Detection of Conflicting Functional Requirements in a Use Case-Driven Approach

A static analysis technique based on graph transformation

Jan Hendrik Hausmann
Dept. of Math. and Comp. Sci.
University of Paderborn
33095 Paderborn, Germany
corvette@upb.de

Reiko Heckel
Dept. of Math. and Comp. Sci.
University of Paderborn
33095 Paderborn, Germany
reiko@upb.de

Gabi Taentzer*
Dept. of Math. and Comp. Sci.
University of Paderborn
33095 Paderborn, Germany
gabi@upb.de

ABSTRACT

In object-oriented software development, requirements of different stakeholders are often manifested in use case models which complement the static domain model by dynamic and functional requirements. In the course of development, these requirements are analyzed and integrated to produce a consistent overall requirements specification. Iterations of the model may be triggered by conflicts between requirements of different parties.

However, due to the diversity, incompleteness, and informal nature, in particular of functional and dynamic requirements, such conflicts are difficult to find. Formal approaches to requirements engineering, often based on logic, attack these problems, but require highly specialized experts to write and reason about such specifications.

In this paper, we propose a formal interpretation of use case models consisting of UML use case, activity, and collaboration diagrams. The formalization, which is based on concepts from the theory of graph transformation, allows to make precise the notions of conflict and dependency between functional requirements expressed by different use cases. Then, use case models can be statically analyzed, and conflicts or dependencies detected by the analysis can be communicated to the modeler by annotating the model.

An implementation of the static analysis within a graph transformation tool is presented.

Keywords

requirements specification, use cases, UML, unified process, graph transformation

1. INTRODUCTION

The development of software consists in a repetition of analysis and synthesis activities. Following the *separation of concerns* principle a complex problem or model is decomposed into different aspects or views, which are refined separately and integrated again. While most engineers seem well trained to the decomposition task, the re-integration of partial models still presents great challenges. A well-known instance of this problem is the integration of scenario descriptions in terms of UML sequence diagrams or message sequence charts into statecharts specifying the behavior of individual classes or components [6, 15]. Another instance, which shall be the focus of this paper, is the integration and consistency of requirements models expressing the views of different users, like clerks and customers, or different aspects of the problem, like static and dynamic requirements. Similar problems arise when separately evolved models shall be re-merged [25].

Requirements engineering is the process of gathering and structuring information both on the problem domain and on expectations toward the new (or improved) system. This activity is relevant for the construction of complex software systems which cannot be handled by small, highly interactive teams of programmers, but require large groups of developers specialized in different roles. In particular, several analysts will be busy capturing requirements of different stakeholders, resulting in a set of overlapping and partly conflicting requirements models. Then, these partial models have to be integrated toward a single consistent requirements document. This document is extremely important as it provides the basis for all relevant development decisions. In fact, the detection of requirements errors late in the development process causes very expensive re-iterations through all phases [2, 30].

In object-oriented software development, the UML [27] has become the standard notation for software models at different levels, including the requirements specification. In particular, class diagrams are used to capture static requirements and use cases are employed for dynamic and functional requirements. For this paper we will align (but not restrict) our terminology and argument to the UML-based Unified Process [19], although our approach can be combined with other use case-driven development processes, too.

The result of the *requirements capture* workflow in the Unified Process consists of a single domain model (a class diagram) and a collection of use cases. To build this com-

*On leave from the Technical University of Berlin, Germany.

ICSE'02, May 19-25, 2002, Orlando, Florida, USA.
Copyright 2002 ACM 1-58113-472-X/02/0005...$5.00.

mon model, all analysts have to compare and integrate their separately developed partial models. For the synthesis of a consistent domain model, describing what are the relevant concepts of the problem domain, we may rely on techniques and tools developed for database schema integration based on the entity-relationship model [29]. They provide means to detect, for example, homonyms and synonyms, and to resolve structural conflicts by refactoring of models. Use cases represent both dynamic and functional requirements. The dynamic aspect—when something should be done—is captured by sequences of actions of the system and the user which interact to fulfill a certain user-goal. They can be modeled by sequence or activity diagrams. The functional aspect—how it should be done—is described by pre- and post-conditions of actions in natural language. In particular, the functional aspect is not formally integrated with the static domain model. Thus, intended connections between static structures and activities can only be indicated by giving meaningful names to use cases or activities. As a consequence, no tool support can be provided for detection and elimination of conflicts, which relies entirely on the intuition of experienced modelers. This applies to two kinds of consistency problems.

Consistency of aspects: Dynamic and functional requirements expressed by use cases and their annotations may refer to terms from the problem domain that are not captured in the static domain model, or that have been renamed or redefined in the integration of the static model. The intended effect of executing a use case may violate constraints of the static model.

Consistency of views: Semantic overlap may exist between use cases expressing requirements of different stakeholders. This may be intended if interaction is required to perform a common task, but it may also be a consequence of conflicting interests of different parties in the real world, or of undocumented dependencies between different use cases. We distinguish *conflicts*, where the execution of one activity may prevent the execution of another one, and *dependencies*, where the execution of one activity may require the prior execution of another activity.

Technically, we define aspects at the language level (e.g., statechart diagrams specify the dynamic aspect) while views are more generally related to users' concerns. Many authors speak in both cases of *views* [26, 13].

Both kinds of inconsistencies may lead to severe misdevelopments which are only detected much later in the process. In fact, since the workflow of *requirements capture* is followed by the decomposition activities of the *analysis* workflow, unresolved consistency problems tend to persist until the next big synthesis step: the *design*. It is thus advisable to detect and eliminate (or at least manage [26]) inconsistencies from the requirements model before progressing further in the development of the system.

In this paper, we approach the first problem by proposing a diagrammatic specification of pre- and post-conditions of actions by UML collaborations which are formally interpreted as graph transformation rules. Based on concepts from the theory of graph transformation, we approach the second problem by formalizing the intuitive notions of conflict and dependency of use cases with the aim of providing an analysis technique and tool support. In this way, we combine the technical benefits of formal requirements specifications using, e.g., a logic-based language, with the intuitive usability of a visual technique. In fact, the logic-based approaches discussed in [23] are mainly targeted at the area of safety-critical systems, where the costs of employing highly-specialized experts for creating and verifying formal specifications are justified. In the context of business applications, however, it is more important that models can be communicated to both domain experts and developers. Therefore, our aim is not formal verification of consistency, but detection of *potential consistency problems.* We shall demonstrate by means of an example how the results of our analysis can be used to annotate use cases and activity diagrams and to trigger a re-iteration of the requirements model to eliminate the undesired effects.

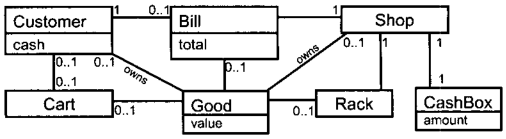


Figure 1: Class diagram of the shop

The paper is organized as follows: Section 2 introduces our running example and motivates the use of UML collaborations as functional specifications in UML use case models. Section 3 gives a formal interpretation of these models by means of graph transformation, which is used in Section 4 to formalize conflicts and dependencies. Analysis method and tool support are introduced in Section 5, while Section 6 is dedicated to the application and interpretation of the analysis results in terms of our running example. The concluding Section 7 summarizes our results and points out ideas for further work on this topic.

2. INTEGRATED MODELING OF STATIC AND DYNAMIC REQUIREMENTS

Object-oriented requirements specifications represent both static requirements concerning the objects of the problem domain, and dynamic requirements concerning the intended workflows. They can be expressed, respectively, by UML class and activity diagrams. Use case diagrams are employed to identify actors and system boundaries. Thereby, they structure the overall workflow into clusters of activities corresponding to the actor's goals, while abstracting from the actual subtasks necessary to reach these goals. Collaborations are used to model pre- and post conditions of actions in activity diagrams thus providing an integration of static and dynamic aspects.

Static model. Employing the UML, static requirements are specified by a class diagram. The class diagram in Figure 1 represents part of the business model of a shop which will be used as a running example to illustrate our approach. The shop provides racks carrying goods and shopping carts for the customers. Customers hold a certain amount of cash, as does the cash box of the shop. Bills list the goods collected by the customers together with the overall total of the prizes.

Since we are at the level of requirement specification,

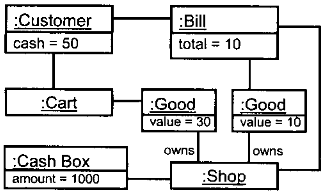


Figure 2: An instance of the class diagram in Figure 1 representing a snapshot of the shop

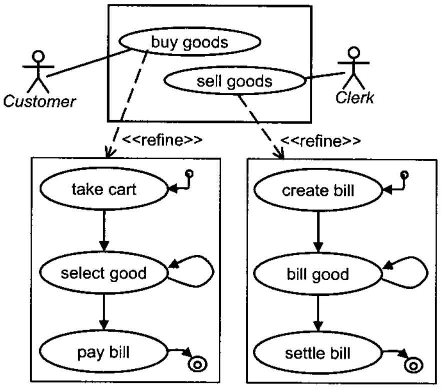


Figure 3: Use case diagram of the shop

classes do not have method signatures associated with them, i.e., the class diagram specifies only classes, associations, attributes, and constraints [19, 7]. An instance of this class diagram, as shown in Figure 2, represents a snapshot of our model.

Dynamic model. Dynamic requirements, like business processes, are described by means of activity diagrams (see Figure 3) consisting of action states (oval vertices) connected by transitions modeling the flow of control. The behavior of the shop is to allow customers to buy goods. Therefore they take a cart, select goods by placing them into the cart, and proceed toward the cash box. There, a clerk is waiting to sell the goods. An entry on the bill is created for each good, the goods are taken out of the cart and, with the settlement of the bill (the payment by the customer), the ownership of the goods is transferred from the shop to the customer. These facts have been captured in the use cases displayed in Figure 3. The dashed lines represent the annotation of each use case by a UML activity diagram.

Functional model. So far, the only link between static and dynamic requirements is given by the names of use cases and action states, like buy goods or take cart, which make reference to the classes in the class diagram. A more formal integration can be achieved by a description of the pre- and post-conditions of the actions and use cases. Such *functional requirements* are often specified in natural language as part of a use case model (see, e.g., [19]). Some approaches, like xUML [20], provide means for formal action specification using a high-level action notation. However, such formal notations, require familiarity with programming concepts. Thus, for discussing requirements with domain experts or users, a diagrammatic action specification is more suitable. Catalysis [7], for example, advocates the use of collaborations for this purpose. The idea goes back to the Fusion method [3] where actions are specified by snapshots of the object configuration before and after the operation.

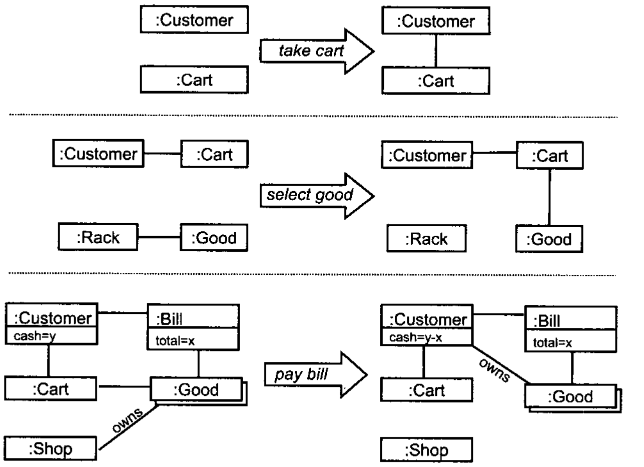


Figure 4: Action specifications for use case buy goods

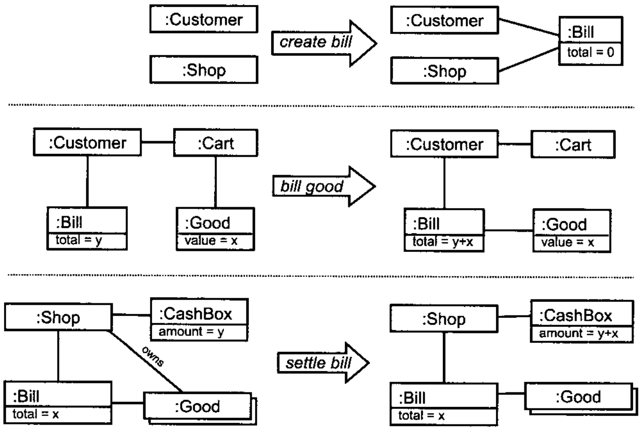


Figure 5: Action specifications for use case sell goods

Building on the latter approach, we propose a rule-based specification of pre- and post-conditions and effects of actions by means UML collaborations. In Section 3, this use of collaborations shall be formalized by means of graph transformation rules (cf. [21, 17]).

For each of the actions of the activity diagrams in Figure 3, their pre- and post-conditions are described as *collaboration rules*, i.e., pairs of collaborations representing transformations on object configurations (see Figures 4 and 5). The precondition of an action is satisfied in a given state (an instance of the class diagram) if the object pattern forming the left-hand side of the corresponding rule has an occurrence in this instance diagram. In this case, the action consists in replacing this occurrence by a copy of the right-hand side pattern. For example, the action bill good specified in Fig-

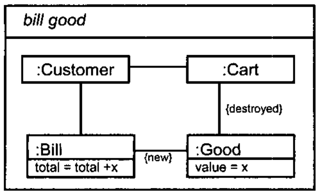


Figure 6: Condensed presentation of the collaboration rule for bill good

ure 5 is applicable if in the current object configuration there exist (instances of) Customer, Cart, Bill, and Good such that the Customer is associated with a Bill and a Cart containing a Good. As a result of the application, the Good is removed from the Cart and added to the Bill. Also, the total of the Bill is increased by the value of the Good (see also Figure 7).

In order to make our point, we have included two inconsistencies which shall later be formalized and detected by formal analysis. The first is between pay bill and settle bill. Both actions include the transfer of ownership of the goods, as described by the redirection of the owns links from the Shop to the Customer. This represents an overlap of responsibilities which requires further negotiations. Second, Customer and Clerk even seem to come from different continents: Use case buy goods uses European standards where customers have to collect their shoppings by themselves after paying the bill. Use case sell goods acts according to the American way where goods are packed into bags while they are entered on the bill.

Alternatively to the rule-based notation stressing the distinction between pre- and post-conditions we can adopt a more compact presentation by plain UML collaborations where pre- and post-conditions are jointly represented in one diagram. In order to distinguish those items that are deleted or newly created, constraints {destroyed} and {new} are used. For the collaboration rule bill good in Figure 5, the corresponding condensed presentation is shown in Figure 6. The decision, which representation is more appropriate in a given situation, depends on the amount of change between the pre- and the post-collaboration. In the sequel we stick to the rule-based notation.

Like the textual action notations in [20], collaboration rules for action specification provide an executable model which can visualize the behavior of a high-level requirements specification as a sequence of snapshots. In the following section, we shall formalize these notions and explain how such a sequence is actually produced.

3. TYPED GRAPH TRANSFORMATION AS SEMANTIC MODEL

In this section, some basic concepts and constructions from the theory of graph transformation are presented in order to formalize the requirements models discussed above.

Graphs as states. Graphs are often used as abstract representation of diagrams, e.g., in the UML meta model [27]. Formally, a *graph* consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V, respectively. Variations include hypergraphs, where edges can be attached to an arbitrary sequence of vertices, attributed graphs [24], whose vertices and edges are decorated with textual or numerical information, or more complex object-oriented or hierarchical graph models. The theory described below is largely independent of the notion of graph, which can be chosen to reflect as closely as possible the concepts of the modeling language. In fact, most of the notions and constructions can be (and have been) described at the level of *high-level replacement systems* [9], an axiomatization based on category theory of the so-called *double-pushout approach* to graph transformation [10], which can be instantiated to a variety of different graph models. In the following we deal with attributed graphs.

In object-oriented modeling graphs occur at two levels: the type level (given by the class diagrams) and the instance level (given by all valid object diagrams). This idea can be described more generally by the concept of *typed graphs* [4], where a fixed *type graph* TG serves as abstract representation of the class diagram. Its instances are graphs equipped with a structure-preserving mapping to the type graph, formally expressed as a *graph homomorphism*.

For example, the instance diagram in Figure 2 can be mapped to the class diagram in Figure 1 by defining $type(o) = C$ for each instance $o : C$ in the diagram. Extending this to links, preservation of structure means that, for example, a link between objects o_1 and o_2 must be mapped to an association in the class diagram between $type(o_1)$ and $type(o_2)$. By the same mechanism of structural compatibility we ensure that an attribute of an object is declared in the corresponding class, etc.

In order to formalize, in an abstract way, the notion of constraints (like upper and lower bounds for the multiplicity of associations), we assume for each type graph TG a class of constraints $Constr(TG)$ that could be imposed on its instances. A class diagram is thus represented by a type graph TG plus a set $C \subseteq Constr(TG)$ of constraints over TG. The class of instance graphs over TG is denoted by $Inst(TG)$ while we write $Inst(TG, C)$ for the subclass satisfying the constraints C. Thus, if (TG, C) represent a class diagram with multiplicity constraints as in Figure 1, an instance like in Figure 2 is an element of $Inst(TG, C)$ (see [11] for an elaboration of this concept).

In particular, we will be interested in negative constraints which enjoy the following monotonicity property: Given a graph G and a subgraph $G_0 \subseteq G$, if G satisfies a negative constraint c then so does G_0. That means, negative constraints, once violated in a configuration, cannot become valid again when the configuration is placed into bigger context. Typical examples include upper bound multiplicity constraints like 0..1, but also more complex properties like the absence of certain paths or cycles.

Rules and transformations. After having defined the valid object configurations as instances of a type graph satisfying the constraints, the idea of collaboration rules describing the evolution of such configurations is formalized in terms of graph transformation. A *graph transformation rule* $p : L \to R$ consists of a pair of TG-typed instance graphs L, R such that the union $L \cup R$ is defined. (This means that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.) The left-hand side

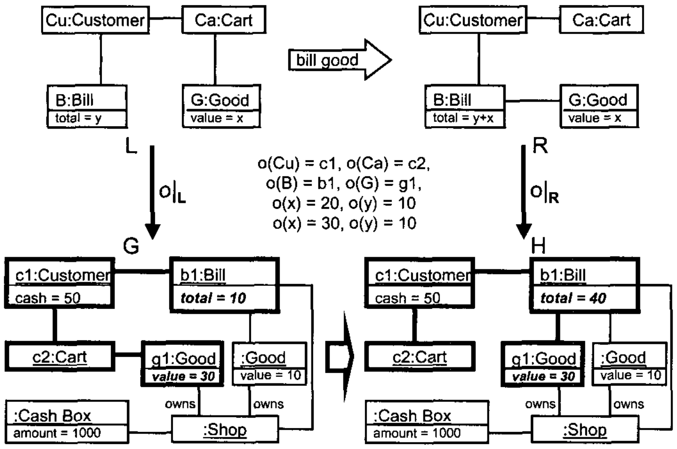


Figure 7: Application of the rule bill good

L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions. Usually, we omit the identities of objects and links in the collaboration rule, assuming that the intended intersection between a rule's left- and right-hand side is obvious from the layout.

A *graph transformation* from a pre-state G to a post-state H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and
- $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

Operationally, the application of a graph transformation rule like bill good in Figure 5 is performed in three steps. First, find an occurrence $o|_L$ of the left-hand side L in the current object graph G. Second, remove all the vertices and edges from G which are matched by $L \setminus R$. Make sure that the remaining structure $D := G \setminus o(L \setminus R)$ is still a legal graph, i.e., that no edges are left dangling because of the deletion of their source or target vertices. In this case, the *dangling condition* [10] is violated and the application of the rule is prohibited. Third, glue D with $R \setminus L$ to obtain the derived graph H. Figure 7 shows a sample application of the rule bill good. Its occurrence is given by the bold objects and links.

Altogether, the static and functional aspects of a model can be formally represented as a *typed graph transformation system* $\mathcal{G} = \langle TG, C, P, \pi \rangle$ consisting of a type graph TG, a set of constraints $C \subseteq Constr(TG)$, a set of rule (or action) names P, not necessarily finite, and mapping π associating with each rule name $p \in P$ a rule $\pi(p) = L \to R$ over TG. In this case, we write $p : L \to R \in \mathcal{G}$. Infinite sets of rules are necessary because collaboration rules with multi objects, like pay bill, represent rule schemes which expand to a countably infinite set of graph transformation rules, one for each legal multiplicity of the multi object. When applying these rules to a given graph, always the maximal rule is chosen among all applicable ones (cf. [32]).

A transformation $G \overset{p(o)}{\Longrightarrow} H$ in $\mathcal{G}$ is a transformation using a rule $p \in \mathcal{G}$ such that whenever G satisfies the constraints C expressed in the class diagram, so does the resulting graph H. This can be checked at runtime or verified statically [18]. It ensures that only consistent configurations are reachable from a consistent starting configuration. The *behavior of* $\mathcal{G}$ is given by the set of its transformation sequences $G_0 \overset{p_1(o_1)}{\Longrightarrow} \dots \overset{p_n(o_n)}{\Longrightarrow} G_n$, i.e., sequences of consecutive transformations in $\mathcal{G}$ starting from a consistent graph $G \in Inst(TG, C)$.

The dynamic model selects among the transformation sequences in $\mathcal{G}$ those which are compatible with the ordering of actions as specified in the activity diagrams. It is possible to encode this selection into the same formal model by "compiling" activity diagrams into sets of graph transformation rules. (This is similar to an encoding of finite automata into (string) grammars, where the nodes of an automaton are turned into non-terminals.) Since our focus is on the analysis of functional requirements, we do not elaborate such encoding but refer the interested reader to [22] where a similar construction is shown for statechart diagrams.

Use cases as views. A use case represents a view of the overall model corresponding to the requirements of a particular actor (or a group of actors). A view on a graph transformation system representing the complete model is defined by a subgraph of the type graph (modeling the relevant fragment of the class diagram) and a subset of the rules [12].[1]

In our example, the view corresponding to the use case by goods comprises the rules take cart, select good, and pay bill. The relevant fragment of the class diagram includes everything except for the CashBox class, its attribute and association with the shop. On the other hand, the use case sell goods consists of the rules create bill, bill good, and settle bill and all classes, associations and attributes, except for the cash attribute of class Customer.

One use case in isolation does not show meaningful behavior because it represents an incomplete view of the functionality from the perspective of a single actor. Thus, interaction is required between different use cases. For example, the execution of the action bill good of the clerk should depend on the previous execution of the action select good of the customer. One important integration problem is to fix these interactions. More dramatically, there may be conflicts between the use cases resulting from different opinions of the stakeholders about the intended behavior or the scope of their responsibility. The next section is devoted to the analysis of such conflicts and dependencies.

4. CONFLICTS AND DEPENDENCIES BETWEEN FUNCTIONAL REQUIREMENTS

Our analysis is based on the notion of independence of graph transformations which captures the idea that, in a given situation, two transformations are neither causally dependent nor in conflict. We distinguish parallel independence (absence of conflicts) and sequential independence

[1] This notion can be extended to include the possibility for renaming, extension, or refinement of types and rules. These issues, which are studied, for example, in [16, 14], are ignored here for simplicity.

$$\begin{tikzcd}
 & G \arrow[dl, "p_1(o_1)"'] \arrow[dr, "p_2(o_2)"] & \\
H_1 \arrow[dr, "p_2(o_2)"'] & & H_2 \arrow[dl, "p_1(o_1)"] \\
 & X &
\end{tikzcd}$$

Figure 8: Independence of transformation steps

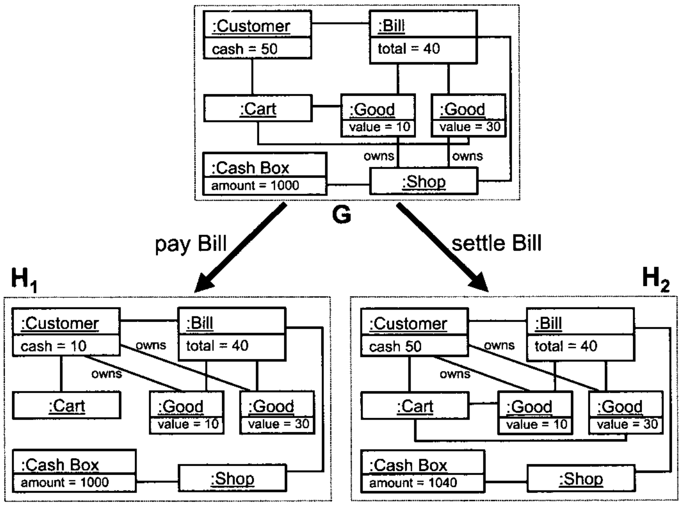


Figure 9: A conflict between pay bill and settle bill

(absence of causal dependencies). For both notions there exist a weak, asymmetric, and a strong, symmetric version (see, e.g., [5] for a recent survey).

Parallel Independence. Given two transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ and $G \overset{p_2(o_2)}{\Longrightarrow} H_2$ like in Figure 8, $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ is *(weakly parallel) independent* of $G \overset{p_2(o_2)}{\Longrightarrow} H_2$ if the occurrence $o_1(L_1)$ of the left-hand side of p_1 is preserved by the application of p_2. This is the case if $o_1(L_1) \cap o_2(L_2 \setminus R_2) = \emptyset$, that is, $o_1(L_1)$ does not overlap with objects that are deleted by p_2. If the two transformations are mutually independent, they can be applied in any order yielding the same result. In this case we speak of *parallel independence.* Otherwise, if one of two alternative transformations is not independent of the second, the second will disable the first. In this case, the two steps are *in conflict.*

An example of a conflict between transformations of pay bill and settle bill is given in Figure 9: Both transformations destroy the owns links between the goods and the shop. Thus, they overlap in items that are deleted. As a consequence, each of the two disables the other one, i.e., they cannot be part of the same transformation sequence. This is unfortunate because both transformations capture important aspects of the intended overall behavior. For example, pay bill updates the cash attribute of the Customer while settle bill does the same for the amount attribute of the Shop. We will see in the next section how these different sides of the same coin can be integrated.

Sequential Independence. Given a sequence of two transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ like in Figure 8, $H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ is *(weakly sequential) independent* of $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ if the occurrence $o_2(L_2)$ of the left-hand side of p_2 is already present before the application of p_2. This is the case, if $o_2(L_2) \cap o_1(R_1 \setminus L_1) = \emptyset$, that is, $o_2(L_2)$ does not overlap with objects that are created by p_1. Otherwise, if the second transformation is not independent of the first, the first *causes* the second.

If, moreover, p_2 does not delete objects that are needed for the application of p_1, that is, $o_1(L_1) \cap o_2(L_2 \setminus R_2) = \emptyset$, the two applications can be exchanged without affecting the overall result of the sequence. In this case, we say that the two steps are *sequentially independent.*

We observe for future reference that the notions of *parallel and sequential independence* as well as *conflict and dependency* are exchangeable. Due to the symmetry of both rules and transformations (which is clearly visible in the set-theoretic formulation in Section 3), for each rule $p : L \to R$ an inverse rule $p^{-1} : R \to L$ can be build so that every transformation $G \overset{p(o)}{\Longrightarrow} H$ has an "undo" $H \overset{p^{-1}(o)}{\Longrightarrow} G$. Using this, transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ are sequentially independent if and only if $H_1 \overset{p_1^{-1}(o_1)}{\Longrightarrow} G$ and $H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ are parallel independent. For example, the sequence pay bill^{-1}; settle bill is *not* sequentially independent because the first step creates two owns links which are consumed by the second.

The above conditions for parallel and sequential independence, resp. their negations, have to be checked for given graphs and occurrences, that is, at run-time. This could be done, for example, in a simulation environment for debugging purpose. The focus of this paper, however, is on static analysis of *potential* conflicts and dependencies, rather than on run-time analysis. Therefore, the above notions have to be lifted to the level of rules.

Potential conflicts and dependencies. For two given rules $p_1 : L_1 \to R_1$ and $p_2 : L_2 \to R_2$ we say that

- p_1 *may be disabled by* p_2 if there exist transformation steps $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ and $G \overset{p_2(o_2)}{\Longrightarrow} H_2$ like in Figure 8, such that $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ is *not* independent of $G \overset{p_2(o_2)}{\Longrightarrow} H_2$,
- p_1 *may cause* p_2 if there exist transformation steps $G \overset{p_1(o_1)}{\Longrightarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ like in Figure 8, such that $H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ is *not* independent of $G \overset{p_1(o_1)}{\Longrightarrow} H_1$.

Thus, a potential conflict or dependency is witnessed by a pair of transformations, either alternative or consecutive, which provide a counterexample to parallel or sequential independence, respectively.

The *essence* of such a counterexample is a pair of objects or links $\langle x_1, x_2 \rangle$ with $x_1 \in p_1$ and $x_2 \in p_2$ such that $o_1(x_1) = o_2(x_2)$ and

- $x_1 \in L_1$ and $x_2 \in L_2 \setminus R_2$, in this case $\langle x_1, x_2 \rangle$ represents a *conflict*, or
- $x_2 \in L_2$ and $x_1 \in R_1 \setminus L_1$, in this case $\langle x_1, x_2 \rangle$ represents a *dependency.*

In the next sections, we will show how potential conflicts and dependencies between two (sets of) rules can be detected by a tool and presented to the modeler.

5. STATIC ANALYSIS OF CONFLICTS AND DEPENDENCIES

Given two use cases, we are interested in potential conflicts and dependencies between their functional specifications. In technical terms this amounts to compute for two graph transformation systems $\mathcal{G}_1$ and $\mathcal{G}_2$ all pairs of rules of $p_1 \in \mathcal{G}_1$ and $p_2 \in G_2$ such that p_1 *may be disabled by* p_2, or p_1 *may cause* p_2, or vice versa (cf. Section 4). The results of the analysis shall be presented to the modeler via an annotation of the model. If further explanations are requested, minimal counterexamples can be provided.

The computation of conflicts and dependencies is based on the idea of critical pair analysis which is known from term rewriting. Usually, this technique is used to check whether a rewriting system has a functional behavior, i.e., if it is confluent. Critical pairs have been generalized to graph rewriting in [28]. They formalize the idea of a minimal example of a conflicting situation. From the set of all critical pairs we can extract the objects and links which cause conflicts or dependencies. In the following, we discuss the analysis technique and its implementation in the AGG tool.

Critical pair analysis. A critical pair is a pair of transformations

$$H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$$

which are in conflict, and such that graph G is minimal, i.e., a gluing $G = o_1(L_1) \cup o_2(L_2)$ of the left-hand sides of the rules p_1 and p_2. This ensures that the set of all critical pairs for two given rules p_1 and p_2 is finite. It can be computed by overlapping L_1 and L_2 in all possible ways, such that the intersection $o_1(L_1) \cap o_2(L_2) \subseteq G$ contains at least one item that is deleted by one of the rules and both rules are applicable to G at their respective occurrences.

The set of critical pairs represents precisely all *potential conflicts*, that is, there exists a critical pair like above if, and only if, p_1 *may disable* p_2 or, vice versa, p_2 *may disable* p_1. The (obvious) reason is that every pair of conflicting transformations contains a critical pair consisting of all links and objects that are matched by the rule's left-hand sides.

As we have mentioned in Section 3, a collaboration rule containing a multi object is interpreted as a rule schema yielding an infinite number of graph transformation rules. Apparently, this presents an obstacle to an exhaustive pairwise analysis. However, since rules resulting from the same schema differ only in the number of copies of the corresponding multi object, it is enough to consider the instance where the multi object is represented by one normal object. It can be shown that every critical pair using a rule with more copies can be reduced to one with just a single representative.

Consider, for example, the critical pair in Figure 9 between two instances of pay bill and settle bill each containing two copies of the multi object Good. The pair can be reduced by dropping, in all three graphs, e.g., the Good object with value = 30 together with its links. The resulting transformations still represent a critical pair because they still share one owns-link in G that is deleted.

By means of the duality between conflicts and dependencies noticed in Section 4, we can also use critical pair analysis to find all potential dependencies among rules. In fact, a rule p_1 *may cause* p_2 (or vice versa) if, and only if, there exists a critical pair

$$G \overset{p_1^{-1}(o_1)}{\Longleftarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$$

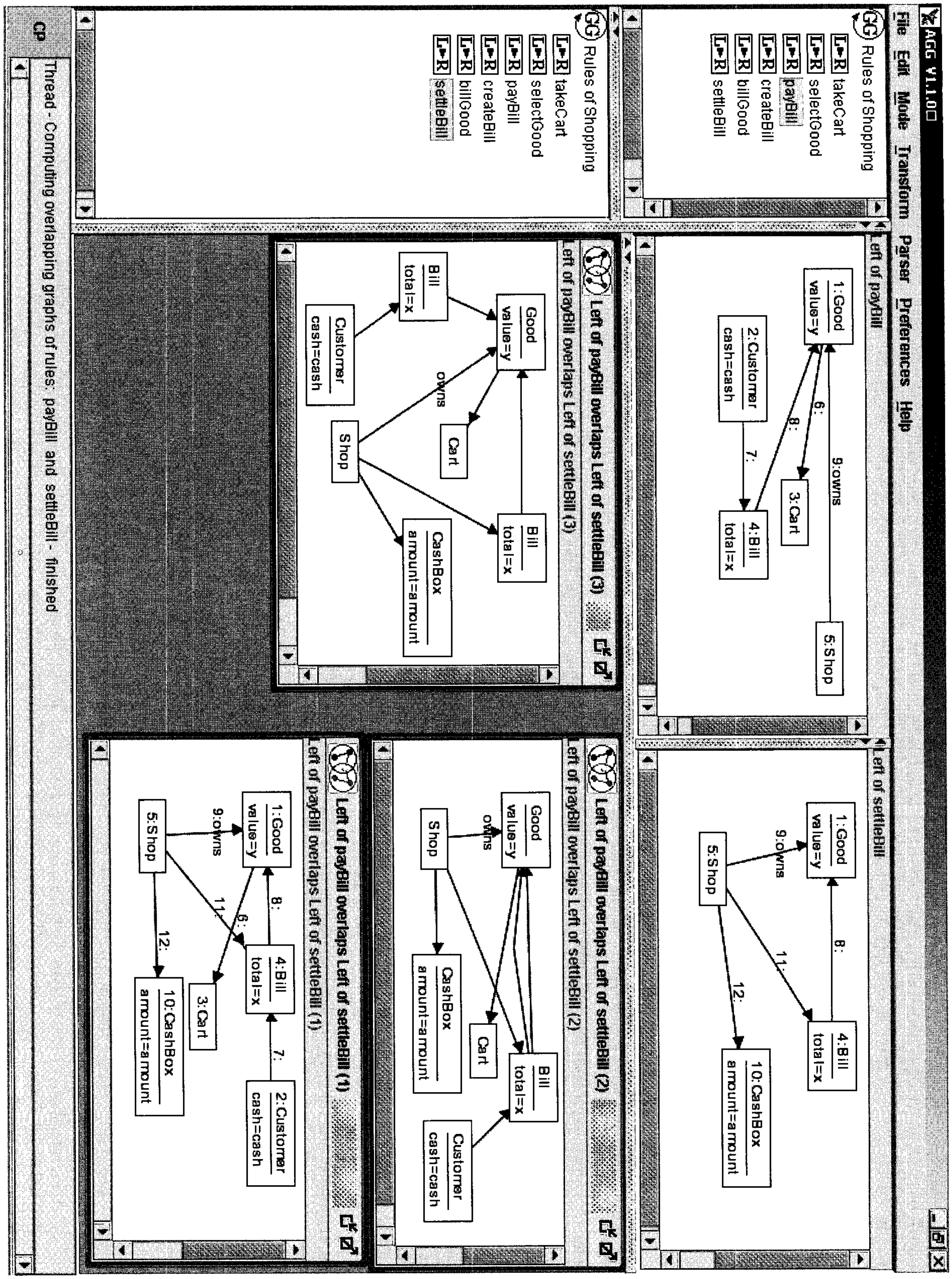
using the inverse of p_1. This follows from the discussion in Section 4 and the analogous statement for critical pairs and potential conflicts above.

Tool support. Critical pair analysis is implemented in the graph transformation engine AGG (see `http://tfs.cs.tu-berlin.de/agg`). The tool provides several graphical editors to create and manipulate graph transformation systems, an interpreter for executing the systems and animating the transformation process, and a debugger. Recently, an initiative has been started to implement static analysis techniques for graph transformation. The critical pair analysis is offered through a graphical user interface to browse through the computed pairs. In Figure 10, a screen dump of AGG shows all critical pairs of the rules pay bill and settle bill. The left-hand sides of both rules are depicted in the upper part, while three overlapping graphs are shown in the lower part. Note, that AGG is not an UML tool, i.e., although the graph representation looks similar, it does not strictly follow the UML notation. Nevertheless, presentational differences, like directed edges vs. undirected links (whose ends could be named in order to distinguish them) do not affect the results of the analysis.

For the overlapping graph in the lower right, the occurrences of the left-hand sides in this graph are indicated by numbers. Of the three critical pairs shown in Figure 10, only this one is actually relevant: The other two violate the negative constraint that each Good must have at most one link to a Bill (cf. the class diagram in Figure 1). As discussed in Section 3, such negative constraints, once violated, remain so under embedding into bigger context. Thus, all conflicting situations containing these critical pairs will also violate the multiplicity constraint.

To allow the exchange with other tools, graph transformation systems and critical pairs are stored in the XML-based Graph Transformation Exchange Language (GTXL) and Critical Pair Exchange (CPX) format, respectively. These formats are part of an initiative to integrate graph-based tools via common exchange formats for graphs (GXL) and graph transformation systems (GTXL) (see `http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html`).

To apply AGG for the static analysis of UML use case models, transformations between GTXL/CPX and the XML Metadata Interchange (XMI) format shall be provided based on the Extensible Stylesheet Language (XSL). Then, models defined in a UML CASE tool with XMI export can be transformed into graph transformation systems in GTXL format following the formalization in Section 3. These provide input for the AGG tool performing the critical pair analysis. Although the computed pairs can be inspected within the tool itself, it is more natural to visualize the analysis results directly on the original models. For this purpose, the computed pairs have to be exported in CPX format and further transformations have to be performed in order to annotate, e.g., use case diagrams with the conflicts and dependencies



Thread - Computing overlapping graphs of rules: payBill and settleBill - finished

Figure 10: Critical pair interface of AGG

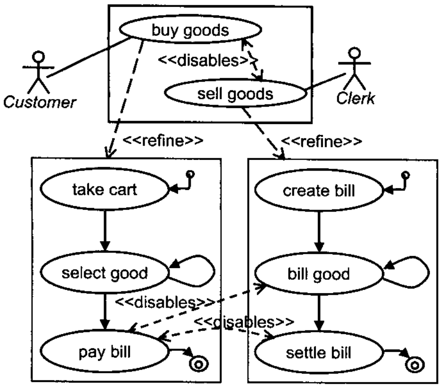


Figure 11: Conflicts between use cases buy goods and sell goods

shown in Figure 11 and 12.

The modeler can then browse these annotated models with the original CASE tool, changing the model, ignoring or deferring detected conflicts according to their priority and interpretation.For the time being the feasability of automatic conflict detection is shown. A smooth tool integration based on XML technology is under development.

The next section is devoted to the interpretation of the analysis results in the case of our shopping example.

6. APPLICATION TO THE EXAMPLE

In Section 2, when introducing the shopping example we have sketched two possible conflicts between the use cases buy goods and sell goods. Now, we discuss the results of the analysis in detail and consider possible iterations of the model.

Analysis. Figures 11 and 12 show the use case diagram of Figure 2 enriched by potential conflicts and causal dependencies between activities, as well as their abstractions at the level of use cases. The inter-use case conflict captures exactly the two conflicts we already noticed in Section 2 between bill good and pay bill as well as between pay bill and settle bill. As discussed in Section 5, in the case of pay bill vs. settle bill only one "real" critical pair exists, which is shown in the lower right graph of Figure 10. In a larger context the same overlapping is shown in the graph G of Figure 9. The essence of this critical pair is the owns-link between the Shop and the Good which is deleted by both transformations. This formalizes the expectations of Section 2.

Figure 12 shows the causal dependencies within and between the two use cases. The dependencies between activities inside each of the use cases follow largely the specified control flow, except of the iteration of selecting and billing goods. These activities are required to be performed sequentially although there are no causal dependencies. Thus, they could also be executed in parallel. Figure 12 also shows inter-use case dependencies. For example, the customer has to take a cart and to select goods before the goods are billed and the bill is settled. Moreover, all goods have to be billed before the bill is payed. Since no further dependencies have been found, this means that, e.g., a bill may be created even if the customer has not yet selected any good. Whether or not this is a mistake depends on the intention of the modeler, how the activities of the two use cases should be interleaved to perform the overall task.

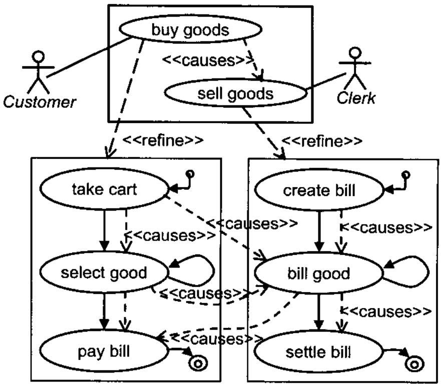


Figure 12: Causal dependencies between use cases buy goods and sell goods

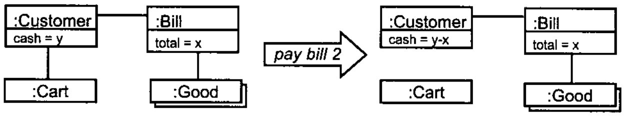


Figure 14: Revised version of pay bill

In order to understand better the results of the analysis, the modeler might be interested in the objects and links responsible for the causal dependencies. For this purpose, a combined presentation of control flow and data dependencies is helpful. Consider, for example, the use case diagram in Figure 13 where the causal dependencies between activities are depicted together with the essential objects or links where these dependencies manifest themselves. The notation is similar to Petri nets with read arcs where activities play the role of transitions while objects and links act as places. Thus, a dashed arrow from an activity like take cart to a link like the one between Customer and Cart represents the fact that this link is created by this activity. Symmetrically, an edge from a link or object to an activity indicates that the item is deleted. Read access is denoted by dashed lines without arrow heads. Thus, the customer first has to take a cart, producing the corresponding link, before goods may be selected or billed, where the link is required, etc. Like the annotations of the use case diagrams in Figure 11 and Figure 12, also these annotations can be extracted from the critical pairs produced by the analysis.

Interpretation. Surveying the results of the analysis, the modeler has to decide which dependencies or conflicts do actually represent errors or inconsistencies in the model. Due to the semi-formal and incomplete nature of use case models, this decision is based on the intention of the modeler and cannot be taken mechanically. Nevertheless, such a walk-through can give valuable hints for changing the model in the next iteration, or for documenting better the relevant decisions.

For example, the conflict between the use cases buy goods

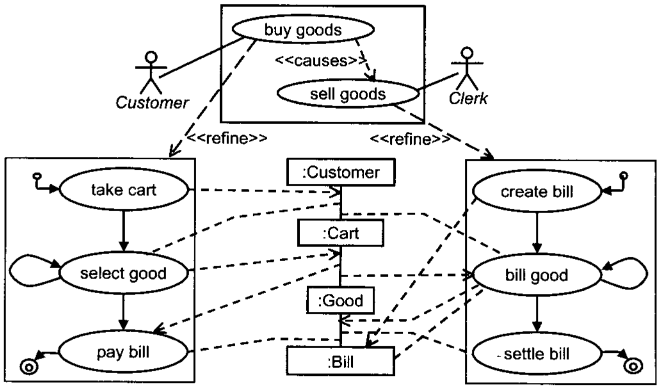


Figure 13: Causal dependencies between activates and essential objects and links

and sell goods contradicts the intuition that both use cases have to be performed in combination to achieve the desired effect. Having decided that there should be no conflicts between these two use cases, we have to correct this at the level of the associated activities pay bill and bill good as well as pay bill and settle bill, respectively. In our case, the conflicts can be resolved by assigning the responsibilities for the deletion of the link between Cart and Good and for the deletion of the owns link exclusively to the operation settle bill of the clerk. The revised rule pay bill2 is shown in Figure 14.

Note that not every conflict must represent an error. If the modeler decides that two use cases or activities are meant to happen alternatively, the conflict simply reflects this requirement at the object level. In this case, an absence of a conflict would indicate possible errors in the specification. In our example, the analysis revealed that the execution of pay bill does not disable select good. So a customer could be able to continue shopping even though he or she already paid. As this is regarded as unwanted behavior, we prevented this case by deleting the link between Customer and Cart in our improved rule (Figure 14). Thus, the activity of paying includes the returning of the cart.

Concerning the dependencies visualized in Figure 12, improvements of the model may be proposed whenever the dependencies defer from the control flow. While most of the dependencies follow the control flow specified in the activity diagram, three edges run between activities of different use cases. This indicates that the use cases may be interrelated, and it has to be re-considered if the separation chosen for the use case model fits the problem domain. On the other hand, two control flows (the two loops) are not accompanied by corresponding dependencies. This could either indicate that the activities may be performed concurrently (which even advanced buyers can only do to a certain degree) or that the specification could be enhanced by explicitly modeling the restrictions that lead to a rather sequential execution. It is thus possible to gain valuable hints for improving the model both from the presence and the absence of conflicts and dependencies.

7. CONCLUSION

In this paper, basic concepts from the theory of graph transformation have been used to specify and analyze the functional aspect of UML use case models. The interesting tension in this approach is between semi-formal and incomplete requirements on the one hand, and their formal analysis on the other hand. This leads to a trade-off between understandability and expressivity of modeling concepts which has led us to limit ourselves to basic transformation rules without sophisticated application conditions (which are also supported by the analysis technique implemented in the AGG tool).

For the same reason, we have resisted the temptation to propose extensions of use case diagrams to specify relations between use cases or with the underlying static model, as it is done, for example, in use case maps [1]. Instead, we use relations between use cases solely for visualization of analysis results.

Focusing on the functional aspect, our approach is complementary to other formalization of use case models. Stevens [31], for example, uses labeled transition systems to capture the essence of the dynamic aspect in terms of sequences of activities. The paper also mentions the issue of interference between different use cases but does not investigate it further.

More generally, our analysis approach complements existing model checking techniques which are also aimed at verifying dynamic properties of systems rather than properties of data transformations. It should be interesting to understand if these two views can be combined.

Further work at this topic will include investigation on the impact of use case relations and use case inheritance and the handling of inheritance in the underlying static model. An issue that has to be evaluated in forthcoming larger case studies is the necessary selection of useful information from the set of all detected conflicts. Moreover, model inherent information like structural constraints, has to be identified that decreases the number of detected conflicts considerably and leads to more efficient tool support, though.

8. REFERENCES

[1] D. Amyot and G. Mussbacher. On the extension of UML with use case maps concepts. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000, York, UK*, volume 1939 of *LNCS*. Springer, 2000.

[2] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] D. Coleman, P. Arnold, S. Bodof, C. Dollin,

H. Gilchrist, F. Hayes, and P. Jeremes. *Object Oriented Development, The Fusion Method.* Prentice Hall, 1994.

[4] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.

[5] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.

[6] W. Damm and D. Harel. Breathing life into message sequence charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. Formal Methods for Open Object Based Distributed Systems (FMOODS'99)*, Florence, Italy, 1999. Kluwer.

[7] D. D'Souza and A. Wills. *Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1998.

[8] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer-Verlag, 2000.

[9] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.

[10] H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[11] H. Ehrig and A. Tsioalikis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 Workshop on Graph Transformation Systems (GraTra), Berlin, Germany*, March 2000.

[12] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engeneering*, 7(4):457–477, 1997.

[13] P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures. In *Proc. ESEC/FSE'99*, volume 1687 of *LNCS*, 1999.

[14] M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [8].

[15] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. TR MCS99-20, Dept. of Comp. Sci. and Applied Math., The Weizmann Institute of Science, Rehovot, Israel, Apr. 2000.

[16] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996.

[17] R. Heckel and S. Sauer. Strengthening UML collaboration diagrams by state transformations. In H. Hußmann, editor, *Proc. FASE'2001, Genova, Italy*, volume 2185 of *LNCS*. Springer-Verlag, 2001.

[18] R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in TCS*, 1995.

[19] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley, 1999.

[20] Kennedy-Carter. eXecutable UML. `http://www.kc.com/html/xuml.html`.

[21] H. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML diagrams for production control systems. In *Proc. of the 22th International Conference on Software Engineering (ICSE), Limerick, Irland.* ACM Press, 2000.

[22] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In *Proc. UML 2001, Toronto, Kanada*, volume 2185 of *LNCS*. Springer-Verlag, 2001.

[23] A. Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proc. International Conference on Software Engineering (ICSE 2000), Limerick (Ireland).* ACM Press, 2000.

[24] M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep, M. J. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

[25] T. Mens. Conditional graph rewriting as a domain-independent formalism for software evolution. In *Proc. Int. Agtive '99 Workshop: Applications of Graph Transformations with Industrial Relevance*, volume 1779 of *LNCS*. Springer-Verlag, 2000.

[26] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *Software Engineering*, 20(10):760–773, 1994.

[27] Object Management Group. UML specification version 1.4, 2001. `http://www.celigent.com/omg/umlrtf/`.

[28] D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M. Sleep, M. Plasmeijer, and M. C. van Eekelen, editors, *Term Graph Rewriting*, pages 201–214. Wiley, 1993.

[29] S. Spaccapietra and C. Parent. View integration: A step forward in solving structural conflicts. *IEEE Transactions on Software and Data Engineering*, 6(2):258–274, 1994.

[30] Standish Group. Software chaos. `www.standishgroup.com/chaos.html`.

[31] P. Stevens. On Use Cases and their relationships in the Unified Modelling Language. In H. Hussmann, editor, *Proc. FASE 2001, Genova, Italy*, volume 2029 of *LNCS*. Springer-Verlag, 2001.

[32] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems.* PhD thesis, TU Berlin, 1996. Shaker Verlag.