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

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

[1] Technische Universität Berlin, Germany, {leen,olga}@cs.tu-berlin.de
[2] Philipps-Universität Marburg, Germany, {sjurack,taentzer}@mathematik.uni-marburg.de
[3] Siemens AG, 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 semi-automatic checking of these criteria is supported by a tool for graph transformation.

1 Introduction

Requirements engineering is the process of gathering, structuring, analyzing, and validating requirements of a software system. Requirements analysis and validation is needed to come up with a consistent requirements specification which 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 the requirements analysis 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.

* Many thanks to Hartmut Ehrig for several useful remarks.

An early consistency check of activity diagrams taking into account pre- and post-conditions is difficult 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 as introduced in [2]. These object diagrams link formally the pre- and post-conditions with the domain model. The aim of the consistency 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 to 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 inconsistencies between activities during use case integration, however not taking into account the control flow. The idea was extended 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] provided sufficient applicability criteria for graph transformation rule sequences as formal foundation. This idea was employed in [6] to validate control flow structures for adaptable service-based applications. The structures are described by live sequence charts [7].

Based on our previous work cited above, this paper presents sufficient criteria for consistency analysis of activity diagrams. In order to apply the existing applicability criteria for graph transformation rule sequences from [5], 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 automated checking of rule sequence applicability has been integrated into the graph transformation tool AGG [8]. We demonstrate the feasibility of our approach with an example. Note that this paper presents more technical details as presented already in [9] and also a section on tool support is added.

The paper is organized as follows: Section 2 illustrates the use-case driven approach to requirements modeling and introduces the running example. In Section 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 Section 4 we present a graph transformation based semantics of refined activity diagrams, define consistency, introduce an efficient way to analyze for consistency, and present tool support. We analyze our running example in Section 5. Related work is discussed in Section 6. Section 7 contains our conclusion and outlook. In Section 8 the proofs of some results are listed.

2 Use Case-Driven Requirements 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 this approach with a small case study of an online pizza service.

2.1 Use Cases and Domain Model

The center of our requirements analysis is an online service for a pizzeria offering pizzas and beverages. The pizzeria staff enters master data, e.g. different kinds 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”).

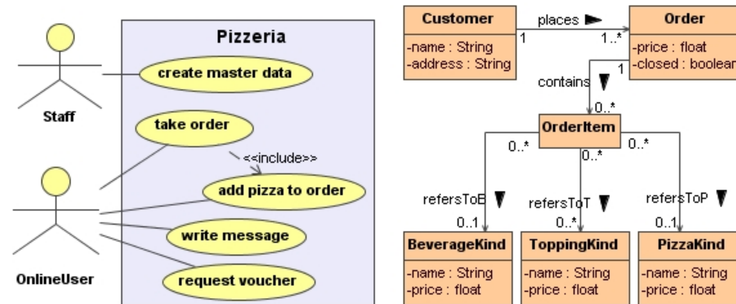


Fig. 1. Pizzeria example: Use cases (left), Domain model (right)

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 allows different pizza and/or beverages 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 of the corresponding use cases prevent the occurrence of such links though.

2.2 Activities Refined by Pre- and Post-Conditions

Fig. 2 shows the activity diagrams refining corresponding use cases as shown in Fig. 1. Use case “create master data” is refined in Fig. 2(a). It allows to create pizza, topping and beverage kinds. Fig. 2(c) shows the activity diagram refining use case “take order”. It conditionally creates a customer followed by order creation. A Pizza or 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 Fig. 2(b). 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).

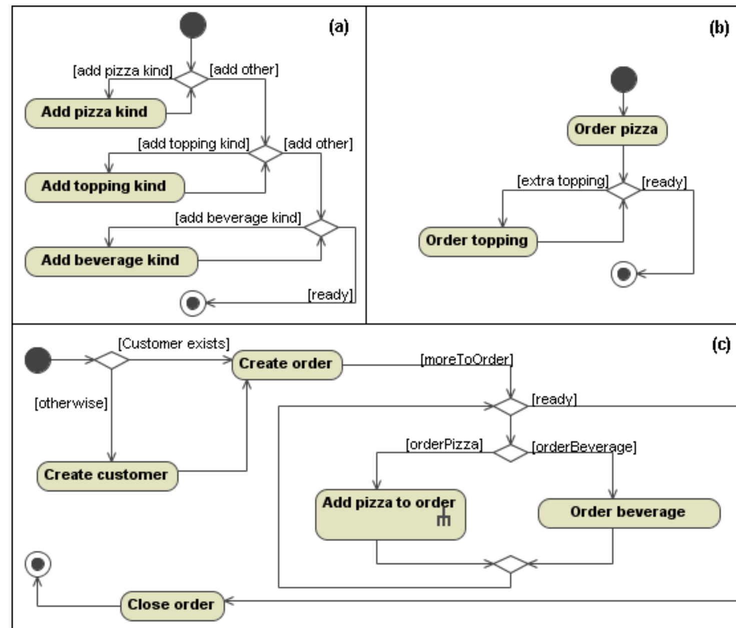


Fig. 2. Activity diagrams: Create master data(a), Add pizza to order(b), Take order(c)

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. Object creation is specified by introducing a new element in a post-condition. Object deletion is specified 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, are used in pre-conditions to specify objects which must not exist.

Fig. 3 shows the pre- and post-condition pair 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 *n* and *p* provide 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. Note that the pre- and post-conditions of activity ”Order pizza” and ”Order beverage” ensure

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

that each ordered pizza and/or beverage is associated to a different *OrderItem*. The pre- and post-conditions can be described as follows:

(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 *Order* is closed i.e. the attribute *closed* which must be *false* before is set to *true*.

3 Formalization by Graph Transformation

The UML variant presented in Sect. 2 can be precisely defined by the theory of graph transformation [10]. 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

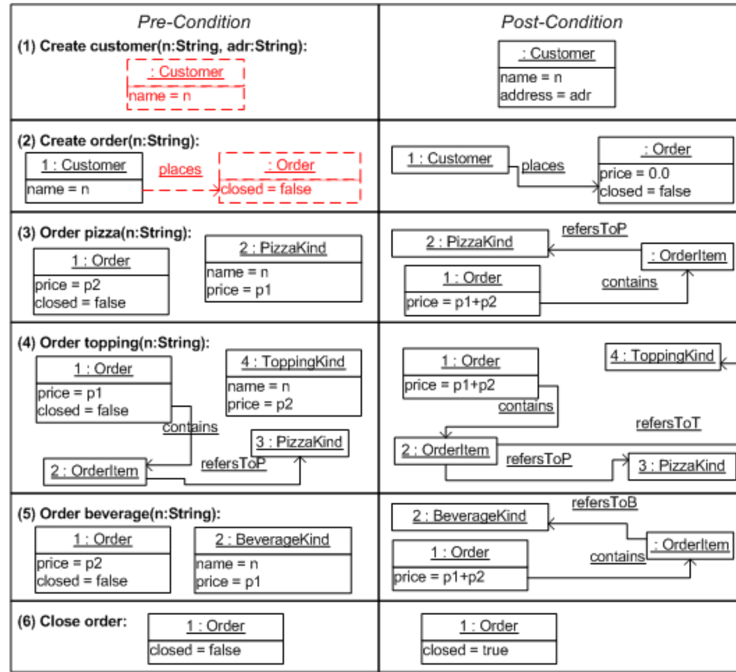


Fig. 4. Pre- and post-conditions for activities of activity diagram “Take order”

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* $G \stackrel{r,m}{\Longrightarrow} H$ 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) [10, 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 can 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 [10, 5]. The following dependencies can 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 the left-hand side of r_2 can be completely embedded into 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 start graph. They describe which 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 $r_1, r_2, \ldots r_n$ (see Def. 6 in the appendix). 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 [11, 10]. 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. Thereby a concurrent rule fixes all interrelations between the summarized rule applications. Moreover we have a summarized set of NACs on the concurrent rule expressing which graph parts are forbidden when applying the corresponding rule sequence.

We can for example build a concurrent rule p_c with NACs from rule *Create order* and *Order pizza* as depicted in Fig. 4. The lhs of p_c would contain a *Customer* node since this is needed by rule *Create order* and in addition a *PizzaKind* node needed by rule *Order pizza*. The concurrent NAC would forbid the *Customer* to have placed already an *Order* since this is forbidden by *Create order*. Rule *Order pizza* does not forbid anything such that nothing else has to be added to the concurrent NAC. The rhs of p_c consists of the rhs of *Order pizza* together with a *Customer* object appended to the *Order* node by means of a *places* edge. Now p_c summarizes in one rule the effect of *Create order* and *Order pizza* consecutively.

Applicability criteria Given a rule sequence $s : r_1 r_2 ... r_n$, the applicability criteria[4] 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
- **nearest:** There exists a concurrent rule r_c of rule r and a (sequence of) predecessor rule(s) s' such that r_c is applicable to G_0, and all predecessors of r_c do not cause a conflict with r_c. Moreover r is not in conflict with each rule between s' and r OR
- **not needed:** Rule r itself is applicable to G_0.

Note that in order to fulfill the *enabling predecessor(s)* criterion it is important to check its three possibilities *pure, nearest* and *not needed*. It is recommended to follow this order to keep the set of conditions on graph G_0 minimal. Moreover note that as soon as a

[4] In addition to the criteria presented in [5] the criteria in this paper contain two improvements in the *enabling predecessor(s)* criterion. In particular, 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. Moreover an enabling predecessor does not necessarily have to be direct if intermediate rules fulfill additional constraints. The correctness of both improvements is proven in Theorem 3 in the appendix.

rule is enabled by reusing elements delivered by some predecessor rule, this dependency should be fixed while checking the criteria for all remaining rules. This is expressed by the *compatibility condition* defined in Theorem 3 in the appendix. It is up to future work to integrate object flow in our activity diagrams in order to fix these dependencies from the beginning. Finally note that as in [5] we assume injective matching.

Reduction of Rule Sequences A rule sequence s can be reduced before checking its applicability, i.e. sequence s is applicable to a graph G_0, if s can be reduced in one of the following ways $(1-4)$ to a rule sequence s' which satisfies the applicability criteria above:

1. *Repeated element reduction (Theorem 1 in appendix)* 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_0, or is equal to a predecessor rule.
2. *Loop reduction (Theorem 2 in appendix)* Given a rule sequence $s : s_{start}(r_1 r_2 ... r_m)^n s_{end}$ with s_{start} and s_{end} being rule sequences and $m \geq 1, n > 2$. Sequence s can be reduced to $s' = s_{start}(r_1 r_2 ... r_m)^2 s_{end}$ by executing $loop = (r_1 r_2 ... r_m)$ only two times, if each rule r in s_{end} has a pure enabling predecessor, is applicable to graph G_0, or is equal to some rule in $loop$ or in s_{start}.
3. *Shift-equivalent reduction [5]* Rule sequences s and s' are shift-equivalent (as defined in Sect. 3.2).
4. *Summary reduction [5]* Rule sequence s' can be deduced from s by summarizing neighbored rules in s into concurrent rules.

Note that since the reduced rule sequence s' satisfies the applicability criteria, it is in particular applicable to G_0.

4 Consistency Analysis with Refined Activity Diagrams

Activities are defined more precisely by pre- and post-conditions as explained in Sect. 2. They can be formalized by graph transformation rules 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 present reduction techniques enabling the efficient analysis of refined activity diagrams for consistency.

4.1 Refined Activity Diagrams: Semantics and Consistency

As presented in [12, 9], we restrict our considerations to well-structured activity diagrams. The building blocks are sequences, fork-joins, decisions, 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:

- Empty: An empty activity block is not depicted.
- 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 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 is empty, $S_B = \emptyset$.
- 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 $xs'_x || ys'_y = \{x\} \ seq \ s'_x || ys'_y \cup \{y\} \ seq \ xs'_x || s'_y$.

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 $\mathcal{S}$ such that each rule sequence in $Sem(A)$ is applicable to at least one of the graphs in $\mathcal{S}$. Such a non-empty set of graphs $\mathcal{S}$ making an activity diagram A consistent is *without junk*, if for each graph in $\mathcal{S}$ at least one rule sequence in $Sem(A)$ exists such that it is applicable to this graph. Each graph in $\mathcal{S}$ without junk then represents a potential snapshot of the system to which an activity sequence in A can be applied consistently.

Finally, note that activity diagrams considered in this paper do not contain object flow. Integrating object flow is part of future work and promises to enable a more precise analysis. For now we restrict to *maximal object flow*. This means that each activity (resp. its corresponding rule) is expected to reuse as much as possible objects delivered or passed through by the former activity (resp. rule).

4.2 Analysis of Reduced Semantics

Based on the semantics and consistency definition of an activity diagram A we apply the criteria for checking the applicability of rule sequences given in Sect. 3 in order

to check for consistency. The reductions presented in Sect. 3.3 allow to cut down the number of sequences in $Sem(A)$.

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 to each sequence loop reduction, repeated element reduction, shift-equivalent reduction or summary reduction[5] (as defined in Sect. 3.3). It follows directly from Theorem 1, Theorem 2 and [5] that if all sequences in $Red(A)$ satisfy the applicability criteria, then all sequences in $Sem(A)$ are applicable.

4.3 Tool support

Formalizing activity diagrams by graph transformation has not only the advantage that consistent behavior modelling can be precisely defined, but also offers tool support for checking conflicts and dependencies of rule applications and moreover, applicability checks. The algebraic approach to graph transformation is supported by AGG [8], a tool environment which consists of a graph transformation engine, analysis tools and a graphical user interface for convenient user interaction. Here, we concentrate on conflicts and dependency as well as applicability checks.

To check conflicts and dependencies between rules, all corresponding critical pairs can be computed by AGG. A critical pair represents the conflict (resp. dependency) of rules in a minimal context. The minimal conflicts (resp. dependencies) are represented in a conflict (resp. dependency) matrix. See Figures 6 and 8 showing conflicts and dependencies for all rules belonging to activity diagram *Create master data* (resp. *Take order*). The entry numbers within these matrices indicate how many minimal conflicts and dependencies, resp. have been found. More precisely, entry (r_j, r_i) (row, column) in the conflict matrix in AGG describes all direct transformations $G \overset{(r_i,m_i)}{\Rightarrow} H_i$ which cause a conflict with $G \overset{(r_j,m_j)}{\Rightarrow} H_j$ in a minimal context. Entry (r_j, r_i) in the dependency matrix in AGG describes all two-step transformation sequences $G \overset{(r_j,m_j)}{\Rightarrow} H_j \overset{(r_i,m_i)}{\Rightarrow} G'$ such that the second graph transformation is dependent on the first one in a minimal context. The conflict and dependency matrices can be used to further check semi-automatically, if a rule sequence fulfills the applicability criteria presented in Sect. 3 as explained in detail in the next section for the example given in Sect. 2.

AGG [8] moreover features a new analysis module checking the applicability criteria given in Sect. 3 for a given graph and rule sequence automatically. For a given graph transformation system several rule sequences can be created which may be tested then for applicability. In Fig. 5 a screenshot of this analysis module in AGG is depicted showing that two sequences have been tested successfully for applicability. The result of this check is shown by a table reporting how far the criteria are fulfilled (resp. not fulfilled). See Fig. 7 and 9 in the next section showing a table for rule sequence *Add pizza kind Add topping kind Add beverage kind* (resp. *Create customer Create order Order beverage Close order*). E.g. if a rule has a purely enabling predecessor in the

[5] Note that often it is desirable to apply more than one of the above reductions successively to the same sequence. Such kind of stronger reductions are important for automating consistency analysis and part of ongoing work.

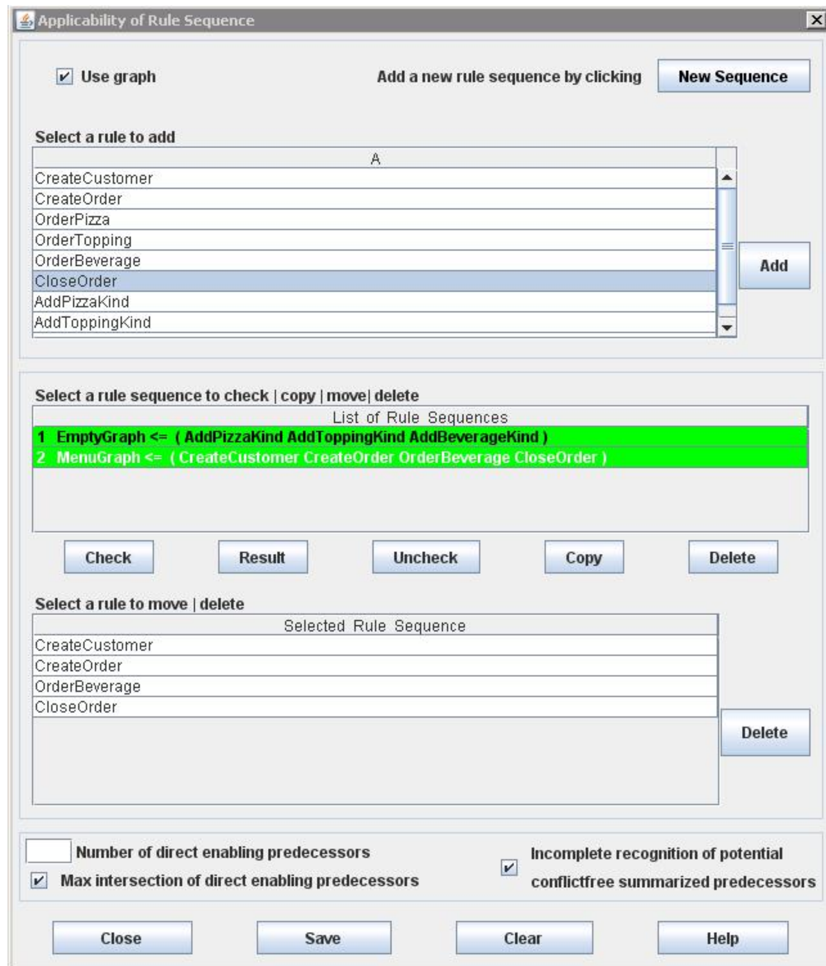


Fig. 5. Checking applicability criteria in AGG.

sequence, the name of this purely enabling predecessor is depicted. That way the user gets a chance to understand why a rule sequence is applicable.

The implementation of this new analysis module is still a prototypical one. It enables the automatic check of rule sequences entered by the user, but it also features an XML-based import of lists of rule sequences to be tested (resp. XML-based export of the results for each rule sequence). This is important to allow for integration with a CASE tool generating automatically e.g. the lists of sequences to be checked for applicability. However tool integration with some CASE tool is still part of future work.

5 Example Analysis

Consider the pizzeria scenario in Sect. 2. First we analyze the use cases "create master data" and "take order" separately. Then we reason about the interrelation of both. In order to analyze the use cases the following steps are carried out: (1) Generate potential conflicts and dependencies of activity rules. (2) Examine all rule sequences defined by the activity diagram, and analyze their applicability according to criteria[6] by at first possibly reducing them. (3) Draw conclusions regarding the activities or activity diagrams, respectively.

5.1 Create Master Data

We consider at first use case "create master data" and its activity diagram (cf. Fig. 2(a)). In order to be able to reason in a compact way about activity sequences, we use the following acronyms: AP=Add pizza kind, AT=Add topping kind, AB=Add beverage kind. Fig. 6 depicts the conflict and dependency matrices calculated by the tool AGG. The input data are rules corresponding to the pre- and post-conditions in Fig. 3. Note that each of these rules is parameterized. Consider e.g. rule *AP(n:String, p:float)*. It holds a parameter for the pizza name and another one for its price. This rule can only be applied if concrete values for its parameters are given. E.g. rule *AP("Margherita", 5)* can be applied whenever a Margherita pizza has not already been added. Otherwise, a produce-forbid conflict occurs which is shown in the conflict matrix computed by AGG. In particular, there is a conflict between *AP(n:String, p:float)* and *AP(n':String, p':float)* if the parameter values for *name* i.e. *n* and *n'* are the same. This holds analogously for the rules *AT* and *AB*. In all other cases there are neither conflicts nor dependencies. Considering again the activity diagram in Fig. 2(a), we start checking the applicability

Minimal Conflicts

first \ second	1: AddPizzaKind	2: AddToppingKind	3: AddBeverageKind
1: AddPizzaKind	1	0	0
2: AddToppingKind	0	1	0
3: AddBeverageKind	0	0	1

Minimal Dependencies

1: AddPizzaKind	2: AddToppingKind	3: AddBeverageKind
0	0	0
0	0	0
0	0	0

Fig. 6. Conflict and dependency matrix of "Create master data"

of all sequences in *Sem(Create master data)* according to the criteria. The first criterion to check is *initialization*. Its satisfaction depends on the first rule in each sequence only. Since each sequence in Sem(Create master data) starts with AP, AT or AB, by checking these three cases we cover the criterion for all sequences. Each rule's left-hand side (LHS) is empty i.e. no requirements concerning the existence of objects in the start graph are given. However, considering their negative application conditions

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

(NAC) and the reasoning above, we conclude that the start graph may be empty or at least must not contain an object forbidden by a corresponding NAC, i.e. containing an object with a name according to the rule's parameter. Since all sequences contain rules with empty left-hand sides only, criterion *no node deleting* is obviously satisfied as well. As mentioned above there is one conflict between each rule with itself if their parameter values for *name* are the same. Thus all sequences with at most one occurrence of each rule satisfies the criterion *no impeding predecessor*. Otherwise, the parameters for attribute *name* of equal rules must differ in order to fulfill the criterion. The last criterion to check is *enabling predecessor(s)*. Since the dependency matrix in Fig. 6 does not show any dependencies, each rule after the first rule should be applicable to the start graph. Thus each rule fulfills the sub-item *not needed* of the enabling predecessor criterion. Summarizing, the refined activity diagram "Create master data" is consistent, if multiple occurring equal rules within a sequence do not use the same parameter values for their attribute *name* and the start graphs do not already contain objects forbidden by some rule in the sequence. If both constraints hold, the loops in the activity diagram can be run through as long as desired. Therefore pizza, topping and beverage kinds cannot occur twice with the same name in the menu.

Fig. 7 depicts a table computed by AGG for rule sequence *Add pizza kind, Add topping kind, Add beverage kind* and an empty start graph. This table shows that all applicability criteria are fulfilled for this rule sequence as explained step by step for all rule sequences above.

Applicability of Rule Sequence at Graph: EmptyGraph

Rule / Criteria	(1) initialization	(2) no node-deletin...	(3) no impeding pr...	(4a) pure enabling...	(4b) direct enablin...	(4c) not needed
AddPizzaKind						
AddToppingKind						(applicable)
AddBeverageKind						(applicable)

Close

Fig. 7. Applicability table for rule sequence *AP, AT, AB* and empty start graph.

5.2 Take Order

Now we consider use case "take order" and its activity diagram (cf. Fig. 2(c)). 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. We use acronyms again in the sequences for better readability: OP=Order pizza, OT=Order topping, OB=Order beverage, CC=Create customer, CO=Create order, CL=Close order. At first we take a look at the dependency and conflict matrices calculated by AGG (cf. Fig. 8). Irrelevant[7] entries have been colored gray. This is conceivable to be done automatically in the future. Considering again the activity diagram in Fig. 2(c), we start checking the applicability of all sequences in

[7] First only conflicts caused by (resp. dependencies on) some predecessor are relevant.

Minimal Conflicts

Export

first \ second	1: Crea...	2: Crea...	3: Ord...	4: Ord...	5: Ord...	6: Clos...
1: CreateCustomerEntity	1	0	0	0	0	0
2: CreateOrderEntity	0	1	0	0	0	0
3: OrderPizza	0	0	0	0	0	0
4: OrderTopping	0	0	0	0	0	0
5: OrderBeverage	0	0	0	0	0	0
6: CloseOrder	0	0	1	1	1	1

Minimal Dependencies

1: Creat...	2: Creat...	3: Orde...	4: Orde...	5: Orde...	6: Clos...
0	1	0	0	0	0
0	0	1	0	1	1
0	0	0	1	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Fig. 8. Conflict and dependency matrix of "Take order" with respect to the control flow

Sem(Take order) according to the criteria. We start our analysis with criterion *initialization*. Each sequence starts with CC or CO. For sequences starting with $CO(n)$ each start graph should have a *Customer* object named according to parameter n and additionally this object must not be linked to an *Order* with attribute *closed* being *false*. For sequences starting with $CC(n, adr)$ the start graph must not contain a *Customer* object named according to parameter n of rule CC. This ensures that two customers with the same name do not exist. Criteria *no node deleting* and *no impeding predecessor* are satisfied for all sequences, since no node is deleted by any rule and the conflict matrix (cf. Fig. 8) 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. For rule sequences starting with CC, rule CO is the first one to be enabled. Now $CO(n2)$ is purely enabled by the preceding rule $CC(n1, adr1)$ if $n1 = n2$. By assuming maximal object flow (as explained in Sect. 4.1) it is clear that the customer which is created by CC is the one for which an order is created by CO. Moreover rule CL is purely enabled, namely by rule CO. Thus only for OP, OT and OB in the activity diagram we still have to check if they have enabling predecessor(s). Therefore at first we apply *summary reduction* (as defined in Sect. 3.3) to each OP with its corresponding successive OT's. Thus, each ordering of a pizza together with its toppings can be summarized in a concurrent rule which we call OPT. Independent of the number of toppings rules OB and OPT are not in conflict with each other and also not with themselves. This means that some pizza with toppings or beverage can be ordered without disabling the ordering of other items. Thus we conclude that each OPT as well as each OB has CO (resp. the sequence CC followed by CO for the case that a customer needs to be created) as nearest enabling predecessor(s)[8], since it delivers the *Order* to which the pizza, topping, and beverage items should be added. Consequently the start graph should contain the corresponding *PizzaKind*, *ToppingKind* and *BeverageKind* objects in addition. Summarizing, *Sem(Take order)* contains two classes of sequences appli-

[8] Note that when checking the last criterion *enabling predecessor(s)* we fixed the dependencies between CC,CO and OPT (resp. OB). This means that CO creates an order for some customer created by CC, and OPT (resp. OB) adds a pizza (resp. beverage) to the order for the same customer (instead of for some other customer). Therefore the compatibility condition defined in Lemma 3 in the appendix is fulfilled.

cable to two different kinds of start graphs. The first one contains sequences without rule CC applicable to start graphs that already hold a *Customer* object. The other one contains sequences starting with rule $CC(n, adr)$ applicable to start graphs not already holding a *Customer* object with name n. Moreover the *PizzaKind*, *ToppingKind* as well as *BeverageKind* of ordered items should be available in the start graph.

Fig. 9 depicts a table computed by AGG for rule sequence *Create customer, Create order, Order beverage, Close order* and as start graph a menu graph in which the ordered *BeverageKind* is present. This table shows that all applicability criteria are fulfilled for this rule sequence as explained step by step for all rule sequences above.

Applicability of Rule Sequence at Graph: MenuGraph

Rule / Criteria	(1) initialization	(2) no node-del...	(3) no impedin...	(4a) pure enabling ...	(4b) direct enabling predeces...	(4c) not needed
CreateCustomer						
CreateOrder				CreateCustomer		
OrderBeverage					CreateCustomer-CreateOrder	
CloseOrder				CreateOrder		

Close

Fig. 9. Applicability table for rule sequence *CC, CO, OB, CL* and menu start graph.

As result we reveal the applicability of all given sequences in *Sem(Create master data)* and *Sem(Take order)* with respect to specific sets of start graphs. This implies that the activity diagrams are consistent with respect to corresponding system snapshots. Finally we observe that almost every rule sequence of use case "take order" requires objects of type *PizzaKind*, *ToppingKind* and *BeverageKind* which are created by use case "create master data". This is meaningful since the pizzeria staff has to provide product information before a customer can order food and beverages. Only sequences $< CO, CL >$ and $< CC, CO, CL >$ do not require such objects as they do not order anything.

6 Related Work

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

Eshuis [13] 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 [14] defines a denotational semantics for control flow of UML 2.0 activity diagrams including procedure calls by means of petri nets. The standard petri net analysis provides analysis of generic properties, e.g. reachability or deadlock freeness. In [12, 15] a graph transformation based semantics is proposed for activity diagrams. This semantics is then analyzed by a graph transformation based model checker. Moreover activity diagrams with such a semantics are translated into a textual programming language having also a graph transformation

based semantics. Correctness of the transformation can then be shown based on trace equivalence. The activities are not refined by pre- and post- conditions.

We concentrate in this paper on the analysis of compatibility of control flow with refinement of activities by pre- and post- conditions. Our approach is dedicated to checking this compatibility which we called consistency. We provide sufficient criteria for statically checking if each run in the activity diagram can be completed. 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 use this refinement during the analysis.

Jayaraman et al. [16] 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 are considered for the analysis.

Fujaba [17] and Moflon [18] 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 critical pair analysis and the applicability checks on rule sequences as proposed 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 by analyzing sets of rule sequences in a static way.

Although the given criteria seem to apply often, there are applicable graph transformation rule sequences which cannot be examined by this new kind of static analysis. In that case, we can try to adapt our activity diagrams to the criteria, if sensible. Alternatively, stronger reduction mechanisms (their development is part of ongoing work) as the ones introduced in this paper should enable consistency analysis of a wider range of activity diagrams. Otherwise, a validation technique on execution level such as model checking should be used. For the time being, we assume that all rules do not delete object nodes. This criterion seems to be the most restricting one, however, it can be circumvented by just detaching an object node from the main object structure if it shall be deleted. For example, in use case “delete beverage order” an ordered beverage should be canceled. The corresponding *OrderItem* would be detached from its *Order*. Besides the small presented example of a pizza service, our criteria have been applied to the modeling of services and service orchestration [6] and need further evaluation in the future.

The 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

8.1 Graph Transformation with NACs and Parallel Independence

We repeat the basic definitions for double pushout graph transformation with negative application conditions (NACs). A graph rule holding a NAC n can be applied on a graph G only if the forbidden structure expressed by n is not present in G.

Definition 1 (graph, graph morphism, rule). *A* graph $G = (G_E, G_V, s, t)$ *consists of a set G_E of edges, a set G_V of vertices and two mappings $s, t : G_E \to G_V$, assigning to each edge $e \in G_E$ a source $q = s(e) \in G_V$ and target $z = t(e) \in G_V$. A* graph morphism *(short* morphism*)* $f : G_1 \to G_2$ *between two graphs $G_i = (G_{i,E}, G_{i,V}, s_i, t_i)$, $(i = 1, 2)$ is a pair $f = (f_E : G_{E,1} \to G_{E,2}, f_V : G_{V,1} \to G_{V,2})$ of mappings, such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. A morphism $f : G_1 \to G_2$ is injective (resp.surjective) if f_V and f_E are injective (resp. surjective) mappings. A graph transformation* rule $p : L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ *consists of a rule name p and a pair of injective morphisms $l : K \to L$ and $r : K \to R$. The graphs L, K and R are called the left-hand side (LHS), the interface, and the right-hand side (RHS) of p, respectively.*

Definition 2 (rule and transformation with NACs, applicability of rule with NACs).

- *A* negative application condition *or $NAC(n)$* on p *for a rule $p : L \stackrel{l}{\leftarrow} K \stackrel{r}{\rightarrow} R$ (l,r injective) is an arbitrary morphism $n : L \to N$. A morphism $g : L \to G$* satisfies *$NAC(n)$ on L, written $g \models NAC(n)$, if and only if $\nexists\, q : N \to G$ injective such that $q \circ n = g$.*

$$\begin{array}{ccc} L & \xrightarrow{n} & N \\ {\scriptstyle g}\downarrow & \swarrow_{q} \!\!\!\times & \\ G & & \end{array}$$

A set of NACs on p is denoted by $NAC_p = \{NAC(n_i) | i \in I\}$. A morphism $g : L \to G$ satisfies *NAC_p if and only if g satisfies all single NACs on p i.e. $g \models NAC(n_i)\ \forall i \in I$.*

– *A* rule (p, NAC_p) with NACs *is a rule with a set of NACs on* p*.*
– *A* direct transformation $G \overset{p,g}{\Rightarrow} H$ *via a rule* $p : L \leftarrow K \rightarrow R$ *with* NAC_p *and a match* $g : L \rightarrow G$ *consists of the double pushout [19] (DPO)*

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ {\scriptstyle g}\downarrow & & \downarrow & & \downarrow{\scriptstyle h} \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

where g *satisfies* NAC_p*, written* $g \models NAC_p$*. Since pushouts along injective morphisms always exist, the DPO can be constructed if the pushout complement of* $K \rightarrow L \rightarrow G$ *exists. If so, we say that the match* g *satisfies the gluing condition of rule* p*. If there exists a morphism* $g : L \rightarrow G$ *which satisfies the gluing condition and* $g \models NAC_p$ *we say that rule* p *is* applicable *on* G *via the match* g*.*

Moreover we reintroduce[5, 19] the following definitions formalizing what it means for a rule r_1 to *be not in conflict* (resp. *cause no conflict*) with some other rule r_2.

Definition 3 (parallel independent transformations and rules). *Two direct transformations* $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$ *with* NAC_{r_1} *and* $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ *with* NAC_{r_2} *are* parallel independent *if*

$$\exists h_{12} : L_1 \rightarrow D_2 \; s.t. \; (d_2 \circ h_{12} = m_1 \; and \; e_2 \circ h_{12} \models NAC_{r_1})$$

and

$$\exists h_{21} : L_2 \rightarrow D_1 \; s.t. \; (d_1 \circ h_{21} = m_2 \; and \; e_1 \circ h_{21} \models NAC_{r_2})$$

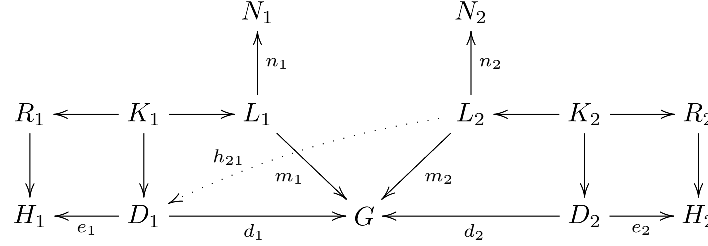
as in the following diagram:

$$\begin{array}{ccccccccccc} & & & & N_1 & & N_2 & & & & \\ & & & & \uparrow{\scriptstyle n_1} & & \uparrow{\scriptstyle n_2} & & & & \\ R_1 & \longleftarrow & K_1 & \longrightarrow & L_1 & & L_2 & \longleftarrow & K_2 & \longrightarrow & R_2 \\ \downarrow & & \downarrow & {\scriptstyle h_{21}} & {\scriptstyle m_1}\searrow & & \swarrow{\scriptstyle m_2} & {\scriptstyle h_{12}} & \downarrow & & \downarrow \\ H_1 & \underset{e_1}{\longleftarrow} & D_1 & \underset{d_1}{\longrightarrow} & & G & & \underset{d_2}{\longleftarrow} & D_2 & \underset{e_2}{\longrightarrow} & H_2 \end{array}$$

The rules r_1 *and* r_2 *are* parallel independent *if every transformation* $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$ *via* r_1 *with* NAC_{r_1} *and any other transformation* $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ *via* r_2 *with* NAC_{r_2} *are parallel independent. We say also that* r_1 is not in conflict with r_2.

Definition 4 (asymmetrically parallel independent transformations). *A direct transformation* $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ *with* NAC_{r_2} *is* asymmetrically parallel independent *on* $G \overset{(r_1,m_1)}{\Longrightarrow}$

H_1 *with* NAC_{r_1} *if* $\exists h_{21} : L_2 \rightarrow D_1 : d_1 \circ h_{21} = m_2$ *such that* $e_1 \circ h_{21} \models NAC_{r_2}$.



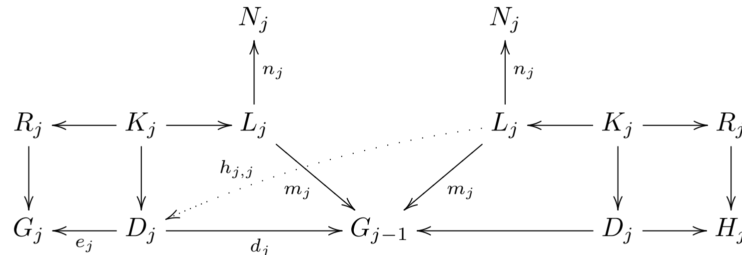
Definition 5 (asymmetrically parallel independent rules). *The rule* r_2 *is* asymmetrically parallel independent *on* r_1 *if every transformation* $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ *via* r_2 *with* NAC_{r_2} *is asymmetrically parallel independent on any other transformation* $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$ *via* r_1 *with* NAC_{r_1}. *We say also that* r_1 causes no conflict with r_2.

8.2 Repeated Element and Loop Reduction

The following lemmata are used in order to prove correctness of the *repeated element* and *loop reduction*. Note that we assume *injective matching* in all following lemmata and theorems.

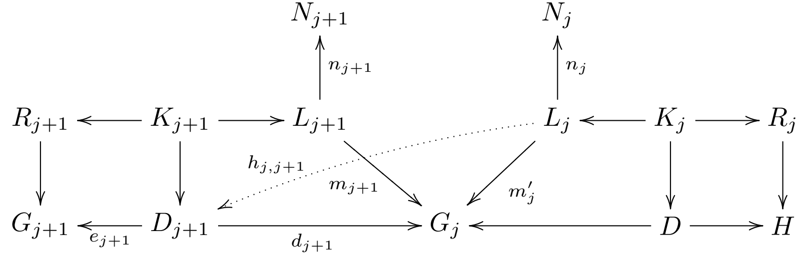
Lemma 1. *Given a transformation sequence* $t : G_0 \overset{r_1}{\Rightarrow} G_1 \ldots G_{n-1} \overset{r_n}{\Rightarrow} G_n$ *via the rule sequence* $r_1 r_2 \ldots r_n$. *Then* r_j *with* $j \leq n$ *is applicable to* G_n *if* r_j *is not node-deleting and* r_i *with* $j \leq i \leq n$ *causes no conflict with* r_j.

Proof. Let $G_0 \overset{r_1}{\Rightarrow} G_1 \ldots G_{n-1} \overset{r_n}{\Rightarrow} G_n$ be the graph transformation sequence arising by applying rule sequence $r_1 r_2 \ldots r_n$ to G_0. Consider then the following diagram:



Since r_j causes no conflict with r_j the morphism $m'_j = e_j \circ h_{j,j}$ exists satisfying NAC_{r_j}. This makes r_j applicable to G_j. Therefore we can construct the following

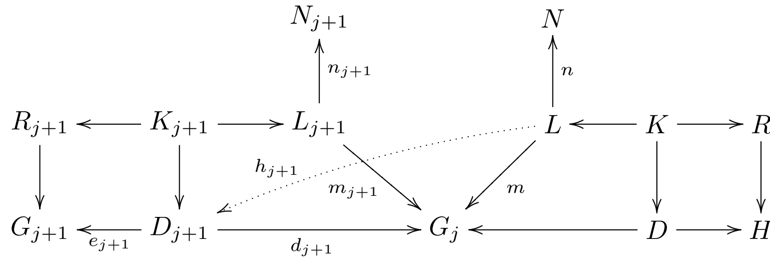
diagram which exists because r_j is not node-deleting:



Since r_{j+1} does not cause a conflict with r_j the morphism $e_{j+1} \circ h_{j,j+1}$ exists satisfying NAC_{r_j}. This makes r_j applicable to G_{j+1}. We can iterate this argumentation up to the conclusion that r_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 $G_{j-1} \overset{r_j}{\Rightarrow} G_j \dots G_{n-1} \overset{r_n}{\Rightarrow} G_n$ and rule r_j. This is because r_j is not node-deleting, r_j does not have impeding predecessors in $r_j \dots r_n$ and r_j is applicable to G_{j-1}. Therefore we can conclude that the graph transformation sequence $G_0 \overset{r_1}{\Rightarrow} G_1 \dots G_{n-1} \overset{r_n}{\Rightarrow} G_n \overset{r_j}{\Rightarrow} G_{n+1}$ exists and thus $r_1 r_2 \dots r_n r_j$ is applicable to G_0 as well.

Lemma 2. *Given a transformation sequence $t : G_0 \overset{r_1}{\Rightarrow} G_1 \dots G_{n-1} \overset{r_n}{\Rightarrow} G_n$ via the rule sequence $r_1 r_2 \dots r_n$. Then r is applicable to G_n whenever r is applicable to some intermediate graph G_j with $0 \leq j \leq n$ in the transformation sequence t and each r_i with $j+1 \leq i \leq n$ causes no conflict with r and r is not node-deleting.*

Proof. Let G_j with $1 \leq j \leq n$ be the intermediate graph in t to which r is applicable via match m. Consider now the graph transformation sequence $t' : G_j \Rightarrow^* G_n$ via $r_{j+1} \dots r_n$ arising by cutting off the first j steps of transformation sequence t. Consider then the following diagram which can be constructed because r is not node-deleting:



Since r_{j+1} does not cause a conflict with r the morphism $e_{j+1} \circ h_{j+1}$ exists satisfying NAC_r. This makes r applicable to G_{j+1}. We can iterate this argumentation up to the conclusion that r is applicable to G_n and in the end a graph transformation sequence $t'' : G_0 \Rightarrow^* G_n \overset{r}{\Rightarrow} G_{n+1}$ exists via $r_1 r_2 \dots r_n r$.

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} is equal to r_i, r_{i+1} is not node-deleting since s' fulfills the applicability criteria, and rule r_i does not cause a conflict with r_{i+1}. The latter condition holds since 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 argue analogously for all r_j with $n \geq j > n + 2$.

Theorem 2 (loop reduction). *Given a rule sequence $s : q(r_1 r_2 \ldots r_m)^n q'$ consisting of a rule sequence q followed by an n-ary loop $(n > 2)$ of rule sequence $r_1 r_2 \ldots r_m$ followed by a rule sequence q'. Sequence s is applicable to G_0 if sequence $s' : q(r_1 r_2 \ldots r_m)^2 q'$ being s with $r_1 r_2 \ldots r_m$ 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 $q(r_1 r_2 \ldots r_m)^2$*
  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 $(r_1 r_2 \ldots r_m)^2$ to H and J the graph obtained after applying q' to I. Since s starts as s' with the rule sequence $q(r_1 r_2 \ldots r_m)^2$ we can simply adopt the first part of the graph transformation sequence $G_0 \Rightarrow^* H \Rightarrow^* I$. Now we have to argue that the remaining rule sequence of s $(r_1 r_2 \ldots r_m)^{n-2} q'$ 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 argue that q' is applicable to K. We argue as follows. Let q'_1 be 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 argue analogously for all remaining rules in q' and conclude that rule sequence s is applicable to G_0.

8.3 Improving the Enabling Predecessor Criterion

At first we reintroduce [10] the definition of a concurrent rule r_c of a sequence of rules $r_0, \dots, r_n$. We only repeat the definition of a concurrent rule without NACs. For the construction of concurrent NACs we refer to [11].

Definition 6 (concurrent rule, lhs-match). *Given a sequence of rules $r_0, \dots, r_n$.*

- *The* concurrent rule *(resp.* lhs-match*) of r_0 is r_0 (resp. $id : L_0 \to L_0$).*
- *Consider $r'_c : L'_c \leftarrow K'_c \to R'_c$ a concurrent rule of $r_0, \dots, r_{n-1}$. Let $(e'_c : R'_c \to E, e_n : L_n \to E)$ be a pair of jointly surjective morphisms as shown in Fig. 10 such that (1) is a pullback and all other squares are pushouts. A* concurrent rule *$r_c = r'_c *_E r_n$ of $r_0, \dots, r_n$ is defined by $r_c = L_c \overset{l \circ k_c}{\leftarrow} K_c \overset{r \circ k_n}{\rightarrow} R_c$. The* lhs-match *of $r_c = r'_c *_E r_n$ is defined by $l_c : L'_c \to L_c$.*

Remark 1. Note that in general more than one concurrent rule for a sequence of rules exists due to the freedom to choose (e'_c, e_n).

The following definition of compatible concurrent matches ensures that when matching a concurrent rule to some graph G_0 the same match is continued to use when extending this concurrent rule.

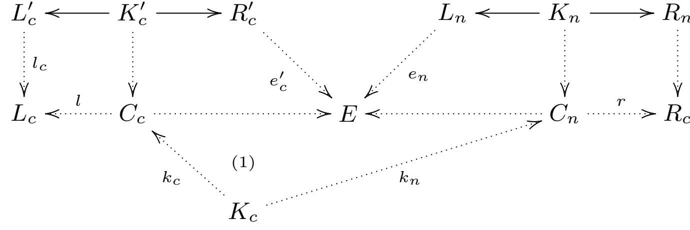
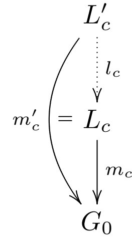


Fig. 10. Definition of concurrent rule

Definition 7 (compatible concurrent matches). *Given a concurrent rule $r_c = r'_c *_E r_n$ of a sequence of rules $r_0, \ldots, r_n$ as defined in Def. 6 and matches $m_c : L_c \to G_0$ and $m'_c : L'_c \to G_0$ from the lhs of r_c (resp. r'_c) into G_0. Let l_c be the lhs-match of r_c. Then m_c and m'_c are* compatible *if $m_c \circ l_c = m'_c$ as shown on the right.*



We introduce the definition of *purely dependent rules* needed for defining the enabling predecessor criterion.

Definition 8 (purely dependent rules). *A rule $r_2 : L_2 \leftarrow K_2 \to R_2$ is* purely dependent *on $r_1 : L_1 \leftarrow K_1 \to R_1$ if there exists an injective morphism $l_{21} : L_2 \to R_1$.*

Note that compared to the definition of *purely dependent rules* in [5] here r_2 is allowed to have NACs.

Theorem 3 (improved form of enabling predecessors). *Thm.3.2 (correctness of applicability criteria) in [5] still holds, if criterion enabling predecessor in Def.3.1 in [5] is replaced by the following improved enabling predecessor(s) criterion:*
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. We say that r is purely enabled by r'. OR*

nearest *there exists a concurrent rule r_c of rule r and a (sequence of) predecessor rule(s) s' such that r_c is applicable via m_{r_c} to G_0, and all predecessors of r_c do not cause a conflict with r_c. Moreover r is not in conflict with each rule between s' and r. We say that r is nearest enabled by s' via r_c and m_{r_c}. OR*

not needed *r itself is applicable to G_0 via some match $m_r : L \to G_0$. We say that r is self enabled via m_r.*

In addition the following compatibility condition *should hold in order to fix dependencies between rules while checking the nearest enabling predecessors criterion: If rule r is nearest enabled by a single rule r', then r' was self enabled via some m_r. In addition $r_c = r' *_E r$ and $m_{r'}$ and m_{r_c} are compatible as in Def. 7. If rule r is nearest enabled by a sequence of rules $s' : r'_1 r'_2 \dots r'_j$ with $j > 1$, then r'_j was directly enabled by $r'_1 r'_2 \dots r'_{j-1}$ via the concurrent rule r'_c of s' and $m_{r'_c}$. In addition $r_c = r'_c *_E r$ and $m_{r'_c}$ and m_{r_c} are compatible as in Def. 7.*

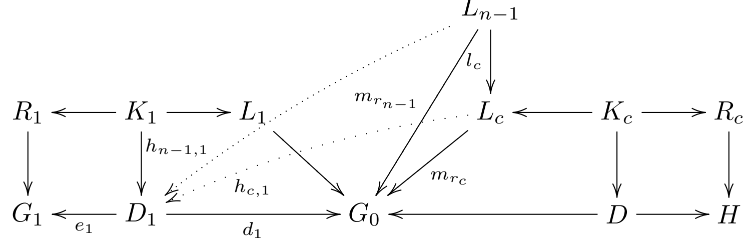
Remark 2. Please note that the differences of the improved enabling predecessor criterion in this paper with the enabling predecessor criterion in [5] are underlined and also that the *compatibility condition* is new. Note moreover that if r fulfills the nearest enabling predecessor criterion without any intermediate rule between s' and r, then we say that r is *directly enabled*.

Proof. **pure** Suppose that the sequence of all predecessors of r in s is applicable to G_0. Because of the correctness proof in [5] it can be proven that then also the sequence of all predecessors of r in s followed by r itself is applicable to G_0, provided that r does not hold any NACs. We denote the last direct transformation, when applying this sequence to G_0, by $G \overset{r,m}{\Rightarrow} H$ with $m : L \rightarrow G$ being the match morphism . In our improved pure enabling predecessor criterion now though rule r holds some NAC. Therefore we show that this NAC is in any case satisfied under the extra assumption that each NAC of r forbids a graph element e of type t such that an element of type t is neither present in G_0 nor produced by any predecessor of r. Suppose that this NAC on r is not satisfied. Then there exists some $NAC(n) : L \rightarrow N$ and some morphism $q : N \rightarrow G$ such that $q \circ n = m$. Therefore q should map the element e of type t to some element e' of type t belonging to graph G. This is a contradiction since the predecessor rules of r never produced any element of type t, nor some element of type t was already present in graph G_0.

nearest Note that there are two differences between the direct enabling predecessor criterion in Def.3.1. in [5] and the nearest enabling predecessor(s) criterion above. The first one is that more than one enabling predecessor in a row is allowed. The second difference is that enabling predecessor(s) should not necessarily be direct, but rules can occur in-between if these intermediate rules are parallel independent with r.

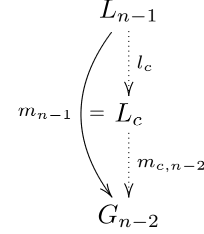
- We prove at first that more than one enabling predecessor is allowed. Consider in Thm.3.2 the induction step (ii) for criterion (4b) and change it as follows. The induction hypothesis says that the transformation $t : G_0 \overset{r_1}{\Rightarrow} G_1 \dots \overset{r_{n-1}}{\Rightarrow} G_{n-1}$ exists. In particular, the shorter transformation $G_0 \overset{r_1}{\Rightarrow} G_1 \dots G_{n-j-1}$ exists. By assumption there exists a concurrent rule r_c of $r_{n-j}, r_{n-j+1}, \dots, r_n$ with $j > 0$ such that r_c is applicable to G_0. Moreover each predecessor of r_c does not cause any conflict with r_c. Because of Lemma 2 it follows that the transformation sequence $G_0 \overset{r_1}{\Rightarrow} G_1 \dots G_{n-j-1} \overset{r_c, m_{c,n-j-1}}{\Rightarrow} G_n$ exists. Because of the Concurrency Theorem with NACs in [11] then also a graph transformation sequence $t' : G_0 \overset{r_1}{\Rightarrow} G_1 \dots \overset{r_{n-1}}{\Rightarrow} G'_{n-1} \overset{r_n}{\Rightarrow} G_n$ exists. We now prove that graphs $G'_i = G_i$ for $n - j \leq i < n$ because of the compatibility condition. We have the following two cases:

$j = 1$ In this case the compatibility condition states that r_{n-1} is self enabled and $r_c = r_{n-1} * r_n$ with compatible matches $m_{r_{n-1}}$ and m_{r_c}. Consider in this case the following diagram:



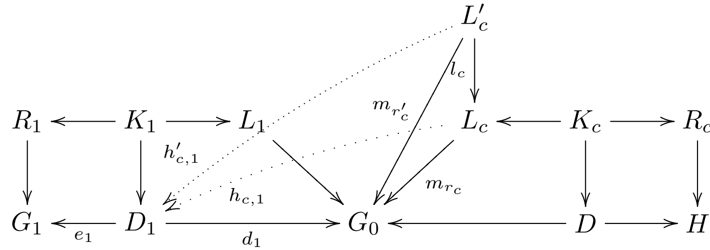
The match of L_c into G_1 is equal to $e_1 \circ h_{c,1}$. The match of L_{n-1} into G_1 is equal to $e_1 \circ h_{n-1,1}$. We now prove that these matches are compatible i.e. $e_1 \circ h_{n-1,1} = e_1 \circ h_{c,1} \circ l_c$. This is true if we can prove that $h_{n-1,1} = h_{c,1} \circ l_c$. Lemma 1 in [20] states that given direct transformations $H_1 \stackrel{r_1}{\Leftarrow} H \stackrel{r_2}{\Rightarrow} H_2$ and an extended match allowing for the application of r_1 (resp. r_2) to H_2 (resp. H_1) then this match is unique.

Therefore it holds that $h_{n-1,1}$ is a unique morphism such that $d_1 \circ h_{n-1,1} = m_{r_{n-1}}$. Now we can prove that also $d_1 \circ h_{c,1} \circ l_c = m_{r_{n-1}}$. This is because $m_{r_{n-1}} = m_{r_c} \circ l_c$ (compatibility condition) and $m_{r_c} = d_1 \circ h_{c,1}$. Continuing this argumentation iteratively, it follows that $m_{c,n-2} : L_c \to G_{n-2}$ is compatible with $m_{n-1} : L_{n-1} \to G_{n-2}$ as depicted in the diagram on the right.



Thereafter it follows that $G_{n-2} \stackrel{r_c, m_{c,n-2}}{\Rightarrow} G_n$ can be decomposed into $G_{n-2} \stackrel{r_{n-1}, m_{n-1}}{\Rightarrow} G_{n-1} \stackrel{r_n, m_n}{\Rightarrow} G_n$.

$j > 1$ In this case the compatibility condition states that r_{n-1} is directly enabled by $r_{n-j}, r_{n-j+1}, \dots r_{n-2}$ via (r'_c, m'_c). Consider in this case the following diagram:



We now argument analogously to the case $j = 1$ for the matches $m_{r'_c}$ (instead of $m_{r_{n-1}}$) and m_{r_c}. We can then deduce that $m'_{c,n-j-1}$ the match of r'_c into G_{n-j-1} is compatible with $m_{c,n-j-1}$ the match of r_c into

G_{n-j-1}. Therefore the concurrent transformation $G_{n-j-1} \overset{r_c, m_{c,n-j-1}}{\Rightarrow} G_n$ can be decomposed into $G_{n-j-1} \overset{r'_c, m'_{c,n-j-1}}{\Rightarrow} G_{n-1} \overset{r_n}{\Rightarrow} G_n$. Because r_{n-1} on its part was enabled via $(r'_c, m_{r'_c})$ we know that $G_{n-j-1} \overset{r'_c, m'_{c,n-j-1}}{\Rightarrow} G_{n-1}$ can be decomposed into the original transformation subsequence $G_{n-j-1} \overset{r_{n-j}}{\Rightarrow} G_{n-j} \dots G_{n-2} \overset{r_{n-1}}{\Rightarrow} G_{n-1}$.

- We now prove that enabling predecessor(s) need not necessarily be direct as described above. Given some intermediate rules $p_1, p_2, \dots p_m$ with $m \geq 1$ between r_{n-1} and r_n such that each of them is parallel independent with r_n. Again we prove this improved criterion by induction. Suppose that $t : G_0 \overset{r_1}{\Rightarrow} G_1 \dots \overset{r_{n-1}}{\Rightarrow} G_{n-1} \overset{p_1}{\Rightarrow} P_1 \overset{p_2}{\Rightarrow} P_2 \dots \overset{p_m}{\Rightarrow} P_m$ already exists. We know by assumption that a concurrent rule r_c of $r_{n-j}, r_{n-j+1}, \dots, r_n$ with $j > 0$ exists which is applicable to G_0 such that each predecessor of r_c does not cause any conflict with r_c. Analogously to the previous item we can then conclude that $t : G_0 \overset{r_1}{\Rightarrow} G_1 \dots \overset{r_{n-1}}{\Rightarrow} G_{n-1} \overset{r_n}{\Rightarrow} G_n$ exists. Now consider $P_1 \overset{p_1}{\Leftarrow} G_{n-1} \overset{r_n}{\Rightarrow} G_n$. Since p_1 and r_n are parallel independent, due to the Local Church-Rosser Theorem with NACs [11] there exists also the following pair of direct transformations $P_1 \overset{r_n}{\Rightarrow} P'_1$ and $G_n \overset{p_1}{\Rightarrow} P'_1$. Therefore transformation sequence $G_{n-1} \overset{p_1}{\Rightarrow} P_1 \overset{r_n}{\Rightarrow} P'_1$ exists. Now consider $P_2 \overset{p_2}{\Leftarrow} P_1 \overset{r_n}{\Rightarrow} P'_1$. Analogously we can conclude that therefore $G_{n-1} \overset{p_1}{\Rightarrow} P_1 \overset{p_2}{\Rightarrow} P_2 \overset{r_n}{\Rightarrow} P'_2$ exists. Continuing analogously this argumentation we can prove that $t : G_0 \overset{r_1}{\Rightarrow} G_1 \dots \overset{r_{n-1}}{\Rightarrow} G_{n-1} \overset{p_1}{\Rightarrow} P_1 \overset{p_2}{\Rightarrow} P_2 \dots \overset{p_m}{\Rightarrow} P_m \overset{r_n}{\Rightarrow} P'_m$ exists.

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, IEEE Computer Society (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). Volume 10., Electronic Communications of the EASST (2008)
6. Lambers, L., Mariani, L., Ehrig, H., Pezze, M.: A Formal Framework for Developing Adaptable Service-Based Applications. In Fiadeiro, J., Inverardi, P., eds.: Proc. Fundamental Approaches to Software Engineering (FASE'08). Volume 4961 of Lecture Notes in Computer Science., Springer-Verlag (2008) 392–406
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. Jurack, S., Lambers, L., Mehner, K., Taentzer, G.: Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In: Proc. 11th Int. Conf. on Model Driven Engineering Languages and System MoDELS08. Volume 5301 of LNCS., Toulouse, France, Springer (2008) 341–355
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs on Theoretical Computer Science. Springer-Verlag (2006)
11. 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.: Proceedings of the ACCAT workshop at ETAPS 2007. ENTCS, Elsevier (2008) to appear.
12. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. In Bosangue, M.M., Johnsen, E.B., eds.: FMOODS 2007. Volume 4468., Springer, Berlin/Heidelberg (2007) 76–90
13. Eshuis, R., Wieringa, R.: Tool Support for Verifying UML Activity Diagrams. IEEE Trans. on Software Eng. **7**(30) (2004)
14. Stoerrle, H.: Semantics of UML 2.0 Activity Diagrams. In: International Conference on Visual Languages and Human Centric Computing VLHCC, IEEE Computer Society (2004)
15. Engels, G., Kleppe, A., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. In: Proceedings of ECMDA 2008. Volume 5095., Springer-Verlag Berlin Heidelberg (2008) 94–109
16. 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. Volume 4735., LNCS (2006)
17. Fujaba: Fujaba Homepage. (`http://www.fujaba.de`)
18. Moflon: Moflon Homepage. (`http://www.moflon.org`)
19. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic approaches to graph transformation I : Basic Concepts and Double Pushout Approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations. World Scientific (1997)
20. Lambers, L., Ehrig, H., Orejas, F.: Conflict Detection for Graph Transformation with Negative Application Conditions. In: Proc. Third International Conference on Graph Transformation (ICGT'06). Volume 4178 of Lecture Notes in Computer Science., Natal, Brazil, Springer-Verlag (2006) 61–76