

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

Claudia Ermel¹, Jürgen Gall¹, Leen Lambers², and Gabriele Taentzer³

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

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

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

Abstract. UML activity diagrams are a wide-spread 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.

Keywords: graph transformation, activity model, plausibility, conflict and causality detection, object rule, AGG

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.

In [10], sufficient consistency criteria for the executability of integrated behavior models have been developed. 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]⁴. This light-weight technique seems to be very appropriate to allow for early plausibility reasoning during development steps of integrated behavior models. We visualize the results of our plausibility checks in an integrated development environment called ACTIGRA⁵. 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 un-

⁴ In [4], we explain in more detail how plausibility reasoning is related to the sufficient criteria in [10].

⁵ <http://tfs.cs.tu-berlin.de/actigra>

derlying 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.

2 Case Study: A Conference Scheduling System

This case study⁶ 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*.

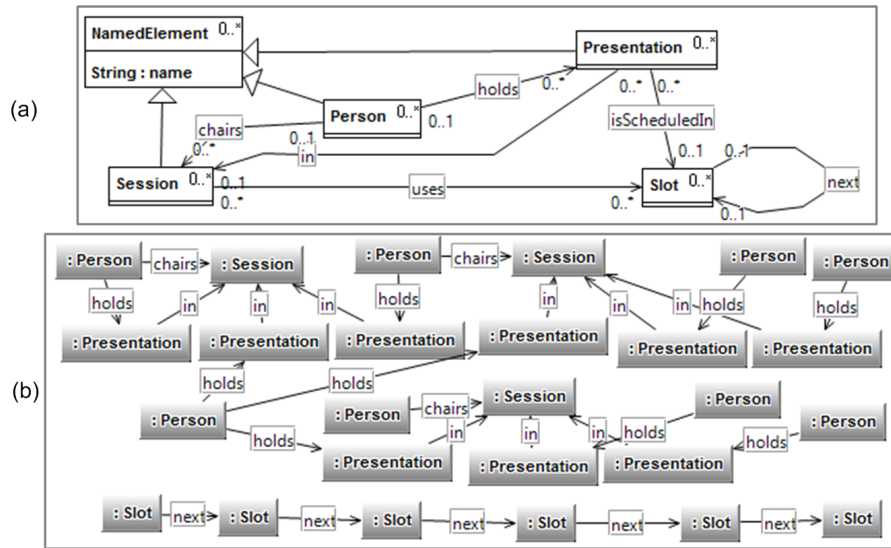


Fig. 1. Class and instance model for the *Conference Scheduling System*

Figure 1 (b) shows a sample object model of an initial session plan before presentations are scheduled into time slots⁷. 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

⁶ taken from the tool contest on *Graph-Based Tools 2008* [17]

⁷ Due to space limitations, we do not show name attributes here.

running simultaneously, (3) nobody chairs two sessions simultaneously, and (4) the presentations in one session are given not in parallel but in consecutive time slots. 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 [16]. 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.

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. 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 slot⁸.

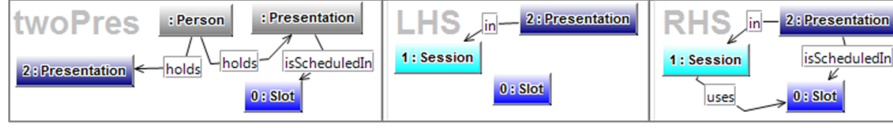


Fig. 2. Graph rule *initial-schedule*

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 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 (parallel independence). If this is not the case, then we say that the corresponding rules are in *conflict*, since one rule *may disable* the other rule. If two rules are applicable one after the other to the same graph, it might be possible to switch their application order without changing the result (sequential independence). 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. See [13] for a formal description of conflict and causality characterizations⁹.

The static analysis of potential conflicts and causalities between rules is supported in AGG¹⁰, a tool for specifying, executing and analysing graph transformation systems. This analysis is based on critical pair analysis (CPA) [2,8] and critical sequence analysis (CSA) [13], respectively. 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.

3.2 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

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

⁹ The different types of conflicts and causalities are reviewed also in [4].

¹⁰ AGG: <http://tfs.cs.tu-berlin.de/agg>

between s and B and another one between B and e . 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.

In our formalization, 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 3 (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.

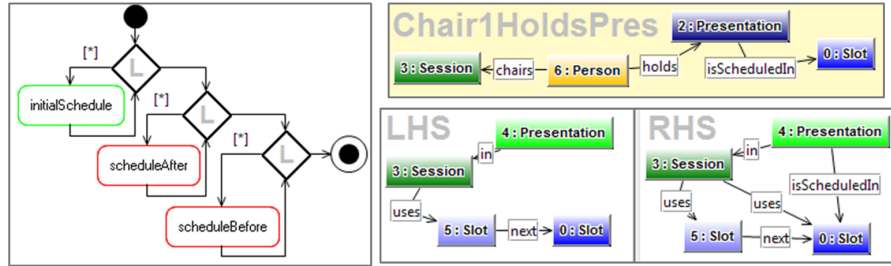


Fig. 3. Activity model *ScheduleControl* and rule *scheduleAfter*

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 3. 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.¹¹ 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 .

3.3 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 [11]).¹² 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 $S_B = S_X \text{ seq } S_Y$, 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 [13,2].¹³ 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 , $S_B = Sem(X)$.

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

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

¹² 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.

¹³ 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.

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 $s \in \text{Sem}(A)$ leading to a complete simulation run. In particular, if A contains user-defined guards, usually more than one complete simulation run should exist.

In ACTIGRA we can execute simulation runs on selected activity models. Chosen activities are highlighted and 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.

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 [4] between rules and guards occurring in the sequences of $\text{Sem}(A)$. Since in A simple activities, fork/joins as well as simple guard patterns correspond to rules¹⁵ 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 $\text{Sem}(A)$. 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 $\text{Sem}(A)$ the first rule is applicable, then the corresponding sequence can lead to a complete simulation run. Otherwise, the correspond-

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.

ing 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: $FaI_A = \{r | r \text{ is first rule of sequence in } Sem(A) \text{ and } r \text{ is applicable to } S\}$. We identify the *critical signs* as the set of impossible initializations: $CrI_A = \{r | r \text{ is first rule of a sequence in } Sem(A) \text{ and } r \text{ is not applicable to } S\}$.

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*¹⁶.

Example 2. Let us assume the system state in Figure 1 (b) as initial state. Figure 3 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: $FaTrAl_A(a) = \{(b, a) | (b, a) \in CFR_A \text{ such that } b \text{ may trigger } a\}$. We say that $FaTrAl_A = \{FaTrAl_A(a) | a \text{ is a rule in } A\}$ 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: $CrNonTrAl_A = \{a | a \text{ is rule in } A \text{ such that } FaTrAl_A(a) = \emptyset \text{ and } a \text{ is not applicable to } S\}$.

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

¹⁶ Concerning fork/join blocks in FaI_A or CrI_A , ACTIGRA colors the fork bar.

Example 3. Consider activity model *GenConfPlans* in (Figure 4) 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*.

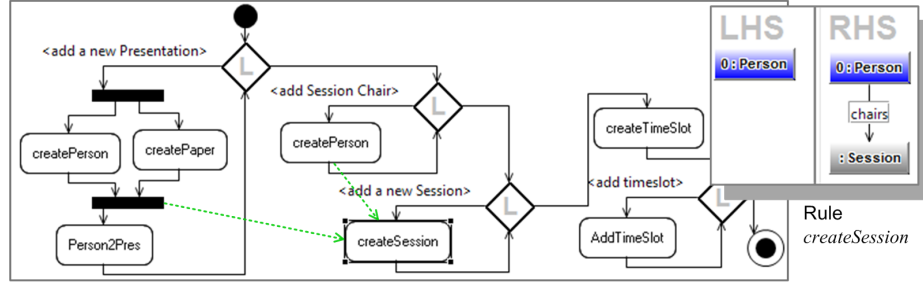


Fig. 4. Potential trigger causalities along control flow in activity model *GenConfPlans*

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

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 5 (a). Here, the set of potential conflicts along control flow caused by rule *initialSchedule* is given

by $CrDisAl_{SchedulingControl}(initialSchedule) = \{(initialSchedule, initialSchedule), (initialSchedule, scheduleAfter), (initialSchedule, scheduleBefore)\}$ ¹⁷. This gives the modeler 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 5 (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 3.

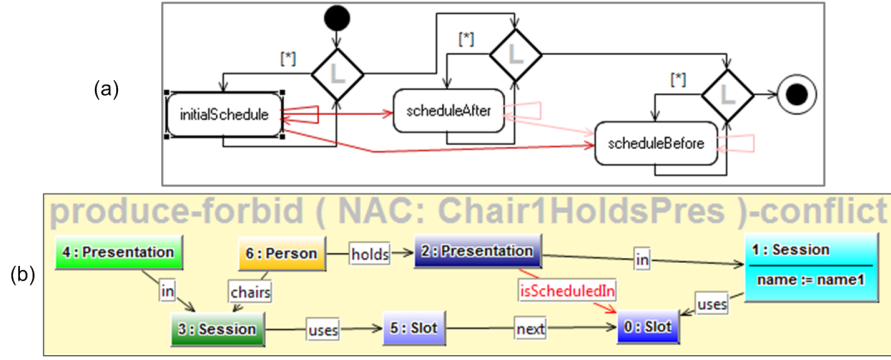


Fig. 5. (a) Potential conflicts along control flow caused by rule *initialSchedule*; (b) Detail view of potential conflict of rule *initialSchedule* with rule *scheduleAfter*.

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 : $CrTrAg_A(a) = \{(a, b) | a, b \text{ rules in } A \text{ and } (a, b) \in ACFR_A \text{ such that } a \text{ may trigger } b\}$. We say that $CrTrAg_A = \{CrTrAg_A(a) | a \text{ is a rule in } A\}$ 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: $FaNoTrAg_A = \{a | a \text{ is rule in } A \text{ and } \nexists(b, a) \in CrTrAg_A\}$.*

¹⁷ Note that one pair in this set may indicate more than one conflict potentially occurring between the corresponding rules.

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 *GenConfPlans* in Figure 6, we get the set of potential causalities against control flow $CrTrAg_{GenConfPlans}(createSession) = \{(createSession, Person2Pres)\}$. The causality $(createSession, Person2Pres)$ indicates that rule *Person2Pres* might be modelled too early in the control flow since rule *createSession* is needed to trigger rule *Person2Pres* completely.

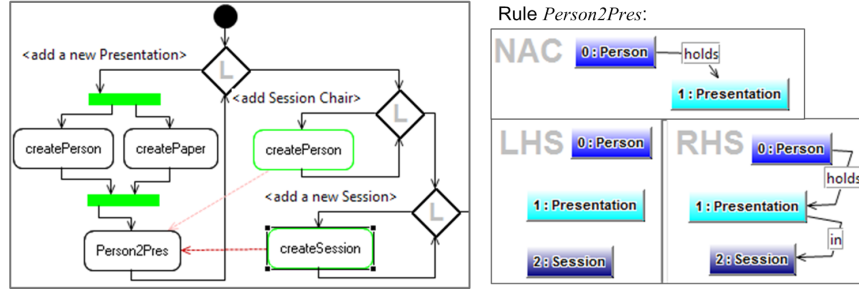


Fig. 6. Trigger causality against control flow $(createSession, Person2Pres)$

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:

$CrFJCa_A = \{(a, b) | (a, b) \in FJR_A \text{ and } (a, b) \text{ causally dependent}\}$.¹⁸ We identify the *favorable signs* as the set of fork/join structures with independent branches: $FaFJNoCa_A = \{fj | fj \text{ is fork/join in } A \text{ and } (a, b) \notin CrFJCa_A \text{ for each } (a, b) \text{ with } a, b \text{ in different branches of } fj\}$.

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

5 Related Work

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 [18] 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. 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.

¹⁸ 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 [13].

Fujaba [6], VMTS¹⁹ and GReAT²⁰ 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. 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 [15], and aspect-oriented modeling [14]. Although sometimes embedded in explicit control flow, it has not been taken into account for inconsistency 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. 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. The Eclipse plug-in ACTIGRA 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 [12].

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. Moreover, we plan to implement and visualize the sufficient criteria for consistency [10] in ACTIGRA. 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.

¹⁹ Visual Modeling and Transformation System: <http://vmts.aut.bme.hu/>

²⁰ Graph Rewriting and Transformation: <http://www.isis.vanderbilt.edu/tools/great>

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. AG-TIVE*. 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. *Tech. Rep. 2011/2*, TU Berlin (2011), <http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/>
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: *Proc. MoDELS*. 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., 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)
13. Lambers, L.: *Certifying Rule-Based Models using Graph Transformation*. Ph.D. thesis, Technische Universität Berlin (2009)
14. 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)
15. 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)
16. Object Management Group: *Unified Modeling Language: Superstructure – Version 2.3* (2010), <http://www.omg.org/spec/UML/2.3/>, formal/07-02-05
17. 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)
18. Störrle, H.: Semantics of UML 2.0 activity diagrams. In: *VLHCC*. IEEE (2004)