, Instituto de Informática - Universidade Federal do Rio Grande do Sul (2019).

[11] L. Lambers, Certifying Rule-Based Models using Graph Transformation, Ph.D. thesis, Elektrotechnik und Informatik der Technischen Universitä Berlin (2010).

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, ECOOP'97, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, Ch. Aspect-Oriented Programming, pp. 220–242.

[13] P. De Leenheer, T. Mens, Using Graph Transformation to Support Collaborative Ontology Evolution, Springer Berlin Heidelberg, 2008, pp. 44–58.

[14] T. Mens, G. Taentzer, O. Runge, Analysing refactoring dependencies using graph transformation, Software & Systems Modeling 6 (3) (2007) 269–285.

[15] K. Rui, G. Butler, Refactoring use case models: The metamodel, in: ACSC '03 - Volume 16, Australian Computer Society, Inc., Darlinghurst, Australia, 2003, pp. 301–308.

[16] K. Rui, Refactoring use case models, Ph.D. thesis, Montreal, P.Q., Canada, Canada (2007).

[17] F. Mantz, S. Jurack, G. Taentzer, AGTIVE 2011, Revised Selected and Invited Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, Ch. Graph Transformation Concepts for Meta-model Evolution Guaranteeing Permanent Type Conformance throughout Model Migration, pp. 3–18.

[18] I. Hajri, A. Goknil, L. C. Briand, T. Stephany, Change impact analysis for evolving configuration decisions in product line use case models, Journal of Systems and Software 139 (2018) 211 – 237. `doi:https://doi.org/10.1016/j.jss.2018.02.021`.