

Use Case Analysis Based on Formal Methods: An Empirical Study

Marcos Oliveira Jr. ^(✉), Leila Ribeiro, Érika Cota, Lucio Mauro Duarte,
Ingrid Nunes, and Filipe Reis

PPGC – Institute of Informatics, Federal University of Rio Grande do Sul (UFRGS),
PO Box 15.064, Porto Alegre, RS 91.501-970, Brazil
{marcos.oliveira,leila,erika,lmduarte,
ingridnunes,freis}@inf.ufrgs.br

Abstract. *Use Cases (UC)* are a popular way of describing system behavior and represent important artifacts for system design, analysis, and evolution. Hence, UC quality impacts the overall system quality and defect rates. However, they are presented in natural language, which is usually the cause of issues related to imprecision, ambiguity, and incompleteness. We present the results of an empirical study on the formalization of UCs as Graph Transformation models (GTs) with the goal of running tool-supported analyses on them and revealing possible errors (treated as open issues). We describe initial steps for a translation from a UC to a GT, how to use an existing tool to analyze the produced GT, and present some diagnostic feedback based on the results of these analyses and the possible level of severity of the detected problems. To evaluate the effectiveness of the translation and of the analyses in identifying problems in UCs, we applied our approach on a set of real UC descriptions obtained from a software developer company and measured the results using a well-known metric. The final results demonstrate that this approach can reveal real problems that could otherwise go undetected and, thus, help improve the quality of the UCs.

Keywords: Use cases · Graph transformation · Empirical study · Model analysis

1 Introduction

Use Cases (UC) [3] are a popular model for documenting software expected behavior. They are used in different software processes, not only for requirement documentation and validation, but also as specifications for system design, verification, and evolution. Hence, they are important reference points within the software development process. In current practice, UC descriptions are typically informally documented using, in most cases, natural language in a predefined structure. Being informal descriptions, UCs might be ambiguous and imprecise.

This work is partially supported by the VeriTeS project (FAPERGS and CNPq).

This may result in a number of specification problems that can be propagated to later development phases and jeopardize the overall system quality [1]. In fact, it is well-known that most software faults are introduced during the specification phase [12]. Nevertheless, it is important to keep UC descriptions in a format familiar to the stakeholders, since they must be involved in the UC definition. Thus, the verification of UCs normally corresponds to manual inspections and walkthroughs [11]. Because the analysis is manual, detecting incompleteness and recognizing ambiguities is not a trivial task. Since software quality is highly dependent on the quality of the specification, cost-effective strategies to decrease the number of errors in UCs are crucial.

Strategies for the formalization of UCs have already been proposed, such as [7, 8, 10, 15]. Many of them assume a particular syntax for UC description tailored for their particular formalisms. This limits the expression of requirements in terms of the stakeholders language and, in some cases, also restrains the semantics of the UC. Moreover, whereas current design techniques are mostly data-driven, which delays control-flow decisions until later phases, many of the used formalisms model UCs as sequences of actions, which may neglect data-related issues. Our aim is to keep the expressiveness of a description in natural language and use a formalism for modeling/analysing UCs that is flexible enough to represent the semantics defined by stakeholders at a very abstract level. Moreover, we advocate that the translation from a UC to a formal model should be performed in a systematic way, guided by well-defined steps (possibly aided by tools), such that the model can be obtained without an expert in the formalism (because the expertise is embedded in the predefined translation process). This is fundamental for the adoption of formal methods in practice.

In this paper, we investigate the suitability of Graph Transformation (GT) [5, 14] as a formal model to describe and analyze UCs. Some reasons for choosing GT are: the elements of a UC can be naturally represented as graphs; it is a visual language; the semantics is very simple yet expressive; GT is data-driven; there are various static and dynamic analysis techniques available for GT, as well as tools to support them. We work towards an approach that integrates UC formalization and tool-supported analysis, with the objective of improving the quality of UCs. As the formalization requires a precise description of the behavior described in the UC, the process of translating it into a formal model may already reveal errors. The goal is to define a sequence of steps to guide the process of building the formal model, executing analyses, and evaluating the results in terms of the level of severity of errors. Diagnostic feedback should also be provided, indicating possible actions to solve the detected problems through modification of the original UC. Hence, the process should, iteratively and gradually, improve an initial UC and generate, as result, not only a more precise UC, which can still be presented to non-technical stakeholders and be readily used without affecting the usual development process, but also a corresponding formal model that can be refined and used in subsequent design activities. This paper presents the first steps towards such a process, presenting an outline of the idea and an empirical evaluation of the effectiveness of the translation and

of the analyses in identifying problems in UCs. We applied our approach on a set of real UC descriptions obtained from a software development company and measured the results using a well-known metric. The final results demonstrate that this approach can reveal real problems that could otherwise go undetected and, thus, help improve the quality of the UCs.

This paper is organized as follows: Sect. 2 presents the necessary background information and details of the translation from UCs to GTs, as well as a detailed description of each step of our approach applied to a running example; Sect. 3 presents the settings of the conducted empirical study; Sect. 4 presents an analysis and discussion of results; Sect. 5 discusses threats to the validity of our work; Sect. 6 presents a comparative analysis of our technique in relation to some similar techniques; and Sect. 7 concludes the paper and discusses future work.

2 Modeling UCs Using GTs

2.1 Background

Use Cases a *Use Case (UC)* defines a contract between stakeholders of a system, describing part of the system behavior [3]. The main purpose of a UC description is the documentation of the expected system behavior and to ease the communication between stakeholders, often including non-technical people, about required system functionalities. For this reason, the most usual UC description is the textual form. A general format of a UC contains a unique name, a primary actor, a primary goal, and a set of sequential steps describing the successful interaction between the primary actor and the system towards the primary goal. A sequence of alternative steps are often included to represent exception flows. Pre- and post-conditions are also listed to indicate, respectively, conditions that must hold before and after the UC execution.

Figure 1 depicts an example of UC of a bank system in a typical textual format, describing the log in operation executed by a bank client. We explain our approach using this UC as example.

Graph Transformations. The formalism of *Graph Transformations (GT)* [5, 14] is based on defining states of a system as graphs and state changes as rules that transform these graphs. Due to space limitations, in this section, we only provide an informal overview of the notions used in this paper. For formal definitions, see e.g. [14]. Examples of graphs, rules and their analysis are presented in the following subsections.

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 total 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 $G2$). If we have a graph, say TG , that represents all possible (graphical) types that are needed to describe a system, a total homomorphism h from any graph G to TG would associate a (graphical) type to each

Use Case Specification (original)		
Number	1	
Name	Log into <i>ATM</i>	
Summary	User logs into <i>ATM</i>	
Priority	5	
Preconditions	User has <i>bank card</i> and registered <i>password</i>	
Postconditions	User receives <i>menu</i> of available <i>ATM</i> operations	
Primary Actor(s)	Bank <i>Customer</i>	
Secondary Actor(s)	Customer <i>Accounts Database</i>	
Trigger	Only option on <i>ATM</i>	
Main Scenario	Step	Action
	1	<i>System</i> asks for a <i>Bank card</i>
	2	<i>User</i> inserts <i>card</i>
	3	<i>System</i> asks for <i>password</i>
	4	<i>User</i> enters <i>password</i>
	5	<i>System</i> validates user's <i>card</i> and <i>password</i> and display <i>menu</i> of operations
Extensions	Step	Branching Action
	5a	<i>System</i> notifies <i>user</i> that <i>password</i> is invalid
	5b	<i>System</i> exits option
Open Issues		

Fig. 1. Login Use Case description.

item of G . We call this triple $\langle G, h, TG \rangle$ a *typed graph*, and TG is called a *type graph* (that is, nodes of TG describe all possible types of nodes of a system, and edges of TG describe possible relationships between these types).

A *Graph Rule* describes a relationship between two graphs. It consists of: a *left-hand side (LHS)*, which describes items that must be present for this rule to be applied; a *right-hand side (RHS)*, describing items that will be present after the application of the rule; and a *mapping from LHS to RHS*, which describes items that will be preserved by the application of the rule. This mapping must be compatible with the structure of the graphs (i.e., a morphism between typed graphs) and may be partial. Items that are in the LHS and are not mapped to the RHS are *deleted*, whereas items that are in the RHS and are not in the image of the mapping from the LHS are *created*. We also assume that rules do not merge items, that is, they are injective.

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. The application of a rule r to a graph G is possible if an image of the LHS of r is found in G (that is, there is a total typed-graph morphism from the LHS of

r to G). The result of a rule application deletes from G all items that are not mapped in r and adds the ones created by r .

Our analysis of GTs is based on concurrent rules and critical pairs, two methods of analysis independent from the initial state of the system and, thus, they are complementary to any other verification strategy based on initial states (such as testing), detailed further ahead.

2.2 UC Formalization and Verification Strategy

Figure 2 depicts the proposed UC formalization and verification strategy, which is divided into four main phases. Starting from a textual description of the UC, the first phase (*UC Data Extraction phase*) is to identify entities (Step 1) and actions (Step 2) that will be part of the formal model. Then, basic verifications can be performed regarding the consistency of the extracted information (*Primary Verifications phase*). We look for inconsistencies that might affect or even prevent the construction of the GT model such as entities or conditions that are mentioned but never used, actions or effects of an action that are not clearly defined, and so on. If inconsistencies are detected, the UC must be rewritten to eliminate them or the analyst can annotate the problem as an open issue to be resolved later on. When no basic inconsistencies are found, the GT can then be generated (*GT Generation phase*). In this process, conditions and effects of actions are modeled as states (graphs) in Step 3. Then, in Step 4, a type graph is built through the definition of a graphical representation of the artifacts generated in Steps 1 and 3. After that (Step 5), each UC step is modeled as a transition rule from one state (graph) to another, using the structures defined in Steps 3 and 4.

Having the GT, a series of automatic verifications (based on concurrent rules, conflict analysis, and dependency analysis) can be performed to detect possible problems (*UC Analysis phase*). We use the AGG tool [18] to perform the automatic analyses on the GT model. All detected issues are annotated as *open issues (OIs)* along with the solutions (when applicable). With this approach, any design decision made over an OI can be documented and tracked back to the original UC. Through analysis, it is possible to verify whether the pre- and post-conditions were correctly included in the model, whether there are conflicting and/or dependent rules, what is the semantics of a detected conflict or dependency, and whether these results were expected or not. One important point is that, during the process of representing the UC in the formal model, clarifications and decisions about the semantics of the textual description must be made. Annotated OIs force the stakeholders to be more precise and explicit about tacit knowledge and unexpressed assumptions about system invariants and expected behavior.

Open issues are classified according to their severity level: code Yellow (⚠️) indicates a warning, meaning a minor problem that can probably be solved by a single person; code Orange (🟡) indicates a problem that requires more attention and probably a definition/confirmation from the stakeholders; code Red (🛑) indicates a serious issue that requires a modification in the UC description.

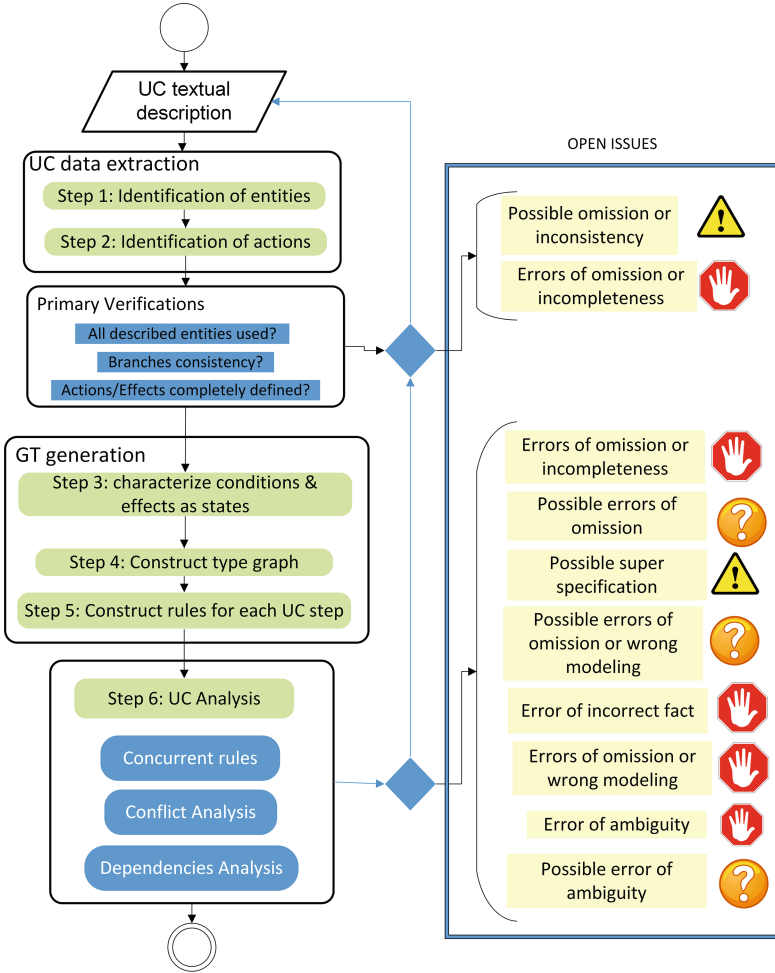


Fig. 2. Overview of the UC formalization and verification strategy.

UC1 Step 1	Artifacts	<i>List of Entities: User, ATM, System, Bank card, Password</i>
---------------	-----------	---

Fig. 3. UC1 - Step 1

Below, we describe the steps of our UC formalization and verification approach and the possible OIs that can be derived from them, illustrating the results for the login UC (UC1 - Fig. 1).

Step 1 - Identification of entities: The analyst manually identifies in the UC text all the entities involved in UC. Figure 3 shows the result for the example.

Table 1. Primary verification steps

Open issue	Verification	Problem	Severity level	Possible action
OI.1	An entity listed in Step 1 is not used (as actor or involved) in any action	Different names for the same entity or entities used in pre-/post-conditions are not used in the steps of the UC	Yellow	Analyze whether this is actually what is intended,
OI.2	A branching condition is not used in any action	The description of the actions may be too abstract	Yellow	Analyze whether this is actually what is intended
OI.3	The effect of an action is not clearly defined	Ambiguous description or omission	Red	Provide more details in the UC description

UC 1 Step 2	Artifacts	<i>Table of Actions UC1</i>			
		Action	Actor Involved	Conditions	Effect
		askCard	System IO	—	Display msg asking card
		insertCard	User System, Card	1.System asks for card 2.User has card	Card becomes connected to system
	...				
	Open Issues	<i>Table of Branch Conditions UC1</i>			
Step Condition			Value: Step		
5		User's card and password are validated	true: 5. false: 5a		

ATM never appeared in the actions table (OI.1)
 "exits option" - step 5b - not clear (OI.3)
 Branching conditions was not used : nothing was said about how validation should be carried out. (OI.2)

Fig. 4. UC1 - Step 2

Step 2 - Identification of actions: This step defines a *Table of Actions*, containing an entry for each action in the UC.

Actions that perform input/output operations involve a special entity called *IO*. Considering the possibility of alternative paths described in the UC, a *Table of Branch Conditions* is also defined.

Based on these two tables, three basic verifications can be performed and may raise open issues, as detailed in Table 1. Figure 4 shows the result of this step to the example UC. Only part of the Table of Actions is presented. As a result, three open issues were raised.

Step 3 - *Modeling conditions and effects as states*: In this step, it must be explicitly defined how to describe the conditions and effects listed in the *Table*

of *Actions*, as well as the pre- and post-conditions of the UC in terms of nodes and edges of a graph. The resulting table is called *Table of Conditions/Effects*. At the same time, we build a *Table of Operations* that is used in these formal definitions, with two predefined operations *Input* and *Output*. The tables resulting from this step are illustrated in Fig. 5.



UC 1 Step 3	Arti- facts	<i>(Part of) Table of Conditions Effects UC1</i>					
		Action	Condition/Effect	Characterization			
		pre	User has bank card and registered pass- word				
		insertCard.System askCard asking for card / Sys- tem asks for card	System				
...							
<i>Table of Operations UC1</i>							
OPN	Src	Tgt	RetVal	Pars	UsedIn		
Output	System	—	—	type: String	askCard, askPwd, validate&display, validate¬iy		
Input	—	System	—	type: String pwd: Password			

Fig. 5. UC1 - Step 3

Step 4 - Construction of the Type graph: The nodes of the type graph are the entities (**Step 1**) and operations (**Step 3**). The arcs are the relationships that were necessary to characterize the conditions/effects. If attributes of nodes were used to characterise the conditions/effects, they must also be part of the type graph. Figure 6 shows the type graph for our running example.

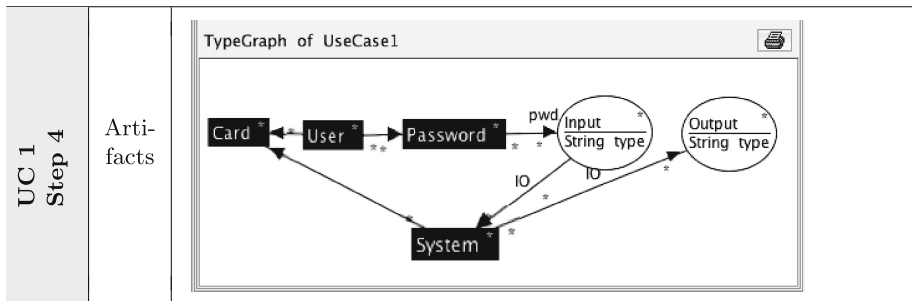


Fig. 6. UC1 - Step 4

Step 5 - Construction of rules: Rules that formally describe the behavior of the UC are constructed. For each action listed in the *Table of Actions*, we build

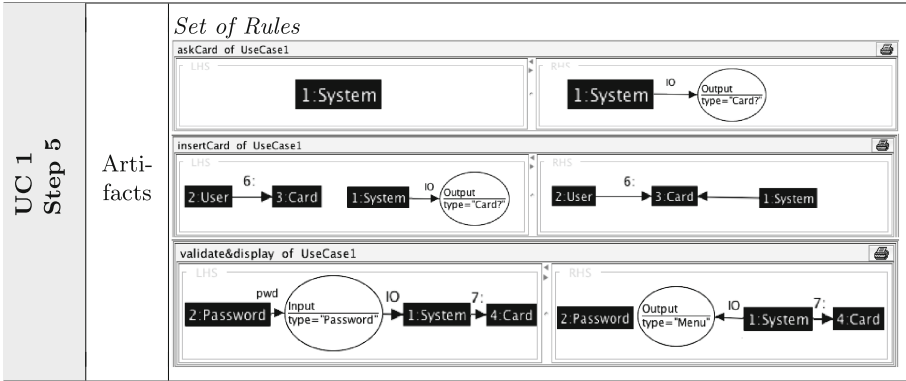


Fig. 7. UC1 - Step 5

a rule having as left-hand side (LHS) the graph that describes the conditions that must be true for this action to occur. The graphs corresponding to each condition are already described in the *Table of Conditions/Effects*, hence it is only necessary to merge them appropriately. Analogously, the right-hand side (RHS) of the rule is built using the effects of each action. Some rules of the example UC are shown in Fig. 7.







Step 6 - Use case analysis: The following analysis techniques may be performed in any order. The result from these analyses are usually complementary.

6.1. Use Case Effect: We first define *rule sequences (RSs)* that represent the execution of each possible path of the UC. RSs are just sequences of rules (defined in **Step 5**) that implement the execution of each scenario of the UC. Based on each RS, we build a single rule, called *concurrent rule*, which shows the effect of the whole UC in one step. This concurrent rule allows us to check whether the overall effect is really the desired one. Table 2 presents the analysis performed on the RSs and possible resulting open issues.

This analysis makes it explicit: (i) everything that is required for the UC to execute (LHS of the rule); and, (ii) the overall effect of the UC (RHS of the rule). To build the concurrent rule, the rules of the UC are joint by dependencies and, therefore, if some items are forgotten, this might lead to the impossibility of building the concurrent rule using all rules of the UC (and, thus, we might discover errors in the description of the UC steps as rules). Figure 8 shows the result of this step for our UC example.

6.2. Conflict Analysis (critical pairs): This type of analysis technique tells us which steps are mutually exclusive, that is, it pinpoints the choice points of the system. Table 3 presents the verifications based on critical pairs analysis and the possible resulting open issues. The result of the conflict critical-pair analysis is a *Conflict Matrix*, having rules as rows and columns, where each cell is filled with a number indicating how many items of a rule (row) are in conflict with items of another rule (column).

Table 2. Verification steps on rules sequences

Open issue	Verification	Problem	Severity level	Possible action
OI.4	A concurrent rule (for any alternative path in the UC) cannot be built using all the rules in the corresponding RSs	Items generated by some rule and used by another one may be missing by omission or modeling error	 Red	Review the rules
OI.5	Multiple concurrent rules are built for a single UC scenario	Multiple instances of one or more entities are possible, leading to different (possibly unexpected) ways of combining the rules of the UC	 Red	Check dependencies between rules to find unexpected sub-paths in the UC behavior
OI.6	UC pre-conditions are not a subgraph of the LHSs of the concurrent rules	Pre-conditions may include unnecessary items	 Yellow	Remove unused pre-conditions from the UC text
OI.7	The LHS of a concurrent rule is not a subgraph of the UC pre-conditions	UC requires something that is not explicitly stated in the pre-conditions	 Orange	Identify the RS in problematic concurrent rule and check whether all actions in this path were correctly modeled. If model is correct, check for missing pre-conditions.
OI.8	Post-conditions of an alternative path of the UC are not contained in the RHS of the corresponding concurrent rule	Some rule is not generating a required item (by UC omission or modeling mistake)	 Red	Check the rules. If all rules seem to be correct, post-conditions might be too strong.
OI.9	The RHS of a concurrent rule is not contained in the corresponding UC post-condition	Some rule is not deleting a required item (by UC omission or modeling mistake)	 Red	If the rules seem to correctly describe each action, post-conditions might be too weak

A value Zero in a cell means there is no conflict between two rules. The conflicts are only between items of the LHSs of the rules. The results of this step for our example are presented in Fig. 9.

6.3. Dependency Analysis (critical pairs): Similarly to critical pair analysis, (potential) dependency analysis is independent of an initial situation and is performed by building a *Dependency Matrix*. It shows relationships between rules and can be used to check whether the dependencies that we intuitively expected to occur are actually there. The verifications based on this matrix are presented in Table 4.

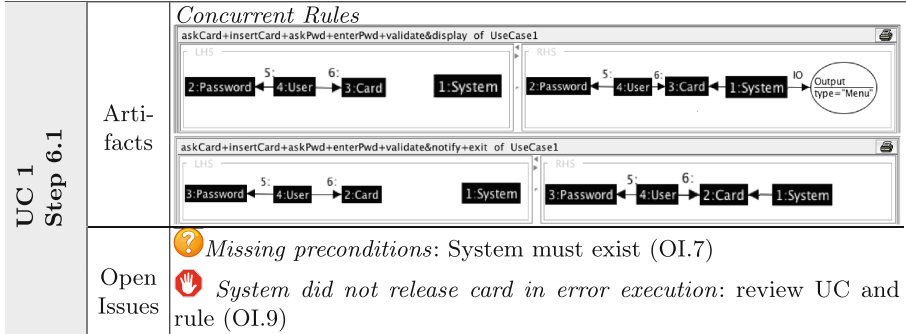


Fig. 8. UC1 - Step 6.1

Table 3. Verification steps based on critical pairs analysis

Open issue	Verification	Problem	Severity level	Possible action
OI.10	A rule is not conflicting with itself	The rule could be applied an arbitrary number of times	🟡 Yellow	Analyze whether this is actually the intended behavior
OI.11	There is no conflict between rules that represent the branching points of the UC behavior	Non-deterministic behavior: any alternative path can be taken no matter the condition	🚫 Red	Revise the conditions (LHSs) associated with rules representing alternative paths in the UC
OI.12	Conflicts between rules other than the ones described above (with itself and branch points)	These conflicts represent branches in system execution that must be explicitly stated in the UC (and in the model) as an alternative path	🟡 Orange	Revise the conflicting rules

Table 4. Verification steps based on dependencies analysis

Open issue	Verification	Problem	Severity level	Possible action
OI.13	Dependencies listed do not represent dependencies that are desired in the system	Possible omission in the UC description or a modeling error	🟡 Yellow	Check the RHS of a rule and the LHS of the other rule that depends on the first one
OI.14	An expected dependency between rules does not appear	Possible omission in the UC description or a modeling error.	🟡 Yellow	Check the rules involved

UC 1 Step 6.2	Arti- facts	<p style="text-align: center;"><i>Conflicts Matrix</i></p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>first \ second</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> <th>7</th> </tr> </thead> <tbody> <tr> <td>1 askCard</td> <td style="border: 2px solid black;">0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>2 insertCard</td> <td>0</td> <td>5</td> <td style="border: 2px solid black;">0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>3 askPwd</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table>	first \ second	1	2	3	4	5	6	7	1 askCard	0	0	0	0	0	0	0	2 insertCard	0	5	0	0	0	0	0	3 askPwd	0	0	0	0	0	0	0
	first \ second	1	2	3	4	5	6	7																										
1 askCard	0	0	0	0	0	0	0																											
2 insertCard	0	5	0	0	0	0	0																											
3 askPwd	0	0	0	0	0	0	0																											
Open Issues		<p>⚠️ <i>No self-conflicts on rules askCard and askPwd: Add a status attribute to prevent unexpected applications of these rules (OI.10)</i></p>																																

Fig. 9. UC1 - Step 6.2

UC 1 Step 6.3	Arti- facts	<p style="text-align: center;"><i>Dependenciess Matrix</i></p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>first \ second</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> <th>7</th> </tr> </thead> <tbody> <tr> <td>1 askCard</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>7 exit</td> <td style="border: 2px solid black;">0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> </tbody> </table>	first \ second	1	2	3	4	5	6	7	1 askCard	0	1	0	0	0	0	0	7 exit	0	0	0	0	0	0	0
	first \ second	1	2	3	4	5	6	7																		
1 askCard	0	1	0	0	0	0	0																			
7 exit	0	0	0	0	0	0	0																			
Open Issues		<p>⚠️ <i>askCard does not depend on exit: Rule exit must re-establish the initial conditions (changing the status variable accordingly) (OI.14)</i></p>																								

Fig. 10. UC1 - Step 6.3

Figure 10 shows the result of this step for our example UC. If two rules that we would like to occur always in some specific order are shown to be independent, then they could actually occur in any order, which represents a possible problem.

As a final result, we obtain a UC textual description more accurate and complete, as shown in Fig. 11, after all open issues have been analysed.

Currently, most of the steps are manual and some of them are carried out aided by tools. In steps 1 and 2, the analysis is purely manual, however, in steps 3, 4 and 5, the analyst should use a tool such as AGG, which helps build a formal model of the system by supporting the visual construction of graph grammars. This tool is also very useful in step 6 to perform the analysis of critical pairs and the generation of concurrent rules, which are not trivial processes.

3 Study Settings

Considering the methodology presented in the previous section, we now detail the study conducted to evaluate its usefulness for UC formalization and its effectiveness for tool-supported analysis of UCs in order to detect real and potential problems. This empirical study was crucial to obtain concrete evidences that using GTs in the context of UCs promotes quality.

In order to adequately evaluate our approach, we followed the principles of experimental software engineering [19]. We first present our study goal in

Use Case Specification (after verification)

Number	1	
Name	Log into <u>System</u> via ATM	
Summary	User logs into <u>System</u> via ATM	
Priority	5	
Preconditions	User has bank card and registered password <u>and the system is running</u>	
Postconditions	User receives menu of available <u>System</u> operations	
Primary Actor(s)	Bank Customer	
Secondary Actor(s)	Customer Accounts Database	
Trigger	Only option on ATM	
Main Scenario	Step	Action
	1	System asks for a Bank card
	2	User inserts card
	3	System asks for password
	4	User enters password
	5	System validates user's card and password and display menu of operations
Extensions	Step	Branching Action
	5a	System notifies user that password is invalid
	5b	System exits option <u>and goes back to step 1. System releases the card.</u>
Open Issues		

Fig. 11. Login Use Case description after verification.

Table 5. Goal definition.

Element	Our study goal
Motivation	To understand the usefulness of GTs to improve the quality of UCs
Purpose	Evaluate
Object	The effectiveness of using GTs to identify problems in UCs
Perspective	From a perspective of the researcher
Scope	In the context of a single real software development project

Table 5, which follows the GQM template [2]. Based on the definition of our goal, we derived two research questions, which we aim to answer with our study.

RQ-1. Are system analysts able to detect problems in their own UC descriptions without additional support?

RQ-2. How effective is our GT-based approach in identifying problems in UCs?

In order to answer these research questions, we followed the steps detailed in Sect. 3.1, which describes the study procedure. In Sect. 3.2, we introduce the software development project that is the target system of our study.

3.1 Procedure

1 - Analysis of UCs by System Analyst. In order to answer **RQ-1**, we requested a system analyst responsible for the creation of the UC descriptions, to carefully revise them, and point out problems, such as ambiguity, imprecision, omission, incompleteness, and inconsistency. This analyst has more than three years of experience in software projects with varying lengths (from a few weeks to years) with documentation describing the entire architecture of the solution, including artifacts such as class and sequence diagrams and use cases. If the system analyst found any problem, then such problems should ideally be identified by our approach. However, there was no guarantee the system analyst would be able to identify all existing problems. Therefore, the system analyst was a “*sound but not complete*” oracle, i.e. all problems identified were real problems but not necessarily all of the problems that existed in the UCs would be detected.

2 - UC Formalization. Given a set of 5 UCs, we performed the steps detailed in Sect. 2 to formalize them using GTs and used the AGG tool to analyze them. As a result, we detected some OIs.

3 - Evaluation of Detected Open Issues. After identifying open issues using our GT-based approach, we evaluated whether they were real problems in the analyzed UCs. If a detected OI had been pointed out as a real problem by the system analyst in the first step of our procedure, then it was definitely a real problem. Otherwise, the system analyst was requested to analyze the OI and verify whether it was an actual problem that they were unable to identify during the manual inspection.

4 - Data Analysis. The previous steps of our procedure produced the following data: (i) a list of OIs identified by our approach; and (ii) a list of problems identified by the system analyst with or without the aid of our approach. Our aim is that our approach detects all and only real problems (i.e., all OIs are real problems and all real problems are identified as OIs). This can be seen as a *classification problem*, and thus the effectiveness of our approach can be measured using the metrics widely used in the context of information retrieval of *precision* and *recall* [13], whose formulas are shown below:

$$Precision = \frac{true\ positives}{true\ positives + false\ positives} \quad (1)$$

$$Recall = \frac{true\ positives}{true\ positives + false\ negatives} \quad (2)$$

where *true positives* are OIs that correspond to real problems; *false positives* are OIs that are not real problems; and *false negatives* are real problems not identified as OIs.

3.2 Target System

The UC descriptions we used in our study are part of the analysis documentation of an industrial software project. This project involves the development of a typical system to manage and sell products, and include functional requirements such as adding new products, changing product information, creating sale orders, and releasing products in stock. Because our study procedure involves the manual analysis of UC descriptions, we selected a subset of all available UCs, choosing those that are not trivial, involving basic and alternative flows. The selected UC descriptions were written in English and described actions performed by actors (e.g. user, system, database, etc.) to achieve a particular state of the system or perform a specific operation. The UC set used has an average of ten sequential steps and five alternative branches.

Based on the Use Case Points Method [4], the UCs used in the study were evaluated between *average* and *complex* because of the number of transactions (between 4 and 7 or more than 7) and the type of actors involved (many complex actors) in their descriptions. Figure 12 shows a fragment of a full textual description of a UC along with two examples of *OIs* found.

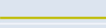

Number	1	
Name	Create <u>Product Item</u>  Same entity?	
Summary	An user creates a new item for sales	
	...	
Preconditions	User selects the option to create <u>sales items</u> on the system	
Post conditions	A new sales item is created	
	...	
Main Scenario	Step	Action
	1	System asks for a product name
	2	User inserts the product name
	3	System asks for a product quantity
	4	User inserts the product quantity
	...	
	11	System <u>validates</u> all input data from user  How to validate?

Fig. 12. Example of *Open Issues* found in the Use Case textual description.

In the example, the first *OI* identified is related to the names of the entities involved in the use case. The first step of the analysis is to identify all existing entities in the description, and in this situation, since there are three different names for an entity which is apparently the same, problems such as uncertainty, inaccuracy or ambiguity may appear in system development. This *OI*, classified as a warning, has not been confirmed as real problem because *Product Item* and *Sales Item* have different semantics in the system and are indeed treated as different entities. The other *OI* found is more serious, because there is an effect that is not fully defined in the use case. It is unclear what means the action of *validate* the data. Validation can be related to the existence of the data in the




database, or if the data entered are compatible with the expected type (numbers, words, dates, etc.), or check the correctness according to some oracle, etc. The definition of this effect possibly will require a project team’s decision, so it is rated at a higher level of severity. In this case, the problem was confirmed, since the analyst was not sure how this validation would be performed. These two situations exemplify the review process applied to the entire set of UCs used in this study. We do not provide any further details about our target system and its UCs due to a confidentiality agreement.

4 Results and Discussion

As result of the execution of the procedure described in the previous section, we collected the data needed to answer our research questions. The system analyst, after revising the original UCs, reported that they had no problems whatsoever. Hence, from the perspective of the analyst who created the UCs, they were correct. However, after applying our approach to these UCs, 32 OIs were identified across the 5 UCs, which gives an average of 6.4 OIs per UC. This is an expressive number, since the system analyst stated that the UCs did not contain any problem. In order to verify whether the identified OIs were false alarms (false positives), the system analyst was asked to check each one of them. Out of the 32 OIs, 24 were pointed out by the analyst as real problems and, consequently, only 8 of the identified OIs were false positives.

Table 6 presents our results in detail. It shows the number of OIs found in each UC (columns #OI) and how many of these OIs were confirmed as real problems (columns #P). The rows show the number of detected OIs with respect to their level of severity (yellow, orange, or red), according to our previously introduced classification. The table also presents the total number of detected OIs of each type and the total number of real problems considering all the 5 UCs.

Table 6. Study results

OI Type	UC 1		UC 2		UC 3		UC 4		UC 5		Total	
	#OI	#P	#OI	#P	#OI	#P	#OI	#P	#OI	#P	#OI	#P
	3	2	4	2	2	1	4	2	1	0	14	7
	1	1	1	1	1	1	0	0	2	2	5	5
	3	3	1	1	3	2	3	3	3	3	13	12
Total	7	6	6	4	6	4	7	5	6	5	32	24

Legend: UC - Use Case; OI - Open Issue; P - Problem.

The results were then analyzed according to the selected metrics. Because the system analyst was unable to identify any problem without support (i.e., to the best of the analyst’s knowledge there was no error in the analyzed UCs), the number of problems not identified by our approach was equals to 0, leading to *recall* = 1.0. However, this result is possibly misleading, as there might be

problems not identified by both the system analyst and our approach. This is an indication that even for a stakeholder (such as a system analyst, or a user) who knows well the system domain, it is not a trivial task to identify problems. A possible reason for this is the fact that this knowledge of the domain causes the stakeholder to understand different names as synonyms as part of this domain-specific knowledge and overlook omissions in the UCs because they believe some information is obvious. This is an evidence that support (e.g. techniques or tools) for the revision process plays a key role in identifying existing UC problems. As for the Precision, the obtained value was 0.75 (24 true positives and 8 false positives) — that is, 75 % of the OIs identified by our GT-based approach were real problems in the UC descriptions. Most of the identified issues were actual problems whereas most of our false alarms (7 of 8) are in the less critical categories. This means that most of the identified issues not only were real errors not detected by the person who created the UCs, but also revealed problems that required attention from the analyst as they could lead to serious consequences in the actual system.

By analyzing OIs not identified as problems, we observed that 6 of them were not necessarily classified as a false positive by the system analyst. They preferred to leave such issues as they were and postpone changes to future design decisions, considering that they alone could not decide what was the best approach to tackle those issues. The other 2 OIs found, confirmed as false positives by the system analyst, were related to words (names or concepts) used in the specification and have been identified as incompleteness or ambiguities due to lack of knowledge of the modeler about the problem domain and the internal processes of the company. Considering these results, it was concluded that, in most cases, our analysis helped the system analyst, even when an OI did not cause an immediate UC fix, but showed issues that might be considered in future phases of the project. These observations will be used as input for a refinement of the steps proposed to formalize UCs using GTs, in order to create a tool-assisted method to support the UC reviewing process.

Note that OIs were identified without the intervention of any stakeholder. The only provided input was the software documentation in the form of UC descriptions and the output was a checklist with OIs to be revised. For the system analyst, this has great value because the detected problems can be resolved not only at the UC level, but also at the design and implementation levels, as they are performed based on UCs. More importantly, had these problems been detected before the design and implementation, when they should have, development costs could have been potentially reduced.

5 Threats to Validity

When planning and conducting our study, we carefully considered validity concerns. This section discusses the main threats we identified to validate this study and how we mitigated them.

Internal Validity. The main threat to internal validity of this study was the selection of a person responsible for performing the modeling of UCs in the

formalism of graphs. Being a formalism mainly used in the sub-area of formal methods, it is difficult to find professionals working on software projects in industry with in-depth knowledge of graph transformations. However, one of our intentions with this work was to show that, correctly following the steps of our strategy, the modeler does not need a deep understanding of the formalism. Moreover, we used the AGG tool to automate the analyses of the generated model and provide a graphical interface for the manipulation of graphs.

Construct Validity. There are different ways of modeling a system through the formalism of graphs that can produce some threats to construct validity. The modeler may not follow correctly the modeling steps, being influenced by their prior knowledge about the formalism. This means that they could change the way of building the model based on their own previous knowledge. Consequently, we cannot guarantee they will obtain similar results to those presented in this work. The same applies when they have an advanced knowledge of the problem domain, because the modeler can insert information in the model that is not documented in the software artifact, hiding a possible omission of information in the UC description. For these reasons, we proposed a roadmap, step by step, on how to model UCs as GTs, for both beginners and experts users.

Conclusion Validity. As the main threat to validity of the conclusion of our study we also highlight potential problems in the generation of the formal model. Besides different forms of modeling and the issue that the modeler may be influenced by their experience or prior knowledge of the problem domain, the modeler may build a model inconsistent with the initial documentation due to errors during the modeling process. Once again, our step-by-step modeling process should be followed to prevent the creation of a model that is not consistent with the UC. Moreover, the tool-supported verifications can also detect some modeling errors, as shown in Tables 2, 3 and 4, thus reducing the risk of this threat.

External Validity. The main threat to the external validity was the selection of artifacts on which we based our study. We did not use any criteria to select either the project or the system analyst who participated of our study. As a consequence of this, the project that was made available for us may not be a representative sample of a large set of software development projects. We were aware of this threat during the study. However, we opted for randomly choosing artifacts to support the applicability of our strategy in different scenarios. This way, we guaranteed that we were not selecting UCs that would be more tailored for our approach. We also believe that obtaining good results even in a situation of a random choice of UCs gives greater confidence on our process.

6 Related Work

Some authors have developed approaches for translating UCs to well-known formalisms, such as LTS [16], Petri Nets [20], and FSM [9,17]. Unlike these formalisms, a GT model is data-driven, hence the focus is on the manipulation

of data inside the system. We do not need to explicitly determine the control flow unless it is necessary to guarantee data consistency. Considering other approaches that formalize UCs using a GT model, there are two closest to ours. The approach presented in [21] allows the simulation of the execution of the system but do not report the use of any type of analysis, which, in our opinion, reduces the advantage of having a formal model. The work described in [6] considers analyses such as critical pairs and dependencies involving multiple UCs and provides some ideas on the interpretation of the results. However, we propose a more structured way of providing diagnostic feedback about single UCs, which serves as a guide to point out the possible errors as well as their severity level. As problems in individual UCs can affect the inter-UCs behavior, we chose to initially study how to improve each UC and then move on to the study of how to apply similar ideas for inter-UC formalization and analysis.

Although we could find similar approaches regarding the formalization of UCs as GT models, we could not find any description of an empirical study as the one described in this paper. We believe that this type of study is important not only to provide confidence on the proposed approach, encouraging us to develop it further in terms of its formalization as a validation and verification process, but also to allow us to quantify how good it is. This type of result is also important from the stakeholders' point of view, as they can see in practice and numerically the benefits of applying formal methods to a usually informal process. Moreover, unlike most of the other approaches, our work focuses on helping the developers to construct the formal model by the definition of a systematic translation.

7 Conclusions and Future Work

In this paper, was investigated the suitability of GT as a formal basis for UC description and improvement. Was defined an outline of a translation process from UCs to GTs in a step-by-step manner, which describes how to use an existing tool to analyze the generated model and diagnose real and potential problems. The detection of such problems may cause changes to the UCs and trigger a new round of analyses, incrementally and iteratively improving the initial specification. The process also generates a formal model that can be used for further analyses. The approach was evaluated through an experiment with real software artifacts, where it was possible to detect existing errors, allowing an improvement on the original UCs.

Making a general analysis of the experiment, the results were considered promising, since it was possible to identify a large number of real problems based on a documentation that was produced at an early stage of a software development. Considering the proposed strategy, was observed the need for further automation of the process, which is the most immediate planned future work.

In this paper, was discussed the application of the approach to one UC at a time. Even though this has already shown benefits regarding the software development process, inter-UC analyses are currently being implemented as well

as the appropriate diagnostic feedback. Within the same model frame, other types of validation and verification techniques on GT models, such as test case generation, model checking, and theorem proving are also subject of current work. These techniques will be incorporated into a comprehensive methodology for software quality improvement targeting other types of errors. We also plan to study how changes in the UCs could be handled by our approach and how we could reduce the impact and cost of changes by identifying which parts are effectively affected and which analyses are actually required.

Also, a tool was designed, which is already in the early stages of development, in order to automate the first steps of the methodology (between steps 1 and 5) to help the analyst to build a formal model through the graph formalism. This tool aims to help produce the model data in a format acceptable by the AGG tool, responsible for the computational analysis of the graphs.

Finally, note that, although was not presented any new formal method or verification technique here, a considerable amount of expertise in formal methods was required to define the OIs: they are meant to bridge the gap between the informal and formal worlds. We believe that this type of work is crucial towards the industrial adoption of formal methods.

References

1. Alagar, V.S., Periyasamy, K.: Specification of Software Systems. Texts in Computer Science, 2nd edn. Springer, London (2011)
2. Basili, V., Caldiera, C., Rombach, H.: Goal question metric paradigm. In: Marciniak, J.J. (ed.) Encyclopedia of Software Engineering, vol. 1. Wiley, New York (1994)
3. Cockburn, A.: Writing Effective Use Cases, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
4. Diev, S.: Use cases modeling and software estimation: applying use case points. SIGSOFT Softw. Eng. Notes **31**(6), 1–4 (2006)
5. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Applications, languages, and tools, vol. 2. World Scientific, River Edge (1999)
6. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: Proceedings of the 24th ICSE, pp. 105–115 (2002)
7. Hurlbut, R.R.: A survey of approaches for describing and formalizing use cases. Technical report XPT-TR-97-03, Expertech, Ltd. (1997)
8. Jin, N., Yang, J.: An approach of inconsistency verification of use case in XML and the model of verification tool. In: Proceedings of MINES 2010, pp. 757–761 (2010)
9. Klimek, R., Szwed, P.: Formal analysis of use case diagrams. Comp. Sci. **11**, 115–131 (2010)
10. Köters, G., werner Six, H., Winter, M.: Validation and verification of use cases and class models. In: Proceedings of the 6th REFSQ (2001)
11. Myers, G., Sandler, C., Badgett, T.: The Art of Software Testing. ITPro Collection. Wiley, New York (2011)
12. Patton, R.: Software Testing, vol. 408, 2nd edn. Sams Publishing, Indianapolis (2005)

13. Powers, D.M.: Evaluation: from precision, recall and F-factor to ROC, informedness, markedness & correlation. Technical Report SIE-07-001, FUSA (2007)
14. Rozenberg, G. (ed.): Foundations, vol. 1. World Scientific, River Edge (1997)
15. Shen, W., Liu, S.: Formalization, testing and execution of a use case diagram. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 68–85. Springer, Heidelberg (2003)
16. Sinnig, D., Chalin, P., Khendek, F.: LTS semantics for use case models. In: Proceedings of the ACM SAC, pp. 365–370. ACM (2009)
17. Sinnig, D., Chalin, P., Khendek, F.: Use case and task models: an integrated development methodology and its formal foundation. *ACM ToSEM* **22** (2013)
18. Taentzer, G.: AGG: a tool environment for algebraic graph transformation. In: Münch, M., Nagl, M. (eds.) AGTIVE 1999. LNCS, vol. 1779, pp. 481–488. Springer, Heidelberg (2000)
19. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in Software Engineering. Springer, Berlin (2012)
20. Zhao, J., Duan, Z.: Verification of use case with petri nets in requirement analysis. In: Gervasi, O., Taniar, D., Murgante, B., Laganà, A., Mun, Y., Gavrilova, M.L. (eds.) ICCSA 2009, Part II. LNCS, vol. 5593, pp. 29–42. Springer, Heidelberg (2009)
21. Ziemann, P., Hävlscher, K., Gogolla, M.: From UML models to graph transformation systems. *ENTCS* **127**(4), 17–33 (2005)