Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams : Long Version

Stefan Jurack[1], Leen Lambers[2], Katharina Mehner[3], Gabriele Taentzer[1]

[1] Philipps-Universität Marburg, Germany,
{sjurack,taentzer}@mathematik.uni-marburg.de
[2] Technische Universität Berlin, Germany, leen@cs.tu-berlin.de
[3] Siemens, Corporate Technology, Germany, katharina.mehner@siemens.com

Abstract. In use case-driven approaches to requirements modeling, UML activity diagrams are a wide-spread means for refining the functional view of use cases. Early consistency validation of activity diagrams is therefore desirable but difficult, due to the semi-formal nature of activity diagrams. In this paper, we specify well-structured activity diagrams and define activities more precisely by pre- and post- conditions. They can be modeled by interrelated pairs of object diagrams based on a domain class diagram. This activity refinement is based on the theory of graph transformation and paves the ground for a consistency analysis of the required system behavior. A formal semantics for activity diagrams refined by pre- and post-conditions, allows us to establish sufficient criteria for consistency. The automated checking of these criteria has been integrated into a tool for graph transformation.

1 Introduction

Requirements engineering is the process of gathering and structuring information on a software system. A consistent requirements document is extremely important as it provides the basis for all relevant development decisions. The detection of requirement errors late in the development process causes expensive iterations through all phases.

In object-oriented software development, the UML [1] has become the standard notation for software models at different stages of the life cycle and at different levels of abstraction, including the requirements specification. The result of requirements elicitation consists of a domain class diagram and a use case specification. The main scenario(s) of a use case are often specified with activity diagrams. The dynamic aspect—when something can be done—is captured by activity diagrams. The functional aspect—how it can be done—is described by pre- and post-conditions of activities which are first described in natural language. In particular, the functional aspect has not been formally integrated with the static domain model. The intended connections between domain classes and activity diagrams can mainly be indicated by giving meaningful names to activities.

An early consistency check of activity diagrams taking into account pre- and post-conditions is not possible due to the informal nature of activity specifications. By consistency we mean that all flow paths of an activity diagram can be performed. In this

situation, a more precise specification of each activity can pave the ground for a consistency analysis. We propose to refine activity diagrams by describing pre- and post-conditions of each activity by a pair of interrelated object diagrams. These object diagrams link formally the pre- and post-conditions with the domain model. The aim of the analysis is to validate that control flow is consistent with pre- and post-conditions.

Graph transformation systems can be used to formalize this problem and provide tool support for the analysis. A pair of pre- and post-conditions can be formalized as a graph transformation rule. The idea was first presented in [2] to analyze conflicts and dependencies between activities during use case integration, however not yet taking into account the control flow structure. The idea was further developed in [3, 4] for analyzing inconsistencies during composition of an aspect-oriented extension of activity diagrams. The approach classified sources of inconsistencies taking into account the control flow. [5] provides sufficient applicability criteria for graph transformation rule sequences as formal foundation. This idea has been employed [6] in the context of adaptable service-based applications in order to validate control flow structures described with live sequence charts[7].

Based on our previous work, this paper presents sufficient criteria for consistency analysis of activity diagrams. In order to apply the existing applicability criteria for graph transformation rule sequences, we provide a semantics for activity diagrams refined with pre- and post-conditions. We also improve the existing applicability criteria and introduce new reduction mechanisms for the analysis phase.

The paper is organized as follows: Sect. 2 illustrates the use-case driven approach to requirement modeling and introduces the running example. In Sect. 3, we recall the concept of graph transformation, explain conflicts and dependencies of graph transformations, and present sufficient criteria for the applicability of graph transformation rule sequences together with two new reduction mechanisms for the applicability analysis of rule sequences. In Sect. 4 we present a graph transformation based semantics of refined activity diagrams, define consistency, and introduce an efficient way to analyze for consistency. We analyze our running example in Sect. 5. Related work is discussed in Sect. 6. Sect. 7 contains our conclusion and outlook.

2 Use Case-Driven Requirement Modeling

An use-case diagram specifies a number of main scenarios by use-cases, actors, and dependencies between them. UML activity diagrams can be used to model the behavior of each use-case. In our approach we formalize activities by pre- and post-conditions based on a domain class diagram. In the following, we introduce into this approach with a small case study of an online pizza service.

2.1 Use-Cases

The center of our requirement analysis is an online service for a pizzeria offering pizzas and beverages. The pizzeria staff enters master data, e.g. different kind of pizzas, toppings and beverages. Customers have to register before they are allowed to order food and beverages. As long as the order is not closed, further order items may be attached.

Correspondingly, the system offers the use cases “create master data” and “take order” (cf. Fig. 1). The complex use case ‘take order” includes use case “add pizza to order”. Further use cases are conceivable but not considered in the following due to space constraints such as requesting a voucher (“request voucher”) or contacting the pizzeria staff (“write message”)

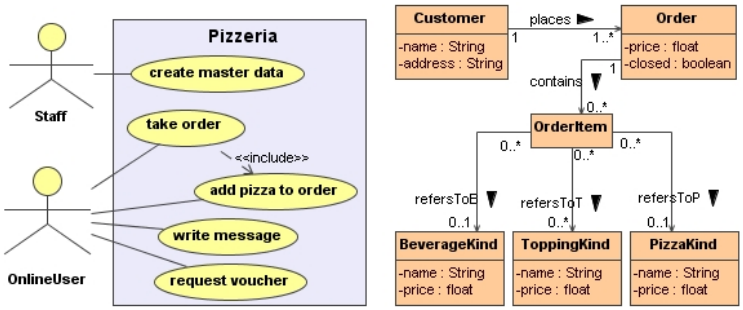


Fig. 1. Pizzeria example: Use cases (left), Domain model (right)

2.2 Domain Model Class Diagram

The corresponding domain model class diagram is shown in Fig. 1. It illustrates an *Order* placeable by a *Customer*. An order is constituted by a number of items (*OrderItem*), whereas each item may be associated with a kind of pizza (*PizzaKind*) and if desired multiple kinds of additional toppings (*ToppingKind*), or the item is associated with a *BeverageKind*. The structure of the domain model also allows a pizza and a beverage both to be associated to the same order item, which is semantically undesired. To solve this constraint, an OCL expression could be formulated. In the following our refinements prevent the occurrence of such associations anyway though. Furthermore there are three classes with equal attributes, which may be resolved by introducing inheritance. Since the tool feature [8] of analyzing inheritances is not fully released yet because of lacking theory for graph transformation with inheritance, we stick to this simple domain model without inheritance this time.

2.3 Activities

Figure 2 shows the activity diagrams refining corresponding use cases. Use case “create master data” is refined in the top left corner. It allows to create pizza, topping and beverage kinds. The bottom of the figure shows the activity diagram refining use case “take order”. It conditionally creates a customer instance followed by order creation. Pizza or a beverage can be added to the order in a loop. If all desired order items are added, the order is closed. The icon in the bottom right corner of activity “Add pizza to order” indicates a reference to the activity diagram shown in the top right corner of

Figure 2. It refines the use case “add pizza to order”, whereas the corresponding *include* relation between the use-cases was already depicted in the use-case diagram (cf. Fig. 1).

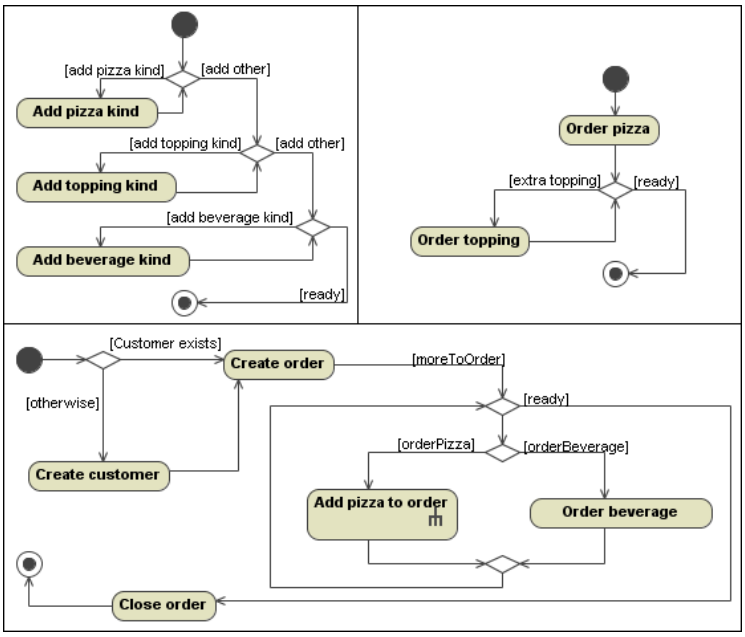


Fig. 2. Activity diagrams: Create master data(left), Add pizza to order(right), Take order(bottom)

2.4 Pre- and post-conditions

Based on the domain model, the pre- and post-conditions of each activity can be specified using interrelated object diagrams extended by negative application conditions. The interrelation is expressed by numbers. Equal numbers refer to same instances. Similar to object flow, object creation is specified by introducing a new element in a post-condition. Object deletion is specified by by presenting it in the pre-condition only. A pair of pre- and post-conditions can be parameterized. Negative application conditions (NAC), depicted in red dashed (out)line specify objects which must not exist.

Fig. 3 shows the pre- and post-condition pairs of activity “Add pizza kind” of activity diagram “Create master data”. A new *PizzaKind* object is created if a corresponding *PizzaKind* object does not exist. Input parameters are *n* and *p* providing values for the *name* and *price* attributes. Conditions for activities “Add topping kind” and “Add beverage kind” are composed analogously.

Fig. 4 shows the pre- and post-conditions of the activities of activity diagram “Take order” where activity “Add pizza to order” has been replaced by its refinement. They can be described as follows:

Fig. 3. Pre- and post-conditions for activity "Add pizza kind" of diagram "Create master data"

(1) A *Customer* is created with *name* and *address* set by input parameters *n* and *adr*, if an equally named customer does not exist already.
(2) An *Order* is created and linked to a *Customer* identified by parameter *n*, but only if no open order is already associated i.e. attribute *closed* is *false*.
(3) A new *OrderItem* is created and linked to an existing *PizzaKind* object. The pizza name is given as parameter.
(4) An existing *ToppingKind* is linked to the *OrderItem* related to a *PizzaKind*. The topping name is given as parameter.
(5) An existing *BeverageKind* is linked to an *Order* via a new created *OrderItem*.
(6) An open *Order* is closed i.e. the attribute *closed* which must be *false* before is set to *true*.

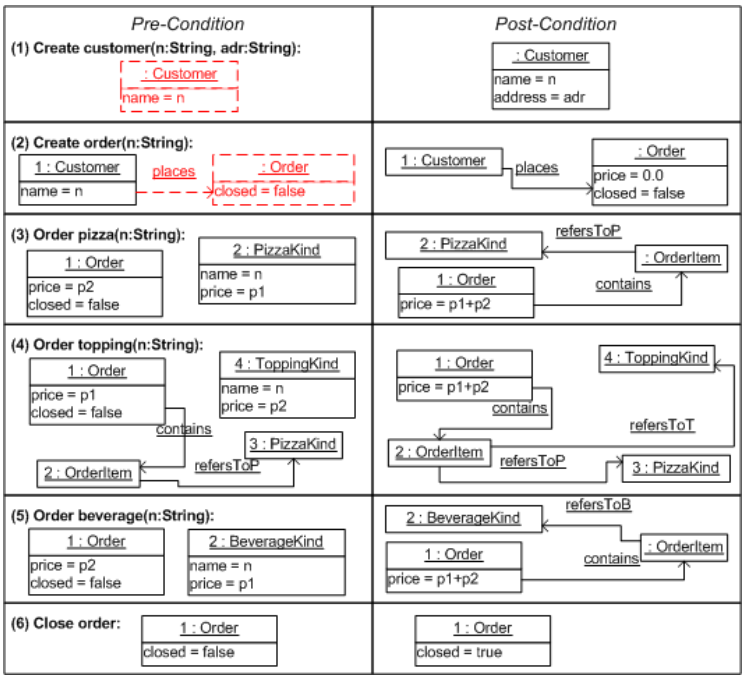


Fig. 4. Pre- and post-conditions for activities of activity diagram "Take order"

3 Formalization by Graph Transformation

The UML variant presented in Sect. 2 can be precisely defined by the theory of graph transformation [9]. While class diagrams are formalized by type graphs, activities with pre- and post conditions are mapped to graph rules. This serves as a basis for analyzing use case-driven requirement models precisely. Here, we summarize the main ideas.

3.1 Graphs and graph transformation

When formalizing object-oriented modeling, graphs occur at two levels: the type level (defined based on class diagrams) and the instance level (given by all valid object diagrams). This idea is described by the concept of *typed graphs*, where a fixed *type graph* TG serves as an abstract representation of the class diagram. Types can be structured by an inheritance relation. Multiplicities and other annotations are expressed by additional graph constraints. Instances of the type graph are object graphs equipped with a structure-preserving mapping to the type graph. A class diagram can thus be represented by a type graph, plus a set of constraints over this type graph.

Graph transformation is the rule-based modification of graphs. Rules are expressed by two graphs (L, R), where L is the left-hand side (LHS) of the rule representing the pre-condition and R is the right-hand side (RHS) describing the post-condition, and a mapping between objects in L and R. $L \cap R$ (the graph part that is not changed) and the union $L \cup R$ should form a graph again, i.e., they must be compatible with source, target and type settings, in order to apply the rule. Graph $L \setminus (L \cap R)$ defines the part that is to be deleted, and graph $R \setminus (L \cap R)$ defines the part to be created. Figure 4 shows pre- and post-conditions of the activities which can be interpreted as graph rules. Numbers indicate a mapping between left- and right-hand sides. Please note that the left-hand sides may contain two kinds of links, drawn by solid and dashed lines. Only the solid parts belong to the left-hand side and formulate a positive pre-condition while the dashed ones prohibit certain graph parts and represent negative application conditions (NACs). For example, rule *Create order* (cf. Fig. 4(2)) describes how an *Order* is created for a certain *Customer* and prohibits the existence of an already associated order with attribute *closed=false* at the same time i.e. the rule is applicable only, if all placed orders have been closed.

A *graph transformation step* $G \overset{r,m}{\Longrightarrow} H$ between two instance graphs G, H is defined by first finding a match m of the left-hand side L of rule r in the current instance graph G such that m is structure-preserving and type-compatible, and second by constructing H in two passes: (1) build $D := G \setminus m(L \setminus (L \cap R))$, i.e., delete all graph items that are to be deleted; (2) $H := D \cup (R \setminus (L \cap R))$, i.e., create all graph items that are to be created. A *graph transformation (sequence)* consists of zero or more graph transformation steps. A set of graph rules, together with a type graph, is called a *graph transformation system* (GTS). A GTS may show two kinds of non-determinism: (1) For each rule several matches may exist. (2) Several rules might be applicable.

The tool environment AGG (Attributed Graph Grammar System) [8] can be used to specify graph transformation systems and analyze their rules.

3.2 Conflicts and dependencies

As discussed in the previous section, several rules may be applicable to a host graph. Either the results might be the same regardless of the application order, or if one of two rules is not independent of the second, the first one will disable the second. In this case, the two rules are in *conflict*. Conversely, two rules are said to be *parallel independent* if they do not disable each other. Instead, *sequential independence* guarantees that the order of application in a transformation sequence does not matter.

One of the main static analysis facilities for GTSs is the check for potential conflicts and dependencies between rules, both supported in AGG. This conflict and dependency analysis is based on critical pair analysis (CPA) [9, 2]. A critical pair is a pair of transformation steps $G \overset{r_1,m_1}{\Longrightarrow} H_1$, $G \overset{r_2,m_2}{\Longrightarrow} H_2$ that are in conflict in a minimal context, identified through matches m_1 and m_2. The following conflicts occur:

delete/use : The application of r_1 deletes an element used by the match of r_2.
produce/forbid : The application of r_1 produces an element that a NAC of r_2 forbids.
change/use : The application of r_1 changes an attribute value used by the match of r_2.

Critical pair analysis is also used to find potential dependencies between a transformation applying r_1 and another one applying r_2. This case is led back to critical pairs consisting of steps $H \overset{r_1^{-1},m_1'}{\Longrightarrow} G$ and $H \overset{r_2,m_2}{\Longrightarrow} K$. The inverse rule r_1^{-1} is obtained by exchanging LHS and RHS and translating the NACs into equivalent ones from LHS to RHS [9, 5]. The following dependencies occur:

produce/use : The application of r_1 produces an element needed by the match of r_2.
delete/forbid : The application of r_1 deletes an element that a NAC of r_2 forbids.
change/use : The application of r_1 changes an attribute value used by the match of r_2.

Rule r_2 *purely depends* on rule r_1 if r_2 its left-hand side can be completely embedded in the right-hand side of r_1. This means that if rule r_1 has been applied then it delivered everything rule r_2 needs to be applied as well. Note that NACs belonging to r_2 are not necessarily satisfied though and still have to be checked.

If there are neither potential dependencies between transformations via r_1 and r_2 nor via r_2 and r_1 then rules r_1 and r_2 are sequentially independent. This leads to the fact that r_1 and r_2 can be switched if they occur next to each other in a rule sequence without any impact on the applicability of the rule sequence. We call such rule sequences in which sequentially independent neighbored rules are switched *shift-equivalent*.

3.3 Criteria for Applicability of Rule Sequences

In [5] sufficient criteria are introduced for the applicability of a rule sequence to a given start graph. In many cases though a rule sequence is given without a graph. Therefore below we present applicability criteria[4] without given start graph. They describe which

[4] Note in addition that comparing to the criteria presented in [5] the criteria above contain also a minor change in the *pure enabling predecessor* criterion. We exploit in this paper the typing in the graph transformation system such that also rules with NACs are allowed to have a pure enabling predecessor.

conditions a given rule sequence and some start graph G_0 should fulfill such that it becomes applicable to G_0.

Concurrent rules For the applicability criteria we need the concept of a *concurrent rule* r_c which can be constructed from a sequence of single rules $r_1, r_2, \ldots r_n$. Such a concurrent rule r_c establishes in one transformation step the same effect as single rules $r_1, r_2 \ldots r_n$ would establish in consecutive transformation steps [10, 9]. Thus, a concurrent rule summarizes in one rule which parts of the graph should be present, preserved, deleted and produced when applying the corresponding rule sequence to this graph. Moreover we have a summarized set of NACs on the concurrent rule expressing which graph parts are forbidden when applying the corresponding rule sequence with NACs to the graph.

Applicability criteria Given a rule sequence $s : r_1 r_2 ... r_n$, the applicability criteria for s to some graph G_0 are defined as follows. Note that all criteria have to be fulfilled.

initialization: Rule r_1 is applicable to graph G_0.
no node deleting: Each rule in s must not delete object nodes.
no impeding predecessors: The predecessors of each rule r in s do not cause a conflict with r.
enabling predecessor(s): For each rule r in s:
- **pure** There is a predecessor r' of rule r in s and r is purely dependent on r'. Moreover each NAC of r forbids a graph element of type t such that an element of type t is neither present in G_0 nor produced by any predecessor of r OR
- **direct** there exists a concurrent rule r_c of rule r and its direct predecessor rule(s) such that r_c is applicable to G_0, all other predecessors do not cause a conflict with r_c, and r_c does not cause a conflict with successor rules OR
- **not needed** r itself is applicable to G_0.

Note that in order to fulfill the last criterion *enabling predecessor* it is important to check all three possibilities *pure enabling, direct enabling predecessor* and *no enabling predecessor needed* in this order. Like this it is possible to impose unnecessary restrictions to G_0.

Reduction of rule sequences A rule sequence s can be reduced before checking its applicability (as proven in the appendix), i.e. sequence s is applicable to a graph G, if the reduced sequence s' satisfies the applicability criteria and the following extra conditions are fulfilled:

1. *Repeated element reduction* Given a rule sequence $s : r_1 r_2 ... r_n$ such that two subsequent rules in s are equal. Sequence s can be reduced to s' by deleting one of these equal rules, if they are not in conflict with themselves and each subsequent rule has a pure enabling predecessor, is applicable to G, or is equal to a predecessor rule.
2. *Loop reduction* Given a rule sequence $s : s_{start}(r_1 r_1 r_2 ... r_m)^n s_{end}$ with s_{start} and s_{end} being rule sequences and $n > 2$. It is enough to execute $loop = (r_1 r_1 r_2 ... r_m)$ at most 2 times, if each rule r in s_{end} has a pure enabling predecessor, is applicable to graph G, or is equal to some rule in $loop$ or in s_{start}.

4 Consistency Analysis with Refined Activity Diagrams

Activities are defined more precisely as explained in Sect. 2 by pre- and post-conditions. They can be formalized by a graph transformation rule as introduced in Sect. 3. This activity refinement enables us to analyze consistency of the required system behavior based on the applicability criteria presented in the previous section. In this section, at first we specify well-structured refined activity diagrams and define their semantics and consistency based on graph transformation. Then we explain how to analyze in an efficient way refined activity diagrams for consistency.

4.1 Refined Activity Diagrams: Semantics and Consistency

We restrict our considerations on well-structured activity diagrams, i.e. those which consist of sequences, fork-joins, and loops only.

A *well-structured activity diagram* A consists of a start activity s, an activity block B, and an end activity e such that there is a transition between s and B and another one between B and e.

An *Activity block* is defined as follows:

- Simple: A simple activity is an activity block.
- Sequence: A sequence of two activity blocks connected by a transition form an activity block.
- Decision: A decision activity with two outgoing transitions going to an activity block each, and a merge activity with two incoming transitions from each of these blocks form an activity block.
- Loop: A decision activity followed by an activity block with an outgoing transition to the same decision activity form an activity block.
- Fork: A fork activity followed by two activity blocks followed by a join activity form an activity block.

A *refined activity diagram* A is a well-structured activity diagram such that each activity occurring in A corresponds to a unique graph transformation rule.

Given an activity block B of a refined activity diagram A its corresponding set of rule sequences S_B is defined as follows.

- If B consists of a simple activity a, $S_B = \{a\}$.
- If B is a sequence of X and Y, $S_B = S_X \; seq \; S_Y = \{s_x s_y | s_x \in S_X \wedge s_y \in S_Y\}$
- If B is a decision block on X and Y, $S_B = S_X \; or \; S_Y = S_X \cup S_Y$
- If B is a loop block on X, $S_B = loop(S_X) = \bigcup_{i \in I} S_X^i$ where $S_X^0 = \{\lambda\}$ with λ being the empty sequence, $S_X^1 = S_X, S_X^2 = S_X \; seq \; S_X$ and $S_X^i = S_X \; seq \; S_X^{i-1}$ for $i > 2$.
- If B is a fork block on X and Y, $S_B = S_X || S_Y = \bigcup s_x || s_y$ with $s_x \in S_X \wedge s_y \in S_Y$ where $s_x || \lambda = \{s_x\}$, $\lambda || s_y = \{s_y\}$, and $x s'_x || y s'_y = \{x\} \; seq \; s'_x || y s'_y \cup \{y\} \; seq \; x s'_x || s'_y$.

Thus we define the *semantics* $Sem(A)$ of a refined activity diagram A consisting of a start activity s, an activity block B, and an end activity e as the set of rule sequences $S(B)$ generated by the main activity block B.

We define an activity diagram A to be *consistent* if there exists a non-empty set of graphs such that all rule sequences in $Sem(A)$ are applicable to each of them. Each of these graphs then represents a potential snapshot of the system to which the activity diagram applies consistently.

4.2 Analysis of Reduced Semantics

Based on the semantics and consistency definition of an activity diagram given in the former paragraph we are able to apply the criteria for checking the applicability of rule sequences given in Sect. 3 in order to check for consistency.

Due to loops though, the sets of rule sequences can become infinite. For criteria checking, a rule set has to be finite, since otherwise the checking procedure would not terminate. To check the applicability criteria, it is sufficient though to repeat a loop at most two times as mentioned in the former section. Moreover we can reduce the rule sequences by the repeated element reduction as defined in the last section.

Thus we define a *reduced set of rule sequences* $Red(A)$ for a refined activity diagram A as follows:

- Reduce the set of rule sequences in $Sem(A)$ by applying the loop reduction as defined in the last section. We call this the loop-reduced semantics $LRed(A)$ for A.
- Reduce further the set of rule sequences in $LRed(A)$ by applying the repeated element reduction as defined in the last section as long as possible.

Note that in the appendix we have proven that by showing that all sequences in $Red(A)$ satisfy the applicability criteria all sequences in $Sem(A)$ will be applicable.

5 Example Analysis

Consider the pizzeria scenario in Sect. 2. For now we analyze the use cases "create master data" and "take order" separately. Afterwards we will reason about the interrelation of both. In order to analyze the scenario the following steps are recommended:

1. Generate conflicts and dependencies of the activity's rules.
2. Identify all rule sequences and reduce them as possible
3. Analyze applicability of reduced set of rule sequences according to criteria[5]
4. Draw conclusions regarding the activities or activity diagrams, respectively.

[5] AGG [8] offers a new module for automatically checking the applicability criteria for given rule sequences and a dedicated start graph. Here we explain manually this check step by step for clarity reasons.

5.1 Create master data

Fig. 5 depicts the conflict and dependency matrices calculated by the tool AGG. The input data are rules analog to the pre- and post-conditions in Fig. 3. Note that each of these rules is parametrized. Consider e.g. rule *Add pizza kind(n:String, p:float)*. It holds a parameter for the pizza name and another for its price. The rule can only be applied if concrete values for its parameters are given. In fact, a parametrized rule describes a set of regular rules. For each possible parameter value there is a corresponding rule. E.g. rule *Add pizza kind("Margherita", 5)* can be applied whenever there does not already exist a Margherita pizza. Otherwise, this is called conflict which is shown in the conflict matrix computed by AGG. In particular, it shows a conflict between *Add pizza kind(n:String, p:float)* and *Add pizza kind(n':String, p':float)* if the parameter value for *name* i.e. *n* and *n'* is the same. This holds analogously for the rules *Add topping kind* and *Add beverage kind*. In all other cases there are neither conflicts nor dependencies.

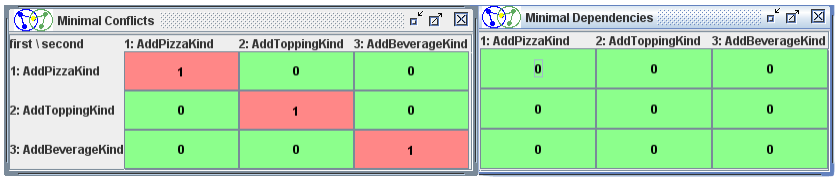


Fig. 5. Conflict and dependency matrix of "Create master data"

Now we identify all activity sequences in order to determine their applicability. In fact we operate on the rules corresponding to the activities in the sequences. For clarity reasons we use acronyms in the sequences: AP=Add pizza kind, AT=Add topping kind, AB=Add beverage kind. Deriving the set of sequences from the activity diagram and applying the loop reduction as defined in Sect. 3.3, we result in 27 sequences. Due to space constraints we only show the analysis for the following two sequences : $s_1 =< AP(n1, p1), AP(n2, p2) >$ and $s_2 =< AP(n3, p3), AT(n4, p4) >$.

The criteria check is straight forward. Both sequences start with rule AP, whose left-hand side (LHS) is empty i.e. no requirements concerning the existence of objects are given. However, considering the negative application condition (NAC) of AP and the reasoning above, we conclude that the start graph may be empty or at least must not contain a *PizzaKind* object with a name according to $n1$. If both constraints are satisfied, the *initialization* criterion is fulfilled for the sequences. Since both sequences contain only rules with empty left-hand sides, criteria *no node deleting* is obviously satisfied as well. Without further assumptions the criterion *no impeding predecessors* is satisfied for sequence s_2 since its rules are not in conflict with each other as the conflict matrix shows (cf. Fig. 5). Contrarily, sequence s_1 satisfies the criterion only, if the values $n1$ and $n2$ differ, otherwise the first rule impedes the second. The last criterion to check is *enabling predecessor(s)*. The second rule AT of sequence s_2 is applicable analogously to AP i.e. an equally named *ToppingKind* object must not exist in the start

graph. Similarly, the second rule in sequence s_1 is applicable to the start graph only, if no equally named *PizzaKind* object exists.

Proceeding this analysis to all rule sequences, we can summarize, that the refined activity diagram "Create master data" is always applicable, if multiple occurring equal rules within a sequence do not use the same parameter values for their attribute *name* and the start graph does not already contain corresponding objects. If both constraints hold, the loop in the activity diagram can be run through as long as desired. Therefore, from a practical point of view pizza, topping and beverage kinds cannot occur twice with the same name in the menu.

5.2 Take order

Now we consider use case "take order" and its activity diagram (cf. Fig 2). As already addressed, an *include* reference to another use case "add pizza to order" is used. We deal with that situation by treating the activities of the corresponding activity diagram as part of the embedding one. Similar to the previous section some rules are parametrized. In the following we show the analysis of one sequence only, due to space limitations.

At first we take a look at the dependency and conflict matrices calculated by AGG (cf. Fig. 6). For the reader's convenience uninteresting entries have been colored gray respecting the control flow. This is conceivable to be done automatically in the future.

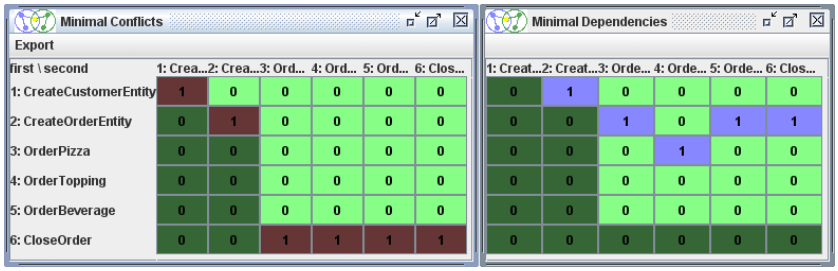


Fig. 6. Matrices with respect to the control flow

The next step is to identify all possible rule sequences. Analog to the previous analysis we use acronyms in the sequences for better readability: OP=Order pizza, OT=Order topping, OB=Order beverage, CC=Create customer, CO=Create order, CL=Close order. Using the *loop reduction* to prevent infinite sequences we result in over 350 sequences, which can be reduced further by about 50% applying the *repeated element reduction*. If we combine this reduction technique with the so-called *shift-equivalence* i.e. neighbored rules in sequences can be switched if they are sequentially independent (cf. Sect. 3.2), about 20 sequences remain. For example, since rules OB and OP are sequentially independent, both sequences $< CO, OB, OP, OB, CL >$ and $< CO, OB, OB, OP, CL >$ can be transfered into one another and therefore only one needs to be analyzed. Note, that both sequences are shown without parameters.

Exemplarily, we analyze the sequence $s_3 =< CO(n), OP(p), OT(t), CL >$. According to the applicability criteria in Sect. 3.3 we start our analysis with criterion *initialization*. Sequence s_3 is exclusively applicable to graphs with a *Customer* object named according to parameter n of rule CO and additionally the object must not be linked with an open order i.e. attribute's *closed* value is *false* .

Criteria *no node deleting* and *no impeding predecessor* are satisfied, since no node is deleted by any rule and the conflict matrix (cf. Fig. 6) does not show any conflict between corresponding rules. The last criterion to check is *enabling predecessor(s)* where we reason about rules needing enabling predecessor(s) if they are not already applicable to the start graph. Rule OP requires a *PizzaKind* object named accordingly to parameter *p*. If available in the start graph, a concurrent rule with CO can be created which is applicable to the start graph. This concurrent rule expresses that a customer with name n must exist for whom an order is created holding one pizza with name p. The concurrent NAC to this rule expresses that the customer with name n must not be already linked with an open order. Rule OT can be analyzed analogously. Since OT is enabled by OP which is enabled by CO itself, a concurrent rule consisting of all three rules is applicable if the start graph contains a *ToppingKind* object named according to parameter *t*. The last rule CL is purely enabled by CO. Concluding, sequence s_3 is applicable if the start graph contains a *Customer* object not associated with an open order, a *PizzaKind* and a *ToppingKind* object, all named according to parameters n, p and t.

The analysis of the remaining sequences is quite similar to the previous one and not shown here. As result we reveal the applicability of all given sequences with respect to specific sets of start graphs. This implies, that the activity diagrams are valid concerning their consistency to corresponding system states.

Finally we observe, that almost every rule sequence of use case "take order" require objects of type *PizzaKind*, *ToppingKind* and *BeverageKind* which are created by use case "create master data". This is quite meaningful since the pizzeria staff has to provide product informations before a customer can order food and beverages. Only sequences $< CC, CO, CL >$ and $< CO, CL >$ do not require such objects as they do not order anything. However, these sequences are not that meaningful.

6 Related work

The work presented in this paper is rooted in formal semantics and analysis of activity diagrams, model driven engineering, and graph transformation systems.

Eshuis [11] proposes denotational semantics for a restricted class of activity diagrams by means of labeled transition systems. Model checking is then used to check generic properties or model specific properties. For reducing the state space it is assumed that a system will not forever stay in a loop. Stoerrle [12] defines a denotational semantics for control flow of UML 2.0 activity diagrams including procedure calls by means of petri nets. By using standard petri net analysis, the translated activity diagrams can be analysed for generic properties, e.g. reachability or deadlock freeness.

Our approach is dedicated to checking one property, namely consistency. The advantage of our approach is that we do not generate the whole state space like model checking. We generate a finite number of sequences from an activity diagram by cutting of loops in a way that is consistent with the criteria we are checking on each sequence. The disadvantage of our approach is that we cannot decide for every activity diagram whether it is consistent or not. It is unique to our approach that we refine activities with pre- and postconditions and that we take into account this refinement during the analysis.

In the area of model driven engineering, Jayaraman, Whittle et al. [13] use critical pair analysis to detect dependencies and conflicts between features modelled as a graph transformation modifying UML diagrams. This approach however is limited to a pairwise analysis of transformations. No control structure such as activity diagrams refine the analysis.

Fujaba [14] and Moflon [15] are mature graph transformation tools usable for specifying and executing transformations similar to AGG. Fujaba uses a kind of activity diagrams called story diagrams and integrates them with graph transformations similar to our approach. But AGG is the only graph transformation tool which supports CPA and the applicability checks on rule sequences as suggested in this paper.

7 Conclusion

In this paper, we present sufficient criteria for checking the consistency of activity-based behavior models in the context of requirements engineering. The additional effort of consistency checking pays off if an early formal analysis is required. We check the consistency of refined activity diagrams. Activities are equipped with pre- and post-conditions which are formulated by interrelated object diagrams typed over a common domain model. This allows us to define the behavior of refined activity diagrams by graph transformation rule sequences. Sufficient criteria developed in the context of graph transformation can then be applied to check for consistency. Thereby one of the main new technical insights in this paper is that we are able to conclude applicability of an infinite set of system runs by analyzing only a finite one. Infinite sets of runs occur due to loops in activity diagrams. We have shown that it is enough to check only a finite set of runs, if a newly introduced reduction mechanism is respected. Further reductions are conceivable by taking the object flow into account. However, this is future work.

Although there are applicable graph transformation rule sequences which do not satisfy the given criteria, they seem to apply often, provided that all rules do not delete object nodes. For the time being, this criterion which seems to be the most restricting one, can be circumvented by just detaching object nodes from the main object structure. Besides their application to modeling of services and service orchestration, our criteria need further evaluation in future work.

Graph transformation tool AGG offers a new analysis module for checking the presented applicability criteria for given rule sequences with dedicated start graph. We intend to extend this tool support by generating graph constraints to be fulfilled by start graphs of applicable rule sequences. For integrating this analysis with a UML CASE tool, it has to support the modeling of refined activities and has to provide a translation

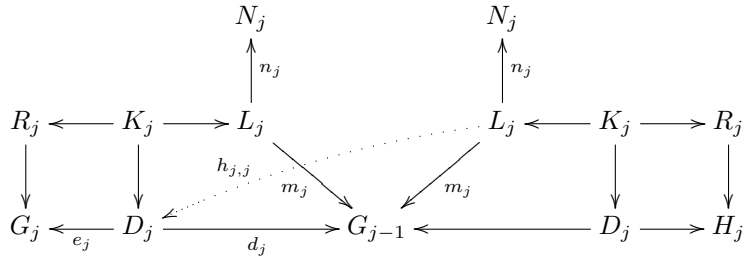
to graph transformation rule sequences. If a description of pre- and post-conditions by interrelated object diagrams is not offered, their definition by OCL constraints might be supported. Translating a restricted form of OCL constraints to graph rules could also enable an automated analysis based on AGG.

Our results are not limited to consistent behavior modeling for requirements engineering only. They help in identifying unintended imprecisions and may also be applied to the rigorous analysis of various kinds of work flow and business process models.

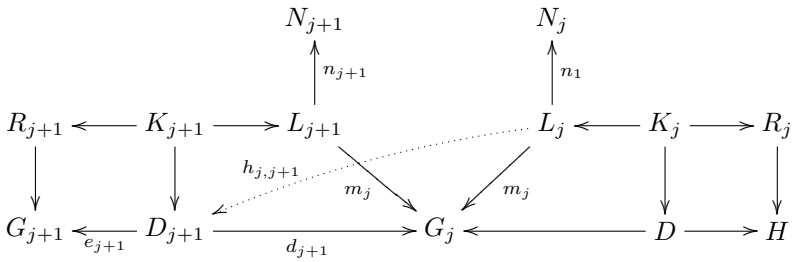
8 Appendix

Lemma 1. *Given a rule sequence $p_1p_2 \dots p_n$ which is applicable to G_0 and each of the rules not node-deleting. The rule sequence $p_1p_2 \dots p_np_j$ with $1 \leq j \leq n$ is applicable to G_0 if p_i with $j \leq i \leq n$ causes no conflict with p_j.*

Proof. Let $G_0 \overset{p_1}{\Rightarrow} G_1 \dots G_{n-1} \overset{p_n}{\Rightarrow} G_n$ be the graph transformation sequence arising by applying rule sequence $p_1p_2 \dots p_n$ to G_0. Consider then the following diagram:



Since p_j causes no conflict with p_j the morphism $e_j \circ h_{j,j}$ exists satisfying NAC_{p_j}. This makes p_j applicable to G_j. Therefore we can construct the following diagram which exists because p_j is not node-deleting:



Since p_{j+1} does not cause a conflict with p_j the morphism $e_{j+1} \circ h_{j,j+1}$ exists satisfying NAC_{p_j}. This makes p_j applicable to G_{j+1}. We can iterate this argumentation up until the conclusion that p_j is applicable to G_n. Note that this argumentation is analogous to the induction argument (i) in the proof of the applicability criteria in [5] for sequence $G_{j-1} \overset{p_j}{\Rightarrow} G_j \dots G_{n-1} \overset{p_n}{\Rightarrow} G_n$ and rule p_j. This is because p_j is not node-deleting, p_j does not have impeding predecessors in $p_j \dots p_n$ and p_j is applicable to G_{j-1}. Therefore we can conclude that the graph transformation sequence

$G_0 \overset{p_1}{\Rightarrow} G_1 \ldots G_{n-1} \overset{p_n}{\Rightarrow} G_n \overset{p_j}{\Rightarrow} G_{n+1}$ exists and thus $p_1 p_2 \ldots p_n p_j$ is applicable to G_0 as well.

Lemma 2. *Given a transformation sequence $t : G_0 \overset{p_1}{\Rightarrow} G_1 \ldots G_{n-1} \overset{p_n}{\Rightarrow} G_n$ via the rule sequence $p_1 p_2 \ldots p_n$. Then p is applicable to G_n whenever p is applicable to some intermediate graph G_j with $0 \leq j \leq n$ in the transformation sequence t and p_i with $j \leq i \leq n$ causes no conflict with p and p is not node-deleting.*

Proof. Let G_j with $1 \leq j \leq n$ be the intermediate graph in t to which p is applicable. Consider now the graph transformation sequence $t' : G_j \Rightarrow^* G_n$ via $p_{j+1} \ldots p_n$ arising by cutting of the first j steps of transformation sequence t. Now we observe that we can apply induction argument (i) of the correctness proof of the applicability criteria in [5] to p and $t' : G_j \Rightarrow^* G_n$. This is because p is not node-deleting, there are no impeding predecessors in $p_j p_{j+1} \ldots p_n$ for p, and p is applicable to G_j. Therefore we can conclude that p is applicable to G_n and in the end a graph transformation sequence $t'' : G_0 \Rightarrow^* G_n \overset{p}{\Rightarrow} G_{n+1}$ exists via $p_1 p_2 \ldots p_n p$.

Theorem 1 (repeated element reduction). *Given a rule sequence $s : r_1 r_2 ... r_n$ such that two rules r_i and r_{i+1} for $1 \leq i < n$ are equal and r_i is not in conflict with itself. Sequence s is applicable to G_0, if sequence $s' : r'_1 r'_2 \ldots r'_{n-1}$ being s without r_{i+1} fulfills the applicability criteria for G_0 such that the following holds for each r'_j with $j > i$:*

- *r'_j has a pure enabling predecessor in s'*
  OR
- *r'_j is applicable to G_0*
  OR
- *r'_j is equal to some r'_k with $k < j$.*

Proof. Rule r_{i+1} can be appended to rule sequence $r_1, r_2 \ldots r_i$ without influencing applicability to G_0 because of Lemma 1. This is because r_{i+1} equals r_i and rules $r_1, r_2 \ldots r_i$ do not cause a conflict with r_{i+1}. The latter condition holds since $r_1, r_2 \ldots r_i$ fulfills the impeding predecessor criterion and r_{i+1} equals r_i and in addition r_i does not cause a conflict with itself. Therefore $r_1 r_2 \ldots r_i r_{i+1}$ is applicable to G_0. Now we can append rule r_{i+2} because of the following argumentation. If $r_{i+2} = r'_{i+1}$ has a pure enabling predecessor in s' it has one in s as well. This is because the set of predecessors of r'_{i+1} in s' is equal to the set of predecessors of r_{i+2} in s. Therefore we can apply Lemma 2, since r_{i+2} is applicable to the resulting graph of its enabling predecessor, each predecessor of r_{i+2} does not cause a conflict since the impeding predecessor criterion holds in s', and r_{i+2} is not node-deleting because the no node-deleting criterion holds for all rules in s'. If $r_{i+2} = r'_{i+1}$ is applicable to G_0 we can apply analogously Lemma 2 again. In particular, the intermediate graph equals G_0 now. If $r_{i+2} = r'_{i+1}$ is equal to some r'_k with $k < i + 1$ we can apply Lemma 1. This is because r'_k equals some predecessor of r_{i+2} in s. Moreover the set of predecessors of r'_{i+1} in s' is equal to the set of predecessors of r_{i+2} in s, and the impeding predecessor criterion holds for s'. We can argument analogously for all r_j with $n \geq j > n + 2$.

Theorem 2 (loop reduction). *Given a rule sequence $s : q(r_1 r_2 \dots r_m)^n q'$ consisting of a rule sequence q followed by an n-ary loop $(n > 2)$ of rule sequence $r_1 r_2 \dots r_m$ followed by a rule sequence q'. Sequence s is applicable to G_0 if sequence $s' : q(r_1 r_2 \dots r_m)^2 q'$ being s with $r_1 r_2 \dots r_m$ repeated only twice fulfills the applicability criteria for G_0 such that the following holds for each r in q' belonging to s':*

- *r is equal to some rule occurring in $q(r_1 r_2 \dots r_m)^2$*
  OR
- *r has a pure enabling predecessor in s'*
  OR
- *r is applicable to the start graph G_0.*

Proof. Since s' fulfills the applicability criteria it is applicable to G_0 such that a graph transformation sequence $G_0 \Rightarrow^* H \Rightarrow^* I \Rightarrow^* J$ via s' exists with H the graph obtained after applying q, I the one obtained after applying $(r_1 r_2 \dots r_m)^2$ to H and J the graph obtained after applying q' to I. Since s starts as s' with the rule sequence $q(r_1 r_2 \dots r_m)^2$ we can simply adopt the first part of the graph transformation sequence $G_0 \Rightarrow^* H \Rightarrow^* I$. Now we have to argument that the remaining rule sequence of s $(r_1 r_2 \dots r_m)^{n-2} q'$ is applicable to I. Consider $G_0 \Rightarrow^* H \Rightarrow^* I$ and the first rule of the remaining rule sequence r_1. r_1 is applicable to I because of Lemma 1, since r_1 is not node-deleting and all rules occurring in $(r_1 r_2 \dots r_m)^2$ do not cause a conflict with r_1. This is because $q(r_1 r_2 \dots r_m)^2$ fulfills the applicability criteria, in particular, the impeding predecessor criterion. Thus because of the twofold loop all rules r_j with $1 \leq j \leq m$ do not cause a conflict with r_1. Therefore the transformation sequence $G_0 \Rightarrow^* H \Rightarrow^* I \overset{r_1}{\Rightarrow} G_1$ exists. We can repeat the same argumentation for all remaining rules r_j with $1 \leq j \leq m$ occurring in $(r_1 r_2 \dots r_m)^{n-2}$ and obtain the graph transformation sequence $G_0 \Rightarrow^* H \Rightarrow^* I \Rightarrow^* K$ via $q(r_1 r_2 \dots r_m)^n$. Now we still have to argument that q' is applicable to K. We argument as follows. Consider q'_1 the first rule in q'. If it is equal to some rule in $q(r_1 r_2 \dots r_m)^2$ we can conclude by Lemma 1 that q'_1 is applicable to K. This is because q'_1 is equal to some rule in $q(r_1 r_2 \dots r_m)^2$ and thus in $q(r_1 r_2 \dots r_m)^n$. Moreover all rules occurring in $q(r_1 r_2 \dots r_m)^n$ do not cause a conflict with q'_1 since $s' : q(r_1 r_2 \dots r_m)^2 q'$ fulfills the impeding predecessor criterion. If q'_1 has a pure enabling predecessor in $q(r_1 r_2 \dots r_m)^2$ it has one in $q(r_1 r_2 \dots r_m)^n$ as well. Therefore q'_1 is applicable to the resulting graph of its pure enabling predecessor in the transformation sequence via $q(r_1 r_2 \dots r_m)^n$. Since moreover there are no node-deleting rules and no impeding predecessors for q'_1 it is applicable to K because of Lemma 2. If q'_1 is applicable to G_0 it will be applicable to K as well again because of Lemma 2. In this case the intermediate graph is in particular equal to G_0. We can argument analogously for all remaining rules in q' and obtain a graph transformation sequence starting with G_0 via the rule sequence s.

References

1. OMG: Uml resource page of the object management group. (`http://www.uml.org/`)
2. Hausmann, J., Heckel, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In: Proc. of Int. Conference on Software Engineering 2002, Orlando, USA (2002)

3. Mehner, K., Monga, M., Taentzer, G.: Interaction Analysis in Aspect-Oriented Models. In: Proc. 14th IEEE International Requirements Engineering Conference, Minneapolis, Minnesota, USA, IEEE Computer Society (2007) 66–75
4. Mehner, K., Monga, M., Taentzer, G.: Analysis of Aspect-Oriented Model Weaving. LNCS Transactions on Aspect-Oriented Software Development (2008) (to appear).
5. Lambers, L., Ehrig, H., Taentzer, G.: Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences. In Ermel, C., Heckel, R., de Lara, J., eds.: Proc. International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT'08), Electronic Communications of the EASST (2008) to appear.
6. Lambers, L., Mariani, L., Ehrig, H., Pezzè, M.: A formal framework for developing adaptable service-based applications. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE). (2008)
7. Harel, D., Marelly, R.: Come, Let's Play - Scenario-Based Programming Using LSCs and the Play-Engine. Springer (2003)
8. AGG: AGG Homepage. (`http://tfs.cs.tu-berlin.de/agg`)
9. Ehrig, H.and Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag (2005) in preparation.
10. Lambers, L., Ehrig, H., Orejas, F., Prange, U.: Parallelism and concurrency in adhesive high-level replacement systems with negative application conditions. In Ehrig, H., Pfalzgraf, J., Prange, U., eds.: Workshop on Applied and Computational Category Theory (ACCAT'07), Elsevier Science (2007)
11. Eshuis, R., Wieringa, R.: Tool support for verifying uml activity diagrams (2004)
12. Stoerrle, H.: Semantics of uml 2.0 activity diagrams. In: International Conference on Visual Languages and Human Centric Computing VLHCC, IEEE (2004)
13. Jayaraman, P., Whittle, J., Elkhodary, A., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Proceedings of the MoDELS 2007, LNCS (2006)
14. Fujaba: Fujaba homepage. (`http://www.fujaba.de`)
15. Moflon: Moflon homepage. (`http://www.moflon.org`)