Technische Universität Berlin

**Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik**

Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency between Control Flow and Functional Behavior

Claudia Ermel[1]
Jürgen Gall[1]
Leen Lambers[2]
Gabriele Taentzer[3]

[1]Technische Universität Berlin, Germany
claudia.ermel@tu-berlin.de, jgall@cs.tu-berlin.de

[2]Hasso-Plattner-Institut für Softwaresystemtechnik, Potsdam, Germany
leen.lambers@hpi.uni-potsdam.de

[3]Philipps-Universität Marburg, Germany
taentzer@mathematik.uni-marburg.de

Bericht-Nr. 2011 – 02
ISSN 1436-9915

Modeling with Plausibility Checking: Inspecting Favorable and Critical Signs for Consistency between Control Flow and Functional Behavior

Claudia Ermel[1], Jürgen Gall[1], Leen Lambers[2⋆] and Gabriele Taentzer[3]

[1] Technische Universität Berlin, Germany
claudia.ermel@tu-berlin.de, jgall@cs.tu-berlin.de

[2] Hasso-Plattner-Institut für Softwaresystemtechnik, Potsdam, Germany
leen.lambers@hpi.uni-potsdam.de

[3] Philipps-Universität Marburg, Germany
taentzer@informatik.uni-marburg.de

Abstract. UML activity diagrams are a common modelling technique to capture behavioral aspects of system models. Usually, pre- and post-conditions of activities are described in natural language and are not formally integrated with the static domain model. Hence, early consistency validation of activity models is difficult due to their semi-formal nature. In this paper, we use integrated behavior models that integrate activity diagrams with object rules defining sets of actions in simple activities. We formalize integrated behavior models using typed, attributed graph transformation. It provides a basis for plausibility checking by static conflict and causality detection between specific object rules, taking into account their occurrence within the control flow. This analysis leads to favorable as well as critical signs for consistency of the integrated behavior model. Our approach is supported by ActiGra, an Eclipse plug-in for editing, simulating and analyzing integrated behavior models. It visualizes favorable and critical signs for consistency in a convenient way and uses the well-known graph transformation tool AGG for rule application as well as static conflict and causality detection. We validate our approach by modeling a conference scheduling system.

⋆ The work of this author was partially funded by the Deutsche Forschungsgemeinschaft in the course of the project - Correct Model Transformations - see http://www.hpi.uni-potsdam.de/giese/projekte/kormoran.html?L=1.

1 Introduction

In model-driven software engineering, models are key artifacts which serve as basis for automatic code generation. Moreover, they can be used for analyzing the system behavior prior to implementing the system. In particular, it is interesting to know whether integrated parts of a model are consistent. For behavioral models, this means to find out whether the modeled system actions are executable in general or under certain conditions only. For example, an action in a model run might prevent one of the next actions to occur because the preconditions of this next action are not satisfied any more. This situation is usually called a *conflict*. Correspondingly, it is interesting to know which actions do depend on other actions, i.e. an action may be performed only if another action has occurred before. We call such situations *causalities*. The aim of this paper is to come up with a plausibility checking approach regarding the consistency of the control flow and the functional behavior given by actions bundled in object rules. Object rules define a pre-condition (which object pattern should be present) and a post-condition (what are the local changes). Intuitively, consistency means that for a given initial state there is at least one model run that can be completed successfully.

We combine activity models defining the control flow and object rules in an *integrated behavior model*, where an object rule is assigned to each simple activity in the activity model. Given a system state typed over a given class model, the behavior of an integrated behavior model can be executed by applying the specified actions in the pre-defined order. The new plausibility check allows us to analyze an integrated behavior model for *favorable* and *critical* signs concerning consistency. *Favorable* signs are e.g. situations where object rules are triggered by other object rules that precede them in the control flow. On the other hand, *critical* signs are e.g. situations where an object rule causes a conflict with a second object rule that should be applied after the first one along the control flow, or where an object rule depends causally on the effects of a second object rule which is scheduled by the control flow to be applied after the first one. An early feedback to the modeler indicating this kind of information in a natural way in the behavioral model is desirable to better understand the model.

For integrated behavior models, sufficient consistency criteria have been developed already in [10]. The analysis consists of generating sets of rule sequences, describing potential model runs, and investigate them with respect to their applicability to initial states in a static way. The advantage thereof is that it is possible to declare consistency of a behavioral model without having to simulate each potential model run. However, especially for an infinite set of potential runs (in case of loops), this technique may lead to difficulties. Moreover, it is based on sufficient criteria leading to false negatives. In this paper, we follow a different approach, focusing on *plausibility* reasoning on integrated behavior models and convenient visualization of the static analysis results. This approach is complementary to [10], since we opt for back-annotating light-weight static analysis results allowing for plausibility reasoning, also in case of lacking consistency analysis results from [10].

The light-weight analysis results do not only visualize the reason for successful consistency analysis with more elaborated analysis techniques as presented in [10]. In addition, the visualization of these light-weight results allows for plausibility reasoning on the integrated behavior model also in case of potential inconsistencies or false negatives. To this end, we determine conflicts and existing as well as non-existing causalities between object rules depending on the control flow. On the one hand they can lead to the detection of potential inconsistencies in the integrated behavior model, and on the other hand they can lead to a better understanding of the reasons for model consistency. By inspecting these potential conflicts and causalities, the modeler can reason about the plausibility of the model and possibly decide to adapt it. This light-weight technique seems to be very appropriate to allow for early plausibility reasoning during development steps of integrated behavior models. As long as no better static analysis techniques as presented in [10] or whenever no more detailed models (using e.g. object flow) are available, such a combination of lightweight static analysis with visual back-annotation allowing for semi-formal plausibility reasoning seems the appropriate way to go. We visualize the results of our plausibility checks in an integrated development environment called ActiGra[4]. Potential inconsistencies and reasons for consistency are directly visualized within integrated behavior models, e.g. as colored arcs between activity nodes and by detailed conflict and causality views.

Structure of the paper: Section 2 presents our running example. In Section 3, we introduce our approach to integrated behavior modeling and review the underlying formal concepts for static analysis based on graph transformation as far as needed. Different forms of plausibility checking are presented in Section 4, where we validate our approach checking a model of a conference scheduling system.

A section on related approaches (Section 5) and conclusions including directions for future work (Section 6) close the paper[5].

[4] http://tfs.cs.tu-berlin.de/actigra

[5] This technical report is an extended version of our contribution to FASE 2010 [4].

2 Case Study: A Conference Scheduling System

This case study[6] models planning tasks for conferences. Its class model is shown in Figure 1 (a). A *Conference* contains *Persons*, *Presentations*, *Sessions* and *Slots*. A *Person* gives one or more *Presentations* and may chair arbitrary many *Sessions*. Note that a session chair may give one or more presentations in the session he or she chairs. A *Presentation* is in at most one *Session* and *scheduled* in at most one *Slot*. Slots are linked as a list by *next* arcs and *used* by *Sessions*.

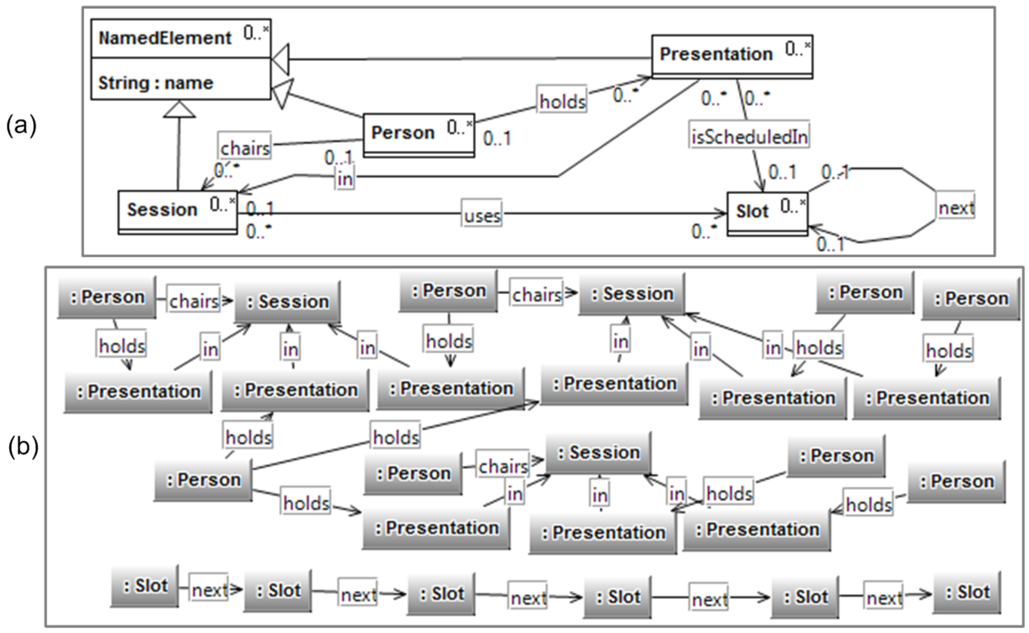


Fig. 1. Class and instance model for the *Conference Scheduling System*

Figure 1 (b) shows a sample object model of an initial session plan before presentations are scheduled into time slots[7]. This object model conforms to the class model. The obvious task is to find a valid assignment for situations like the one in Figure 1 (b) assigning the presentations to available time slots such that the following conditions are satisfied:

1. there are no simultaneous presentations given by the same presenter,
2. no presenter is chairing another session running simultaneously,
3. nobody chairs two sessions simultaneously,

[6] taken from the tool contest on *Graph-Based Tools 2008* [19]
[7] For simplicity, we do not show name attributes here.

4. the presentations in one session are given not in parallel but in consecutive time slots, and
5. unused time slots are only at the begin or end of the conference.

Moreover, it should be possible to generate arbitrary conference plans like the one in Figure 1 (b). This is useful to test the assignment procedure.

3 Integrating Activity Models with Object Rules

Our approach to behavior modeling integrates activity models with object rules, i.e. the application order of object rules is controlled by activity models. An object rule defines pre- and post-conditions of activities by sets of actions to be performed on object models. An object rule describes the behavior of a simple activity and is defined over a given class model. The reader is supposed to be familiar with object-oriented modelling using e.g. the UML [17]. Therefore, we present our approach to integrated behavior modeling from the perspective of its graph transformation-based semantics. In the following, we formalize class models by type graphs and object rules by graph transformation rules to be able to use the graph transformation theory [2] for plausibility checking.

In Section 3.1, we review the basic concepts of typed attributed graph transformation systems and in Section 3.2 the notions of conflicts and causalities that may occur between rules [2]. Section 3.3 – 3.4 introduce well-structured activity models and the semantics of integrated behavior models. Section 3.5 outlines the main features of the ActiGra tool for integrated behavior modeling.

3.1 Graphs and Graph Transformation

Graphs are often used as abstract representation of diagrams. When formalizing object-oriented modeling, graphs occur at two levels: the type level (defined based on class models) and the instance level (given by all valid object models). This idea is described by the concept of *typed graphs*, where a fixed *type graph TG* serves as an abstract representation of the class model. Types can be structured by an inheritance relation, as shown e.g. in the type graph for our *Conference Scheduling* model in Figure 1. Multiplicities and other annotations are not formalized by type graphs, but have to be expressed by additional graph constraints. Instance graphs of a type graph have a structure-preserving mapping to the type graph. The sample session plan in Figure 1 is an instance graph of the *Conference Scheduling* 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 of the rule and R is the right-hand side. Rule graphs may contain variables for attributes. The left-hand side L represents the pre-conditions of the rule, while the right-hand side R describes the post-conditions. $L \cap R$ (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. Furthermore, the application of a graph rule may be restricted by so-called *negative application conditions* (NACs) which prohibit the existence of certain graph patterns in the current instance graph. Note that we indicate graph elements common to L and R or common to L and a NAC by equal numbers.

Figure 2 shows graph rule *initial-schedule* modeling the scheduling of the first presentation of some session to a slot. The numerous conditions for this scheduling step stated in Section 2 are modelled by 8 NACs. The NAC shown in Figure 2 means that the rule must not be applied if the presenter holds already another presentation in the same slot[8].

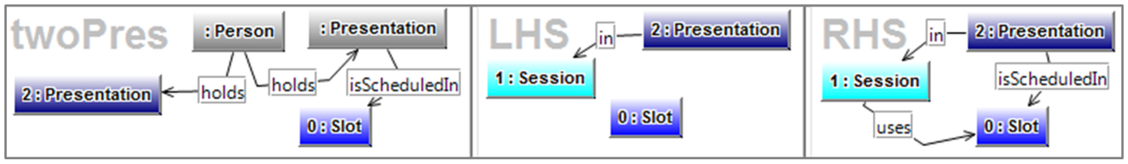


Fig. 2. Graph rule *initial-schedule*

A *direct graph transformation* $G \xRightarrow{r,m} H$ between two instance graphs G and 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 satisfies the NACs (i.e. the forbidden graph patterns are not found in G). We use injective matches only. Attribute variables used in graph object $o \in L$ are bound to concrete attribute values of graph object $m(o)$ in G. The resulting graph H is constructed by (1) deleting all graph items from G that are in L but not also in R; (2) adding all those new graph items that are in R but not also in L; (3) setting attribute values of preserved and created elements.

A *graph transformation (sequence)* consists of zero or more direct graph transformations. 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 to the same instance graph. There are techniques to restrict both kinds of choices. The

[8] For the complete case study with all rules and NACs see [1].

choice of matches can be restricted by object flow, while the choice of rules can be explicitly defined by control flow on activities.

3.2 Conflicts and Causalities between Rules

A reason for non-determinism of graph transformation systems is the potential existence of several matches for one rule. If two rules are applicable to the same instance graph, they might be applicable in any order with the same result. In this case they are said to be *parallel independent*.

Conflict Types. However, one rule *may disable* the second rule. In this case, the first rule r_1 is also said to be causing a *conflict* with the second rule r_2. The following types of conflicts can occur:

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

Causality Types. Conversely, it might be the case that one rule *may trigger* the application of another rule or *may be irreversible* after the application of another rule. In this case, this sequence of two rules is said to be *causally dependent.* The following types of causalities can occur because rule r_1 *triggers* the application of r_2:

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

Moreover, the following types of causalities may occur because the application of r_2 after r_1 makes the application of r_1 *irreversible*:

deliver/delete: The application of r_1 delivers (i.e. preserves or produces) an element deleted by the match of r_2.
forbid/produce: A NAC of r_1 forbids an element which is produced by r_2.
deliver/change: The application of r_1 delivers an attribute value changed by the match of r_2.

If r_1 never triggers r_2 and r_1 is always reversible after the application of r_1 and r_2, then r_1 followed by r_2 are said to be *sequentially independent.* In this case, their application order may be switched at all times leading to the same result. See [14] for a formal description of this conflict and causality characterization.

The tool environment AGG (Attributed Graph Grammar System)[9] is an established academic tool which allows the user to specify, execute and analyse graph transformation systems. AGG is used in the ActiGra-tool as underlying graph transformation engine and for static analysis of potential conflicts and causalities between rules. This conflict and causality analysis is based on critical pair analysis (CPA) [2,8] and critical sequence analysis (CSA) [14], respectively. 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. A critical sequence is a sequence of transformation steps $G \overset{r_1,m_1}{\Longrightarrow} H_1 \overset{r_2,m_2}{\Longrightarrow} H_2$ that are causally dependent in a minimal context, identified through co-match m_1' and m_2. Intuitively, each critical pair or sequence describes which rule elements need to overlap in order to cause a specific conflict or causality when applying the corresponding rules.

3.3 Integrated behavior models

As in [11], we define *well-structured activity models* as consisting 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. Figure 3 shows the visual appearance of activity model building blocks.

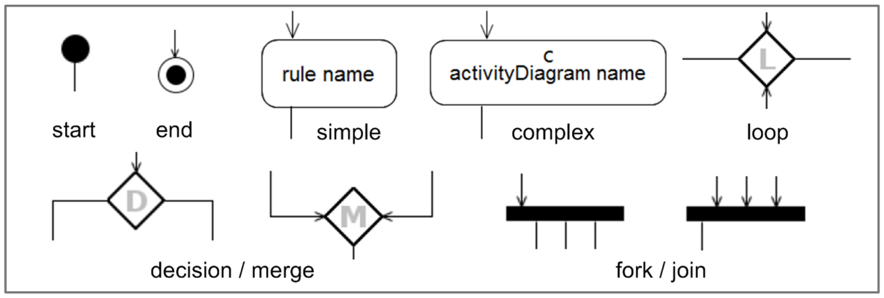


Fig. 3. Visual appearance of activity model building blocks

An *activity block* can be a simple activity, a sequence of blocks, a fork-join structure, decision-merge structure, and loop. In addition, we allow complex activities which stand for nested well-structured activity models. In this hierarchy, we forbid nesting cycles. Activity blocks are connected by transitions (directed arcs). Decisions have an explicit *if*-guard and implicit *else*-guard which equals the negated *if*-guard, and loops have a *loop*-guard with corresponding implicit *else*-guard.

[9] AGG: http://tfs.cs.tu-berlin.de/agg

In our formalization (see Section A), an *integrated behavior model* is a well-structured activity model A together with a type graph such that each *simple activity* a occurring in A is equipped with a *typed graph transformation rule* r_a and each *if* or *loop* guard is either *user-defined* or equipped with a typed *guard pattern*. We have *simple* and *application-checking* guard patterns: a simple guard pattern is a graph that has to be found; an application-checking guard pattern is allowed for a transition entering a loop or decision followed by a simple activity in the loop-body or if-branch, respectively, and checks the applicability of this activity; it is formalized by a graph constraint [7] and visualized by the symbol $[*]$. User-defined guards are evaluated by the user at run time to true or false. An *initial state* for an integrated behavior model is given by a typed instance graph.

Example 1. Let us assume the system state shown in Figure 1 as initial state of our integrated behavior model. The activity diagram *ScheduleControl* is shown in the left part of Figure 4 (please disregard the colors for now). Its first step performs the initial scheduling of sessions and presentations into time slots by applying rule *initial-schedule* (see Figure 2) as long as possible.

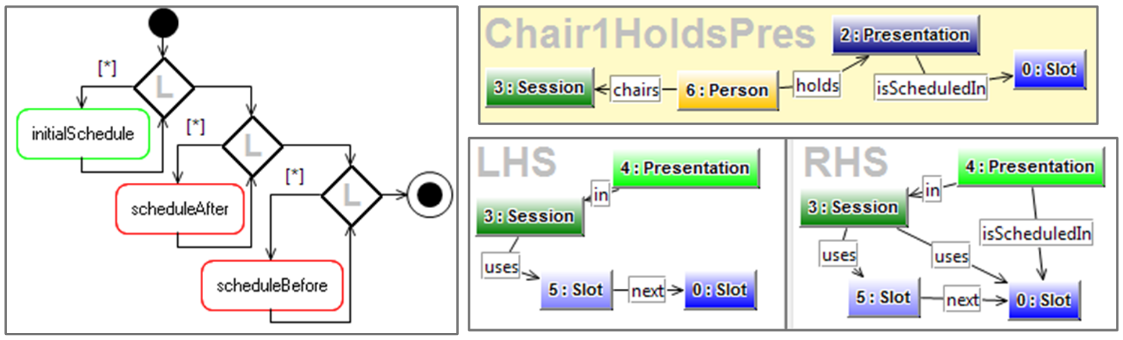


Fig. 4. Activity model *ScheduleControl* and rule *scheduleAfter*

As second step, two loops are executed taking care of grouping the remaining presentations of a session into consecutive time slots, i.e. a presentation is scheduled in a free time slot either directly before or after a slot where there is already a scheduled presentation of the same session. Rule *scheduleAfter* is shown in the right part of Figure 4. Rule *scheduleBefore* looks quite similar, only the direction of the next edge between the two slots is reversed. Both rules basically have the same NACs as rule *initialSchedule* ensuring the required conditions for the schedule (see [1]). The NAC shown here ensures that the session chair does not hold a presentation in the time slot intended for the current scheduling.

As in [11] we define a control flow relation on integrated behavior models.[10] Intuitively, two activities or guards (a, b) are control flow-related whenever b is performed or checked after a. Moreover, we define an against-control flow relation which contains all pairs of activities or guards that are reverse to the control flow relation.

The *control flow relation* CFR_A of an activity model A contains all pairs (x, y) where x and y are activities or guards such that (1)-(4) holds:

(1) $(x, y) \in CFR_A$ if there is a transition from activity x to activity y.
(2) $(x, y) \in CFR_A$ if activity x has an outgoing transition with guard y.
(3) $(x, y) \in CFR_A$ if activity y has an incoming transition with guard x.
(4) If $(x, y) \in CFR_A$ and $(y, z) \in CFR_A$, then also $(x, z) \in CFR_A$.

The *against-control flow relation* $ACFR_A$ of an activity model A contains all pairs (x, y) such that (y, x) is in CFR_A.

3.4 Simulation of Integrated Behavior Models

The *semantics* $Sem(A)$ of an integrated behavior model A consisting of a start activity s, an activity block B, and an end activity e is the *set of sequences* S_B, where each sequence consists of *rules alternated with graph constraints* (stemming from guard patterns), generated by the main activity block B (for a formal definition of the semantics see Section A).[11] For a block being a simple activity a inscribed by rule r_a, $S_B = \{r_a\}$. For a sequence block $B = X \rightarrow Y$, we construct $S_B = S_X \ seq \ S_Y$, i.e. the set of sequences being concatenations of a sequence in S_X and a sequence in S_Y. For decision blocks we construct the union of sequences of both branches (preceded by the if guard pattern and the negated guard pattern, respectively, in case that the if guard is not user-defined); for loop blocks we construct sequences containing the body of the loop i times $(0 \leq i \leq n)$ (where each body sequence is preceded by the loop guard pattern and the repetition of body sequences is concluded with the negated guard pattern in case that the loop guard is not user-defined). In contrast to [11], we restrict fork-join-blocks to one simple activity in each branch and build a parallel rule from all branch rules [14,2].[12] We plan to omit this restriction however, when integrating object flow [11] into our approach, since then it would be possible to build unique concurrent rules for each fork-join-branch. For B being a complex activity inscribed by the name of the integrated behavior model X, $S_B = Sem(X)$.

[10] In contrast to [11], we include guards into the control flow relation.

[11] Note that $Sem(A)$ does not depend on the initial state of A. Moreover, we have a slightly more general semantics compared to [11], since we do not only have rules in the sequences of S_B, but also graph constraints.

[12] This fork-join semantics is slightly more severe than in [11], which allows all interleavings of rules from different branches no matter if they lead to the same result.

Given $s \in Sem(A)$ a sequence of rules alternated with graph constraints and a start graph S, representing an initial state for A. We then say that each graph transformation sequence starting with S, applying each rule to the current instance graph and evaluating each graph constraint to true for the current instance graph in the order of occurrence in s, represents a *complete simulation run* of A. An integrated behavior model A is *consistent* with respect to a start graph S, representing an initial state for A, if there is a sequence $s \in Sem(A)$ leading to a complete simulation run. In particular, if A contains user-defined guards, usually more than one complete simulation run should exist.

3.5 The ActiGra tool

From the modeler's view, the main components of ActiGra[13] are the following *views* which are organized as a special Eclipse *perspective*, shown in Figure 5.

1. The *Tree View* $\boxed{1}$ gives an overview of all elements of an ActiGra, as usual in Eclipse applications. This view offers support to add / delete elements such as object graphs, graph constraints or rules, and to edit element names. By double clicking on an element, the visual view for this type of element is opened.
2. The *Type Graph View* $\boxed{2}$ visualizes a type graph and supports free-hand editing of node types, edge types, generalization edges and multiplicities. The palette on the right-hand side contains the drawing tools.
3. The *Instance Graph View* $\boxed{3}$ visualizes an instance graph and supports free-hand editing of instance graphs. The palette on the right-hand side contains tools to draw nodes and edges of the corresponding types.
4. The *Activity Diagram View* $\boxed{4}$ is a visual editor for well-structured activity diagrams. The panel contains all activity diagram elements. Simple activities contain the name of a rule which is applied when the corresponding activity is executed.
5. The *Rule View* $\boxed{5}$ is a multi-view editor consisting of editor panels for a rule's left- and right-hand sides (LHS, RHS) and (optionally) for one or more negative application conditions (NACs). Editing is supported like in the instance graph view, but in addition, mappings from the LHS to the RHS and to the NACs can be defined to identify common graph nodes. Mappings are visualized by equal node numbers and node colors. The mappings between edges can be inferred from the node mappings.
6. In the ActiGra *Toolbar* $\boxed{5}$, buttons are provided to execute simulation runs on selected activity models. Current activities are highlighted during simulation, and

[13] http://tfs.cs.tu-berlin.de/actigra

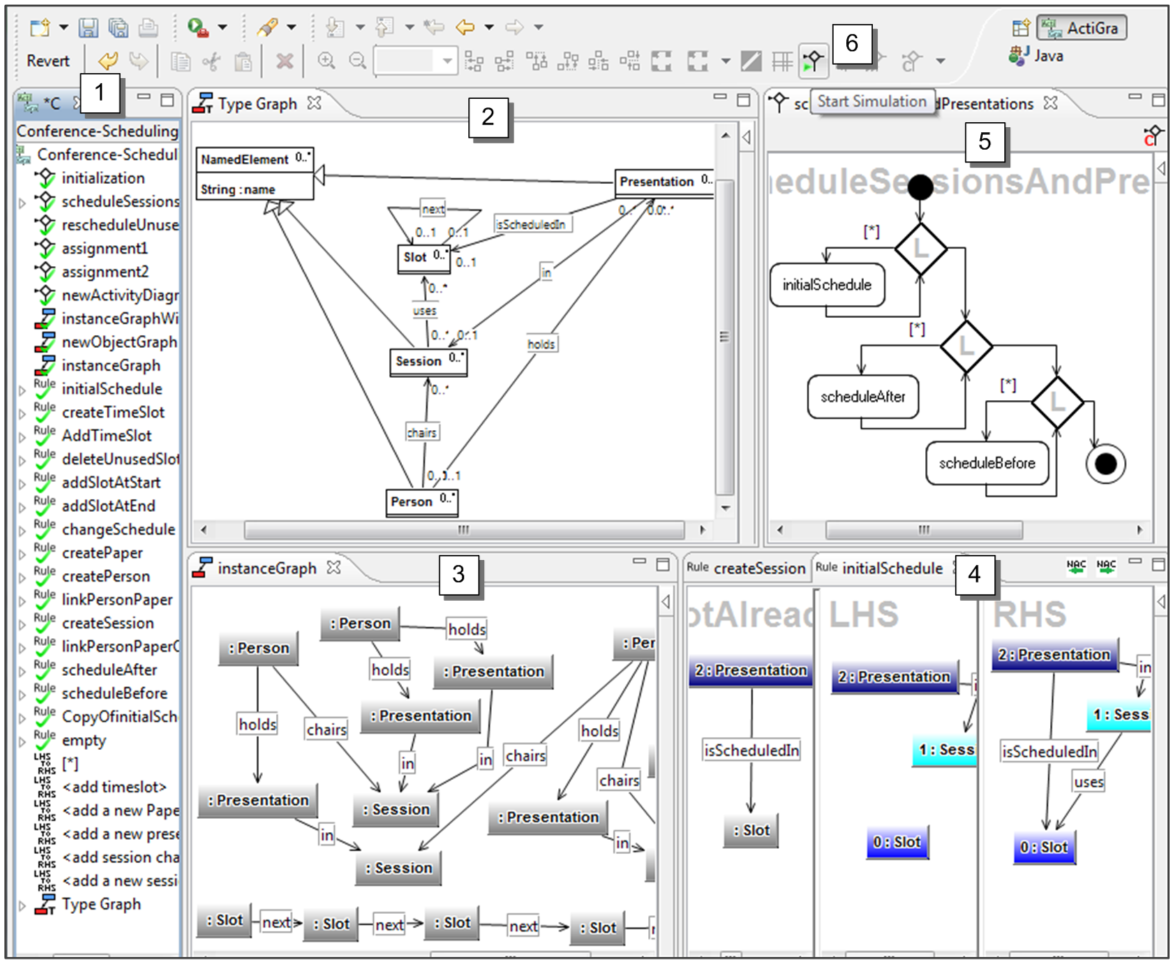


Fig. 5. The ACTIGRA perspective

the completion of simulation runs is indicated. User-defined guards are evaluated interactively. If a simulation run cannot be completed, an error message tells the user which activity could not be executed.

7. The *Graph Constraint View* (not shown in Figure 5) is a visual editor for graph constraints which are used as guards for decision or loop activities. The execution of such an activity checks the guard on the current object graph and chooses the corresponding branch if the graph constraint is fulfilled, or the alternative branch if it is violated.

4 Plausibility Checks for Integrated Behavior Models

We now consider how to check plausibility regarding consistency of the control flow and the functional behavior given by actions bundled in object rules. Thereby, we proceed as follows: We characterize *desired properties* for an integrated behavior model and its initial state to be consistent. We determine the *favorable* as well as *critical signs*[14] for these properties to hold, show, how the checks are supported by ActiGra and illustrate by our case study which conclusions can be drawn by the modeler to validate our approach.

For the plausibility checks we wish to detect potential conflicts and causalities between rules and guards occurring in the sequences of $Sem(A)$. Since in A simple activities, fork/joins as well as simple guard patterns correspond to rules[15] we just call them rules for simplicity reasons. Thereby, we disregard rules stemming from simple activities belonging to some fork/join block, since they do not occur as such in $Sem(A)$. Instead, the corresponding parallel rule for the fork/join is analyzed. As an exception to this convention, the plausibility check in Section 4.5 inspects consistency of fork/joins and analyzes also the enclosed simple activities.

4.1 Inspecting Initialization

If for some sequence in $Sem(A)$ the first rule is applicable, then the corresponding sequence can lead to a complete simulation run. Otherwise, the corresponding sequence leads to an incomplete run. Given an integrated behavior model A with initial state S, the first *plausibility check* computes automatically for which sequences in $Sem(A)$, the first rule is applicable to S. The modeler then may inspect the simulation run(s) that should complete for correct initialization (*desired property*). We identify the *favorable signs* as the set of possible initializations: $FaI_A = \{r | r$ is first rule of sequence in $Sem(A)$ and r is applicable to $S\}$. We identify the *critical signs* as the set of impossible initializations:
$CrI_A = \{r | r$ is first rule of a sequence in $Sem(A)$ and r is not applicable to $S\}$. [16]

[14] In most cases, these favorable and critical signs merely describe *potential* reasons for the property to be fulfilled or not, respectively. For example, some critical pair describes which kind of rule overlap may be responsible for a critical conflict. By inspecting this overlap, the modeler may realize that the potential critical conflict may actually occur and adapt the model to avoid it. On the other hand, he may realize that it does not occur since the overlap corresponds to an invalid system state, intermediate rules deactivate the conflict, etc.

[15] For each *simple guard pattern* we can derive a *guard rule* (without side-effects) for the guarded branch and a negated guard rule for the alternative branch (as described in [11]). Application-checking guard patterns are evaluated for simulation but disregarded by the plausibility checks, since they are not independent guards but check for the application of succeeding *rules* only.

[16] If a rule belongs to FaI_A, then the *initialization criterion* as presented in [12] as one of the sufficient criteria used for determining consistency [10] is satisfied. In case that each sequence in $Sem(A)$ starts

ActiGra visualizes the result of this plausibility check by highlighting the elements of FaI_A in *green*. Rules belonging to CrI_A are highlighted in *red*[17].

Example 2. Let us assume the system state in Figure 1 (b) as initial state. Figure 4 shows the initialization check result for activity model *ScheduleControl*. We have $FaI_{ScheduleControl} = \{initialSchedule\}$ and $CrI_{ScheduleControl} = \{scheduleAfter, scheduleBefore\}$.

Thus, complete simulation runs on our initial state never start with *scheduleAfter* or *scheduleBefore*, but always with *initialSchedule*.

4.2 Inspecting Trigger Causalities Along Control Flow Direction

If rule a may trigger rule b and b is performed after a, then it may be advantageous for the completion of a corresponding simulation run. If for some rule b no rule a is performed before b that may trigger b, this may lead to an incomplete simulation run and the modeler may decide to add some triggering rule or adapt the post-condition of some previous rule in order to create a trigger for b. Alternatively, the initial state could be adapted such that b is applicable to the start graph. Given an integrated behavior model A with initial state S, this *plausibility check* computes automatically for each rule a in A, which predecessor rules may trigger a. The modeler may inspect each rule a for enough predecessor rules to trigger a then (*desired property*). We identify the *favorable signs* as the set of potential trigger causalities for some rule a along control flow: $FaTrAl_A(a) = \{(b,a)|(b,a) \in CFR_A$ such that b may trigger $a\}$. We say that $FaTrAl_A = \{FaTrAl_A(a)\ |a$ is a rule in $A\}$ is the *set of potential trigger causalities in A along control flow*. We identify the *critical signs* as the set of non-triggered rules along control flow that are not applicable to the initial state: $CrNonTrAl_A = \{a|a$ is rule in A such that $FaTrAl_A(a) = \emptyset$ and a is not applicable to $S\}$. [18]

ActiGra visualizes the result of this plausibility check by displaying *dashed green arrows* from b to a selected rule a for each pair of rules (b,a) in $FaTrAl_A(a)$. If no rule is selected, then all pairs in $FaTrAl_A$ are displayed by dashed green arrows. Clicking on such an arrow from b to a opens a detail view, showing the reason(s)

with a rule in CrI_A, the *initialization error criterion* as presented in [12] is fulfilled for each sequence in $Sem(A)$ such that A is not consistent w.r.t. S [10].

[17] Concerning fork/join blocks in FaI_A or CrI_A,ActiGra colors the fork bar (see Example 7 in Section B.1).

[18] The knowledge whether a rule is applicable to the initial state together with the knowledge of the triggering causalities in $FaTrAl_A$ helps to understand why the *enabling predecessor criterion* as presented in [12] as one of the sufficient criteria used for determining consistency [10] is satisfied. Conversely, for each rule a in $CrNonTrAl_A$ not applicable to the initial state, the *no enabling predecessor criterion* as presented in [12] is fulfilled. If each sequence in $Sem(A)$ contains a rule belonging to $CrNonTr_A$ in red, then A is not consistent w.r.t. S [10].

why b may trigger a as discovered by CSA. Conversely, ActiGra highlights each rule belonging to $CrNonTrAl_A$ in *red*.

Example 3. Consider activity model *GenConfPlans* in (Figure 6) for generating conference plans, assuming an empty initial state. The set of potential trigger causalities along control flow for *createSession* is given by $FaTrAl_{GenConfPlans}\ (createSession) = \{(createPerson + createPaper, createSession), (createPerson, createSession)\}$.

Here, we learn that we need at least one execution of a loop containing rule *createPerson* (a rule with an empty left-hand side) to ensure a complete simulation run containing *createSession*.

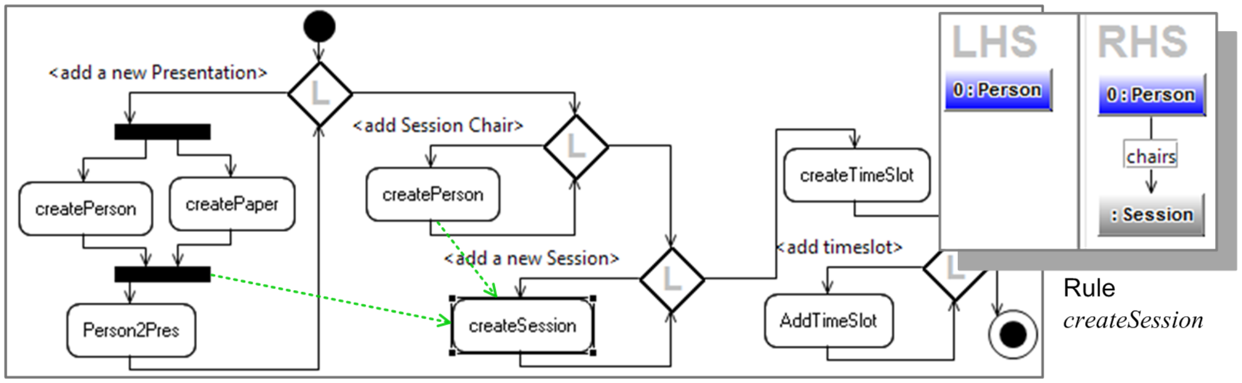


Fig. 6. Potential trigger causalities along control flow in activity model *GenConf-Plans*

The detail view for this potential trigger causality *(createPerson, createSession)* (see Figure 7) informs us that we here have a produce-use-dependency, i.e. rule *createPerson* may produce a node of type *Person* that is used by rule *createSession* to link it to a new *Session* node.

4.3 Inspecting Conflicts Along Control Flow Direction

If rule a may disable rule b, and b is performed after a, then this may lead to an incomplete simulation run. On the other hand, if for some rule a no rule b performed before a exists that may disable rule a, then the application of a is not impeded. Given an integrated behavior model A with initial state S, this *plausibility check* computes automatically for each rule a in A, which successor rules b in A may be disabled by a. The modeler then may inspect each rule a in A for the absence of rules performed before a disabling rule a (*desired property*). We identify the

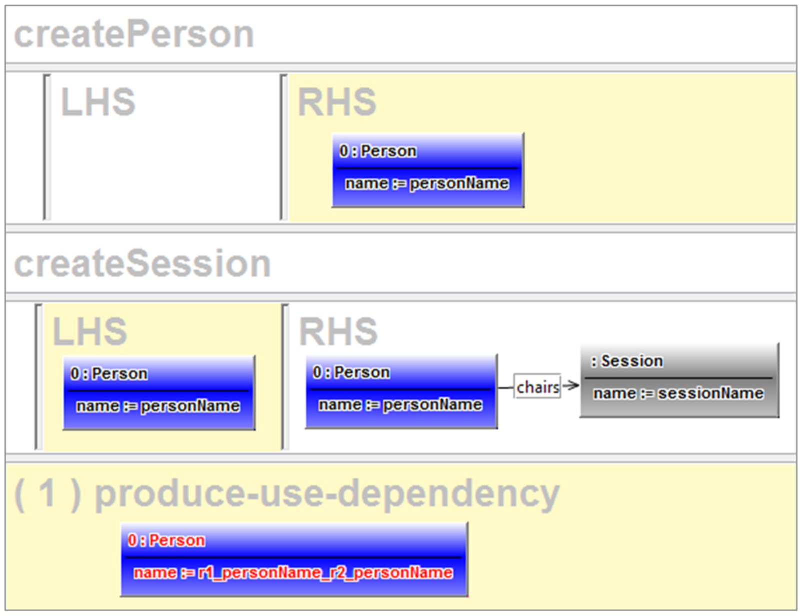


Fig. 7. Detail View of the potential trigger causality (*createPerson, createSession*) from Figure 6

critical signs as the set of potential conflicts along control flow caused by rule a: $CrDisAl_A(a) = \{(a,b)|a,b$ are rules in $A, (a,b) \in CFR_A$ and a may disable $b\}$. We say that $CrDisAl_A = \{CrDisAl_A(a)\ |a$ is a rule in $A\}$ is the *set of potential conflicts along control flow in* A. We identify the *favorable signs* as the set of non-disabled rules along control flow: $FaNonDisAl_A = \{a|a$ in A and $\nexists(b,a) \in CrDisAl_A\ \}$.[19]

ActiGra visualizes the result of this plausibility check by displaying faint red arrows from a to b for each pair of rules (a,b) in $CrDisAl_A$. If rule a is selected, a bold red arrow from a to b for each pair of rules (a,b) in $CrDisAl_A(a)$ is shown. Clicking on such an arrow opens a detail view, showing the reason(s) why a may disable b as discovered by CPA. Each rule a in A belonging to $FaNonDisAl_A$ is highlighted in *green*.

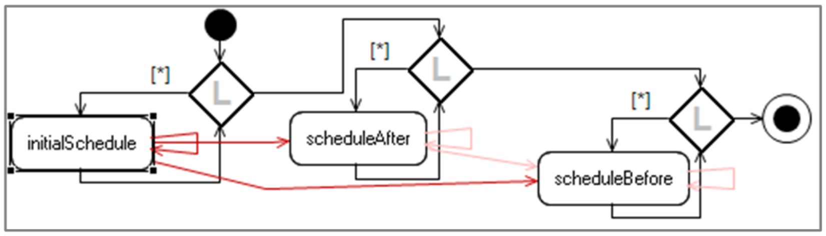
Example 4. Consider activity model *SchedulingControl* in Figure 8 (a). Here, the set of potential conflicts along control flow caused by rule *initialSchedule* is given by $CrDisAl_{SchedulingControl}(initialSchedule) = \{(initialSchedule, initialSchedule)$, $(initialSchedule, scheduleAfter)$, $(initialSchedule, scheduleBefore)\}$[20]. This gives the mod-

[19] If $CrDisAl_A$ is empty, then each rule sequence in $Sem(A)$ satisfies the no-impeding predecessor criterion as presented in [12] as one of the sufficient criteria used for determining consistency [10].

[20] Note that one pair in this set may indicate more than one conflict potentially occurring between the corresponding rules.

eler a hint that in fact a scheduling might not terminate successfully in the case that rule *initialSchedule* creates a situation where not all remaining presentations can be scheduled in a way satisfying all conditions. The detail view of potential conflicts for pair $(initialSchedule, scheduleAfter)$ in Figure 8 (b) shows e.g. a potential produce-forbid conflict where rule *initialSchedule* (Figure 2) produces an edge from 2:Pres to 0:Slot, and rule *scheduleAfter* then must not schedule 4:Pres to 0:Slot because of the NAC shown in Figure 4.

(a)



(b)

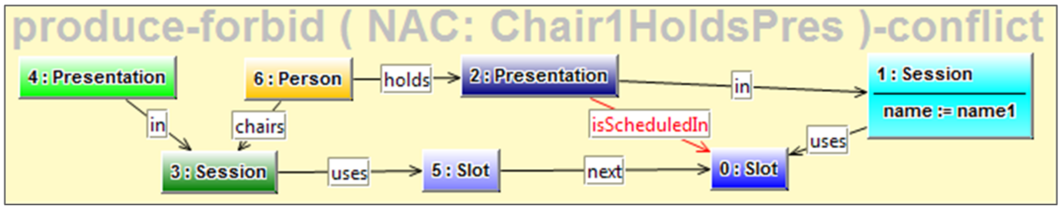


Fig. 8. (a) Potential conflicts along control flow caused by rule *initialSchedule*; (b) Detail view of potential conflict of rule *initialSchedule* with rule *scheduleAfter*.

4.4 Inspecting Trigger Causalities Against Control Flow Direction

If rule a may trigger rule b and b is performed before a, then it might be the case that their order should be switched in order to obtain a complete simulation run. Given an integrated behavior model A with initial state S, this *plausibility check* automatically computes for each rule a in A, which successor rules of a may trigger a. The modeler then may inspect for each rule a in A that no rule performed after a exists that needs to be switched to a position before a in order to trigger its application (*desired property*). We identify the *critical signs* as the set of potential causalities against control flow triggered by a: $CrTrAg_A(a) = \{(a,b)|a,b$ rules in A and $(a,b) \in ACFR_A$ such that a may trigger $b\}$. We say that $CrTrAg_A = \{CrTrAg_A(a)\ |a$ is a rule in $A\}$ is the *set of potential trigger causalities against control flow in* A. We identify the *favorable signs* as the set of rules not triggered against control flow: $FaNoTrAg_A = \{a|a$ is rule in A and $\nexists(b,a) \in CrTrAg_A\ \}$.

ActiGra visualizes the result of this plausibility check by displaying a *dashed red arrow* from a selected rule a to b for each pair of rules (a, b) in $CrTrAg_A(a)$. If no rule in particular is selected, then all pairs in $CrTrAg_A$ are displayed by dashed red arrows. Clicking on such an arrow from a to b opens a detail view, showing the reason(s) why a may trigger b as discovered by CSA. Conversely, each rule belonging to $FaNoTrAg_A$ is highlighted in *green*.

Example 5. In activity diagram *GenConfPlan* in Figure 9, we get the set of potential causalities against control flow $CrTrAg_{GenConfPlan} = \{(createSession, Person2Pres), (createPerson, Person2Pres)\}$. On selecting activity *createSession*, the red dashed arc representing the potential causality $(createSession, Person2Pres)$ is highlighted (see Figure 9). We have several rules highlighted in green (being not triggered against control flow): $FaNoTrAg_{GenConfPlan} = \{createPerson + createPaper, createPerson, createSession\}$, where the first rule is constructed as parallel rule of the fork/join branch rules *createPerson* and *createPaper*.

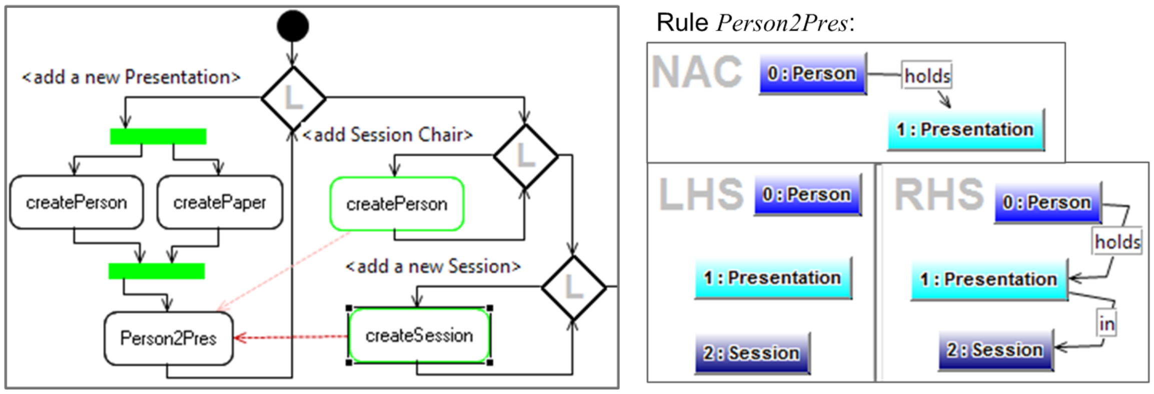


Fig. 9. Trigger causality against control flow *(createSession, Person2Pres)*

The detail view of the potential causality $(createSession, Person2Pres)$ is shown in Figure 10. Here, the user sees that there would be a causality if rule *createSession* generated exactly the session node that would be used by rule *Person2Pres* to schedule a presentation in. Hence, causality *(createSession, Person2Pres)* indicates that rule *Person2Pres* might be modelled too early in the control flow since rule *createSession* might be needed to trigger rule *Person2Pres* completely.

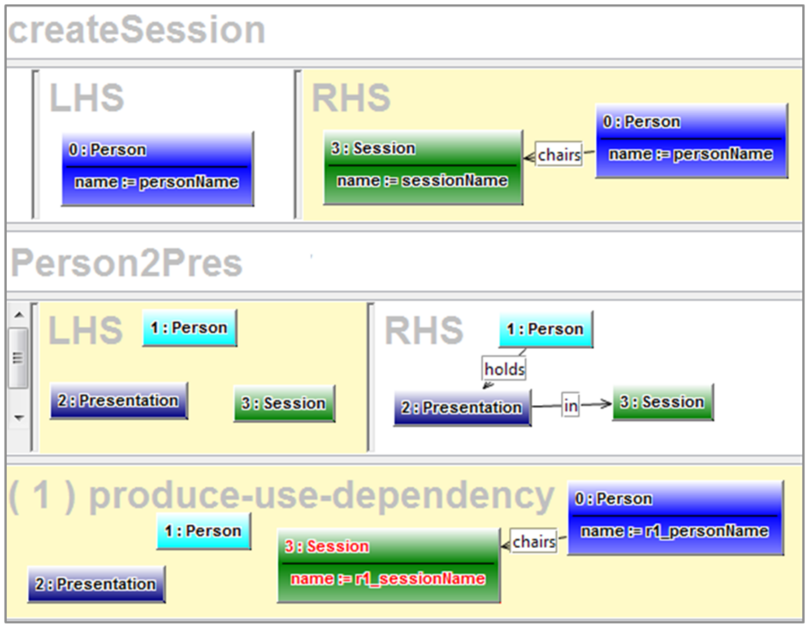


Fig. 10. Detail view of the trigger causality against control flow *(createSession, Person2Pres)* in Figure 9

4.5 Inspecting Causalities in Fork/Joins

We may not only consider the consistent sequential composition of rules as before, but consider also the parallel application of rules as specified by fork/join activities. Whenever a rule pair (a, b) belonging to the same fork/join may be causally dependent, then it is not possible to change their application order in any situation without changing the result. However, the parallel application of rules (a, b) implies that their application order should not matter.

Given an integrated behavior model A with initial state S, this *plausibility check* computes automatically for each fork/join in A, if potential causalities between the enclosed simple activities exist. The modeler may inspect each fork/join for its parallel execution not to be disturbed then (*desired property*).

We need some more elaborated considerations for this case, since we wish to analyze simple activities within a fork/join block that are normally disregarded as they only occur in the form of the corresponding parallel rule in $Sem(A)$. In particular, we define a *fork/join relation* FJR_A consisting of all rule pairs (a, b) belonging to the same fork/join block. We identify the *critical signs* as the set of potential causalities between different fork/join branches: $CrFJCa_A = \{(a, b)|(a, b) \in FJR_A$ and (a, b)

causally dependent}.[21] We identify the *favorable signs* as the set of fork/join structures with independent branches: $FaFJNoCa_A = \{fj | fj$ is fork/join in A and $(a, b) \notin CrFJCa_A$ for each (a, b) with a,b in different branches of $fj\}$.

ActiGra visualizes the result of this plausibility check by displaying in each fork/join block a *dashed red arrow* from a to b for each $(a, b) \in CrFJCa_A$. The detail view shows the reason(s) why (a, b) are causally dependent , as discovered by CSA, and why this dependency might disturb parallel execution. On the other hand, each fork/join in $FaFJNoCa_A$ is highlighted by *green* fork and join bars.

Example 6. The set of potential causalities between different fork/join branches depicted in Figure 11 is given by $\{(createPerson, Person2Pres)\}$. We may have a dependency (shown in the detail view) if rule *createPerson* creates a *Person* node that is used by rule *Person2Pres* to link it to a *Presentation* node.

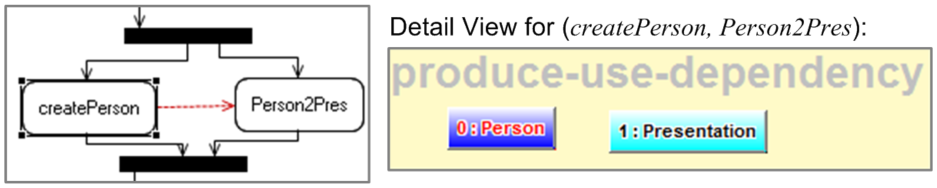


Fig. 11. Potential causality between different fork/join branches and its detail view

5 Related Work

This paper is about formal semantics and analysis of activity models on the one hand, and controlled graph transformation on the other hand. Our approach complements existing approaches that give a denotational semantics to activity diagrams by formal models. This semantics is used for validation purposes thereafter. For example, Eshuis [5] proposes a denotational semantics for a restricted class of activity models by means of labeled transition systems. Model checking is used to check properties. Störrle [20] defines a denotational semantics for the control flow of UML 2.0 activity models including procedure calls by means of Petri nets. The standard Petri net theory provides an analysis of properties like reachability or deadlock freeness. Both works stick to simple activities not further refined. In [3], business process models and web services are equipped with a combined graph transformation semantics and consistency can be validated by the model checker GROOVE.

[21] Here, we do not only regard trigger causalities between a and b, but also causalities making the application of rule a irreversible as described in [14].

In contrast, we take integrated behavior models and check for potential conflict and causality inconsistencies between activity-specifying rules directly. Thus, our technique is not a "push-button" technique which checks a temporal formula specifying a desired property, but offers additional views on activity models where users can conveniently investigate intended and unintended conflicts and causalities between activities. Conflicts and causalities are not just reported as such but reasons for consistencies and inconsistencies can also be investigated in depth.

Fujaba [6], VMTS[22] and GReAT[23] are graph transformation tools for specifying and applying graph transformation rules along a control flow specified by activity models. However, controlled rule applications are not further validated concerning conflict and causality inconsistencies within these tools. AGG is the only graph transformation tool which supports critical pair analysis to detect conflicts and causal dependencies between rules applied in control flows specified. Conflicts and causalities of pairs of rule-specified activities have been considered in various application contexts such as use case integration [8], feature modeling [9], model inconsistency detection [16], and aspect-oriented modeling [15]. Although sometimes embedded in explicit control flow, it has not been taken into account for inconsistency analysis. In this paper, we analyze potential conflict and causality inconsistencies between rule-specified activities wrt. the control flow to specify their execution order. A pair of pre- and post-conditions can be formalized as a graph transformation rule. The idea was first presented in [8] to analyze inconsistencies conflicts and dependencies between activities during use case integration, however not taking into account the control flow. Jayaraman et al.[9] use critical pair analysis to detect dependencies and conflicts between features modeled 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 this analysis.

6 Conclusions and Future Work

Activity models are a wide-spread modeling technique to specify behavioral aspects of (software) systems. Here, we consider activity models where activities are integrated with object rules which describe pre- and post-conditions of activities based on a structural model. These integrated behavior models are formalized on the basis of graph transformation. Considering use case-driven system analysis and design, behavior models can be stepwise refined by first refining a use case by a simple activity model and then, refining activities again by activity models or describing

[22] Visual Modeling and Transformation System: http://vmts.aut.bme.hu/
[23] Graph Rewriting and Transformation: http://www.isis.vanderbilt.edu/tools/great

their pre- and post-conditions by rules over a given structural model. The integrated specification of object rules within a control flow offers the possibility to find out potential conflict and causality inconsistencies. Actually, we can check if the order of rule applications specified by the control flow is plausible w.r.t. inherent potential conflicts and causalities of object rules. We determine potential critical conflicts and causalities if the specified control flow can become inconsistent with inherent conflicts and causalities. In contrast, we can also check for potential favorable causalities to reason about the necessity of the specified control flow. The Eclipse plug-in ActiGra[24] prototypically implements these plausibility checks and visualizes potential conflicts and causalities in different views. Please note that our approach to plausibility reasoning can easily be adapted to any other approach where modeling techniques describing the control flow of operations, are integrated with operational rules like e.g. the integration of live sequence charts with object rules in [13].

As presented in Section 3.4, the semantics of integrated behavior models can be defined by sets of sequences consisting of graph transformation rules alternated with graph constraints. Although the order of rule applications is determined, rule matches still can be chosen non-deterministically. Hence, plausibility checks determine *potential* conflicts and causalities only. It depends on the actual rule application whether conflicts or causalities really occur.

In addition to the plausibility checks for consistency of integrated behavior models w.r.t applicability, we plan to design plausibility checks indicating positive or negative signs for the termination of pattern-guarded loops. Loops require particular attention, since the termination aspect needs to be regarded and because the control flow relation as well as the against-control-flow relation contain the same set of pairs of rules occurring within the loop body (a motivating example can be found in Section B.4).

A further refinement step in activity-based behavior modeling would be the specification of object flow between activities. Additionally specified object flow between two activities would further determine their inter-relation. In this case, previously determined potential conflicts and causalities might not occur anymore. Thus, the plausibility checks would become more exact with additionally specified object flow. A first formalization of integrated behavior models with object flow based on graph transformation is presented in [11]. An extension of plausibility checks to this kind of activity models is left for future work. In [10], sufficient criteria for the application of integrated behavior models have been presented. These criteria could also be considerably sharpened if the object flow is additionally specified. The plausibility checks are expected to support the specification of object flow. Moreover, we plan to implement and visualize the sufficient criteria for consistency [10] in Acti-

[24] http://tfs.cs.tu-berlin.de/actigra

Gra. To conclude, integrated behavior models head towards a better integration of structural and behavioral modeling of (software) systems. Plausibility checks provide light-weight static analysis checks supporting the developer in constructing consistent models. Additionally, they allow modelers to reason about the necessity of sequencing activities.

References

1. Biermann, E., Ermel, C., Lambers, L., Prange, U., Taentzer, G.: Introduction to AGG and EMF Tiger by modeling a conference scheduling system. Int. Journal on Software Tools for Technology Transfer 12(3-4), 245–261 (2010)
2. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science, Springer (2006)
3. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring consistency of business process models and web services using visual contracts. In: Proc. AGTIVE). LNCS, vol. 5088, pp. 17–31. Springer (2007)
4. Ermel, C., Gall, J., Lambers, L., Taentzer, G.: Modeling with plausibility checking: Inspecting favorable and critical signs for consistency between control flow and functional behavior. In: Proc. Fundamental Aspects of Software Engineering (FASE'11). pp. 156–170. No. 6603 in LNCS, Springer (2011)
5. Eshuis, R., Wieringa, R.: Tool support for verifying UML activity diagrams. IEEE Transactions on Software Engineering 7(30) (2004)
6. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Proc. Theory and Appl. of Graph Transformation. LNCS, vol. 1764, pp. 296–309. Springer (1998)
7. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science 19, 1–52 (2009)
8. Hausmann, J., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: Proc. ICSE. pp. 105–115. ACM (2002)
9. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings. LNCS, vol. 4735, pp. 151–165. Springer (2007)
10. Jurack, S., Lambers, L., Mehner, K., Taentzer, G.: Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In: Czarnecki, K. (ed.) Proc. MoDELS. LNCS, vol. 5301, pp. 341–355. Springer (2008)

11. Jurack, S., Lambers, L., Mehner, K., Taentzer, G., Wierse, G.: Object Flow Definition for Refined Activity Diagrams . In: Proc. Int. Conf. on Fundamental Approaches to Software Engineering. LNCS, vol. 5503, pp. 49–63. Springer (2009)
12. Lambers, L., Ehrig, H., Taentzer, G.: Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences. In: Proc. GTVMT. vol. 10. ECEASST (2008)
13. Lambers, L., Mariani, L., Ehrig, H., Pezze, M.: A Formal Framework for Developing Adaptable Service-Based Applications. In: Proc. FASE. LNCS, vol. 4961, pp. 392–406. Springer (2008)
14. Lambers, L.: Certifying Rule-Based Models using Graph Transformation. Ph.D. thesis, Technische Universität Berlin (2009)
15. Mehner, K., Monga, M., Taentzer, G.: Analysis of aspect-oriented model weaving. In: Transactions on Aspect-Oriented Software Development V, LNCS, vol. 5490, pp. 235–263. Springer (2009)
16. Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Proc. MoDELS. LNCS, vol. 4199, pp. 200–214. Springer (2006)
17. Object Management Group: Unified Modeling Language: Superstructure – Version 2.3 (2010), http://www.omg.org/spec/UML/2.3/, formal/07-02-05
18. Pennemann, K.H.: Development of Correct Graph Transformation Systems. Ph.D. thesis, Department of Computing Science, University of Oldenburg, Oldenburg (2009), http://formale-sprachen.informatik.uni-oldenburg.de/~skript/fs-pub/diss_pennemann.pdf, http://oops.uni-oldenburg.de/volltexte/2009/948/Electronic Dissertation
19. Rensink, A., Van Gorp, P. (eds.): Int. Journal on Software Tools for Technology Transfer, Section on Graph Transf. Tool Contest 2008, vol. 12(3-4). Springer (2010)
20. Störrle, H.: Semantics of UML 2.0 activity diagrams. In: Proc. Int. Conf. on Visual Languages and Human-Centric Computing (VLHCC'04). IEEE (2004)

A Formalization of Integrated Behavior Models

Definition 1 (well-structured activity model). *A* well-structured activity model *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 *can be one of the following:*

(1) empty: *An empty activity block is not depicted.*
(2) simple: *A simple activity is an activity block.*
(3) sequence: *A sequence of two activity blocks A and B connected by a transition from A to B form an activity block.*
(4) decision/merge: *A decision activity which is followed by two guarded transitions leading to one activity block each and where each block is followed by a transition both heading to a common merge activity form an activity block. One transition is explicitly guarded, called the if-guard, while the other transition carries a predefined guard "else" which equals the negated if-guard.*
(5) loop: *A loop activity is followed by a guarded transition. This guard is called loop-guard. The transition leads to an activity block with an outgoing transition to the same loop activity as above. Considering this loop activity again, its incoming transition from outside becomes the incoming transition of the new block. Its outgoing transition to outside becomes the outgoing transition of the new block. This transition carries a predefined guard "else" which equals the negated loop-guard. The whole construct forms an activity block.*
(6) fork/join: *A fork activity followed by one or more branches with one simple activity each followed by a join activity form an activity block.*

In the following, our definitions are based on the definitions of typed attributed graphs, graph transformation rules and graph constraints in [2].

Definition 2 (integrated behavior model). *An* integrated behavior model *is a well-structured activity model A together with a type graph such that each* simple activity *a occurring in A is equipped with a* typed graph transformation rule r_a *and each* if *or* loop *guard g is either* user-defined *or equipped with a typed* guard pattern P_g. *We have* simple *and* application-checking *guard patterns, formalized by graph constraints [7]: a simple guard pattern is formalized by a graph constraint of the form $\exists C$, where C is a typed graph that may contain variables for attributes; A typed instance graph G satisfies $\exists C$ if there exists a morphism $q : C \to G$, injective on the graph part. An application-checking guard pattern is allowed for a transition entering a loop or decision followed by a simple activity a (for which applicability is checked) in the loop-body or if-branch, respectively, it is visualized by the symbol* $[*]$, *and it is formalized by a graph constraint of the form* $\exists(L, (\wedge_{i\in I}\neg\exists n_i) \wedge dangl(r_a))$, *expressing that a match* $m : L \to G$ *for* r_a *exists such that the negative application conditions* $n_i : L \to N_i$ *of* r_a *and the dangling edge condition for* r_a *are fulfilled.*[25]*; A typed instance graph* G *satisfies the application-checking guard pattern for the simple activity* a *if and only if* r_a *is applicable to* G*. User-defined guards are evaluated by the user at run time to true or false. An* initial state *for an integrated behavior model is given by a typed instance graph.*

Next, we give the semantics of integrated behavior models. As each simple activity a is equipped with a rule r_a, and each simple or application-checking guard pattern P_g is formalized by a graph constraint, we define the semantics as sequences of rules alternated with graph constraints, where each of the sequences is determined by the control flow of the activity model. Since loops may be guarded by guard patterns, we also give a restricted semantics parametrized by a fixed number of loop executions. Based on this restricted semantics, it may be checked if a predefined number of loop executions is feasible in some simulation run.[26]

Definition 3 (semantics of integrated behavior models). *Given an activity block* B *of an integrated behavior model, its corresponding set of rule sequences* S_B *is defined as follows.*

(1) *If* B *is the* empty *block,* $S_B = \emptyset$.
(2) *If* B *consists of a* simple activity a *equipped with rule* r_a, $S_B = \{r_a\}$.
(3) *If* B *is a* sequence *of block* X *and* Y, $S_B := S_X \; seq \; S_Y = \{s_x s_y | s_x \in S_X \wedge s_y \in S_Y\}$
(4a) *If* B *is a* decision *block on block* X *and block* Y *with guard pattern* P_g *as if-guard,* $S_B = (\{P_g\} \; seq \; S_X) \cup (\{\neg P_g\} \; seq \; S_Y)$
(4b) *If* B *is a* decision *block on block* X *and block* Y *with user-defined if-guard,* $S_B = S_X \cup S_Y$
(5a) *If* B *is a* loop *block on* X *with guard pattern* P_g *as loop-guard,* $S_B := loop(P_g, S_X) = \bigcup_{i\in I} S_X^i$ *where* $S_X^0 = \{\neg P_g\}$, $S_X^1 = \{P_g\} \; seq \; S_X \; seq \; \{\neg P_g\}$,
$S_X^2 = \{P_g\} \; seq \; S_X \; seq \; S_X^1$ *and* $S_X^i = \{P_g\} \; seq \; S_X \; seq \; S_X^{i-1}$ *for* $i > 2$ *such that* $S_B(n) = S_X^n$ *denotes the semantics of loop block* B *with exactly* n *loop executions.*
(5b) *If* B *is a* loop *block on* X *with user-defined loop-guard,* $S_B := loop(S_X) = \bigcup_{i\in I} S_X^i$, *where* $S_X^0 = \emptyset$, $S_X^1 = S_X$, $S_X^2 = S_X \; seq \; S_X^1$ *and* $S_X^i = S_X \; seq \; S_X^{i-1}$ *for* $i > 2$.

[25] In [18], it is described by the so-called *Deletable* condition how to formalize $dangl(r_a)$. Basically, it prohibits the existence of adjacent edges not belonging to the match for nodes that are to be deleted.
[26] In [10] we considered only user-defined guards.

(6) *If B is a* fork *block on simple activity blocks X_i with $i \in I$ and I a finite index set such that $S_{X_i} = \{r_i\}$, then $S_B := \{\bigoplus_{i\in I} r_i\}$ containing the parallel rule with NACs [14] of all r_i.*

The semantics *$Sem(A)$ of activity model A with start activity s, activity block B, and end activity e is defined as the set of rule sequences S_B generated by the main activity block B. If A contains k guarded loops, $Sem_{n_1,\dots,n_k}(A) \subset Sem(A)$ denotes a* restricted *semantics where the semantics of each guarded loop $B_j \in A$ for $1 \leq j \leq k$ is $S_{B_j}(n_j)$.*

B Plausibility Checks of the Case Study

In this section, additional examples and more details of the plausibility checks on the case study are discussed.

B.1 Inspecting Initialization

Example 7. Invoking the initialization check on the activity model *GenConfPlan* in Figure 12 results in green highlighting of the fork/join block, *createPerson* and *createTimeSlot* (being first rules of rule sequences in *Sem(GenConfPlan)*), while rule *createSession* is highlighted in red.

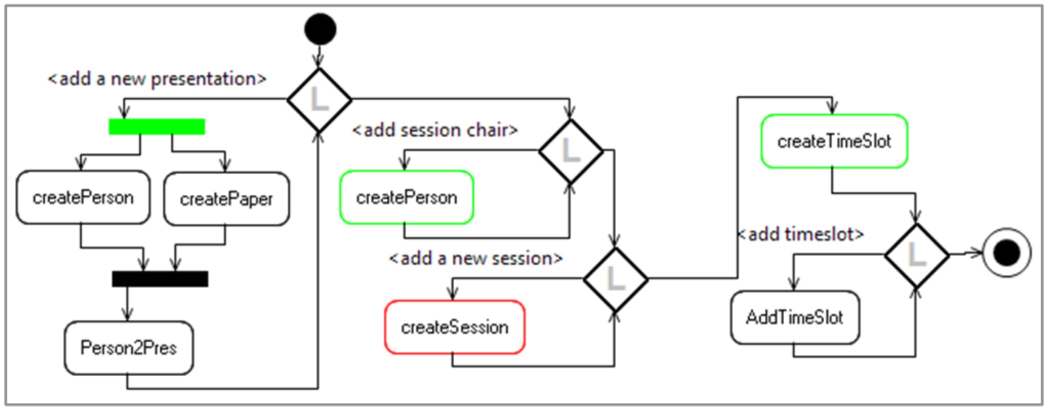


Fig. 12. Possible and impossible initializations in activity model *GenConfPlan*

Having a closer look at rule *createSession* in Figure 13 reveals that this rule expects a *Person* node as precondition to be assigned as session chair of the new session. Thus, rule *createPerson* should be executed at least once before the loop containing rule *createSession* is entered.

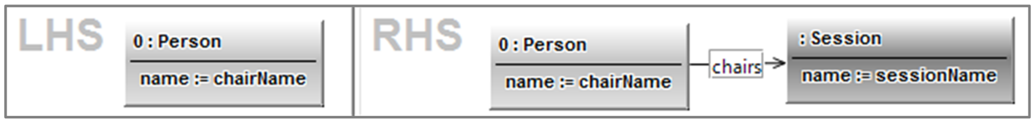


Fig. 13. Rule *createSession*

B.2 Inspecting Trigger Causalities Along Control Flow Direction

Example 8. Consider the draft activity diagram $B2$ for generating conference plans shown in Figure 14. Here, the set $FaTrAl_{B2}(Person2Pres)$ of potential trigger causalities along control flow is empty (it is highlighted in red). Moreover, this rule is not applicable to the initial state of our model (it expects an existing person and a presentation). Hence, we run into an incomplete simulation run if guard <*add a new Presentation*> is chosen by the user. One way to repair this situation is to insert one or more triggering rules before rule *Person2Pres*(like e.g. in Figure 6).

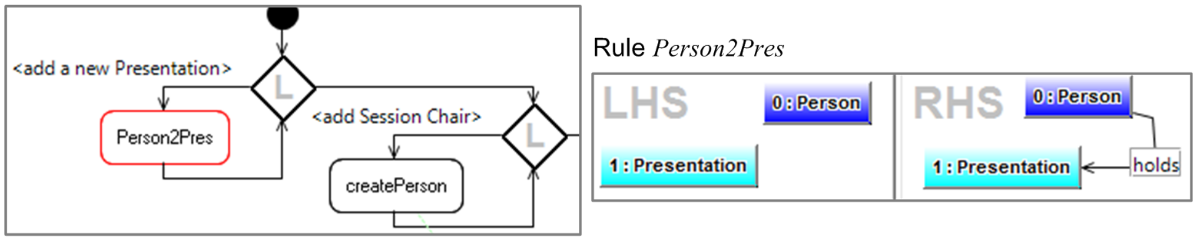


Fig. 14. Non-triggered rule *Person2Pres* along control flow in activity model $B2$

B.3 Inspecting Conflicts Along Control Flow Direction

Example 9. Consider the new draft part of activity model *GenConfPlan* in the left-hand side of Figure 15. Here, the intention is to allow also (controlled) deletion operations in the editing process of a conference plan. The editing extension is supposed to support the user in deleting a currently created time slot before other time slots are added and linked to the list of time slots.

A classical delete-use conflict is found for rule *deleteTimeSlot* with *addTimeSlot*. The detail view in the right part of Figure 15 shows the critical overlapping which could occur if the application of rule *addTimeSlot* tries to insert a *next* edge to the *Slot* node that has been deleted already by the application of rule *deleteTimeSlot*. This potential conflict gives us a hint that the activity model is not optimal: we could improve it e.g. by moving the *deleteTimeSlot* activity behind the *addTimeSlot*

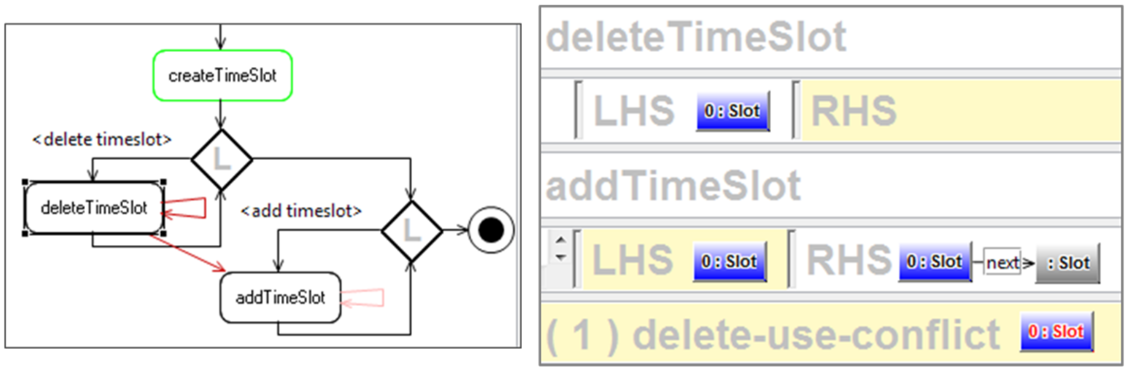


Fig. 15. Conflict of rule *deleteTimeSlot* with *addTimeSlot* in activity model *B3*

activity. Of course, then rule *deleteTimeSlot* must be adapted to reconnect also the *next* links of the deleted *Slot* node.

B.4 Plausibility Checking for Termination of Pattern-Guarded Loops: Motivation

In addition to the plausibility checks for consistency of integrated behavior models w.r.t applicability, we plan to design plausibility checks indicating positive or negative signs for the termination of pattern-guarded loops. Loops require particular attention, since the termination aspect needs to be regarded. Moreover, the control flow relation as well as the against-control-flow relation contain the same set of pairs of rules occurring within the loop body. This is demonstrated by the following example:

Example 10. Consider the activity model *moveEmptySlots* in the left part of Figure 16. This activity model is used after the scheduling of presentations to time slots is finished for moving slots not used for the scheduling to the beginning or the end of the slot queue. This is realized by removing in the first step an empty slot from the queue and re-linking its predecessor slot to its successor slot (rule *deleteUnusedSlot*). If such an empty slot has been found and removed from the queue, in the second step it is added either to beginning or to the end of the queue (rules *addSlotAtEnd, addSlotAtStart*). The activity *deleteUnusedSlot* in activity diagram *moveEmptySlots* may be triggered along control flow by the activities *addSlotAtStart* as well as *addSlotAtEnd*.

In our example, the trigger causality along control flow (*addSlotAtStart, deleteUnusedSlot*) indicates a potentially *unwanted* behavior: we may run into an infinite

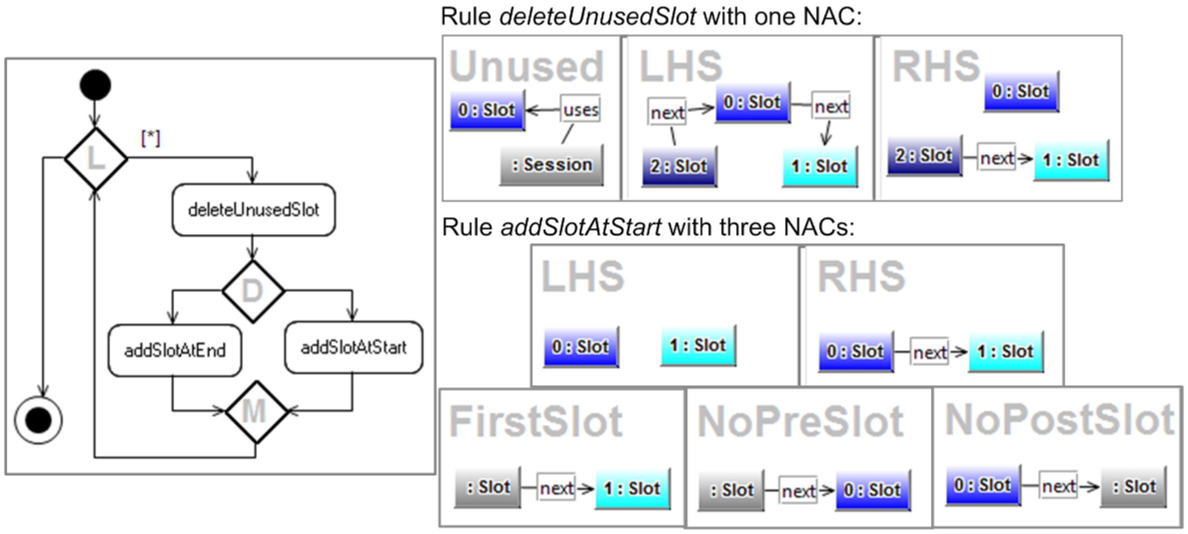


Fig. 16. Rule *deleteUnusedSlot* in activity model *moveEmptySlots* might be triggered by rule *addSlotAtStart* along control flow causing an infinite loop

loop when moving unused slots to the beginning (or end) of the slot queue. Inspecting the rules in the causality's detail view (see Figure 17), we find that we may have the following situation:

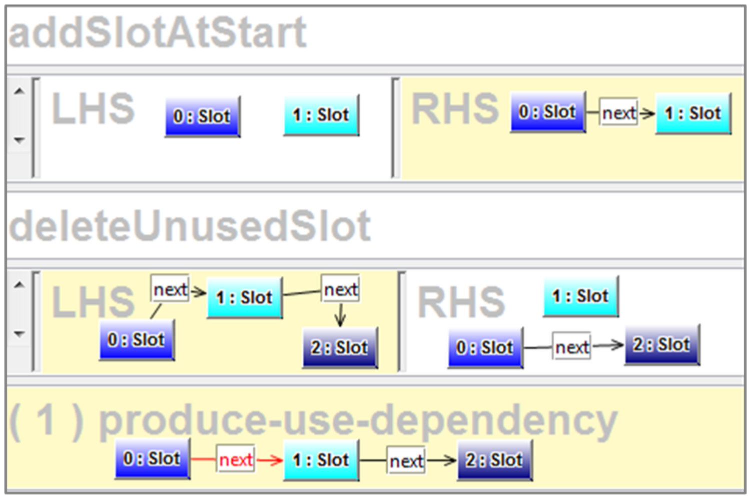


Fig. 17. Detail View of the trigger causality along control flow (*addSlotAtStart, deleteUnusedSlot*)

A slot is added to the beginning of the slot queue, where we now may have two consecutive empty slots. The second empty slot is found to be an ”unused” slot between two other slots, hence the loop is entered again, the slot is deleted and a slot is added to the beginning of the slot queue, etc. . A way to prevent this erroneous behavior is e.g. to mark newly added slots in rule *addSlotAtStart* by a boolean attribute that is checked by rule *deleteUnusedSlot*.

We plan to develop and implement a plausibility check indicating these kind of critical signs for the termination of pattern-guarded loops, highlighting critical trigger causalities between activities/guard patterns of a loop.