Technische Universität Berlin

Forschungsberichte der Fakultät IV – Elektrotechnik und Informatik

Object Flow Definition for Refined Activity Diagrams: Long Version

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

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

Bericht-Nr. 2009-01
ISSN 1436-9915

Object Flow Definition for Refined Activity Diagrams: Long Version

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

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

Abstract. Activity diagrams are a well-known means to model the control flow of system behavior. Their expressiveness can be enhanced by using their object flow notation. In addition, we refine activities by pairs of pre- and post-conditions formulated by interrelated object diagrams. To define a clear semantics for refined activity diagrams with object flow, we use a graph transformation approach. Control flow is formalized by sets of transformation rule sequences, while object flow is described by partial dependencies between transformation rules. The theory of algebraic graph transformation can be used to validate the consistency of control and object flows in refined activity diagrams. This approach is illustrated by a simple service-based on-line university calendar.

1 Introduction

UML2 activity diagrams are a well-known means to model the control flow of system behavior. Their expressiveness can be enhanced by using their object flow notation. Currently, it is an open problem how to formalize coherent object flow for activity diagrams. In this paper we aim at providing a precise semantics for refined activity diagrams with coherent object flow. We use graph transformation as semantic domain, since it supports the integration of structural and behavioral aspects and provides different analysis facilities.

In [1], sufficient criteria for the consistency of *refined* activity diagrams were provided, where interrelated object diagrams are used to specify pre- and post-conditions of single activities. All conditions refer to a domain class model. This refinement improves the integration of behavioral and structural modeling aspects and serves as a basis for consistency analysis. The refinement of activities by pre- and post-conditions was first introduced in [2] to analyze inconsistencies between individual activities refining use cases. Pre- and post conditions are formalized as graph transformation rules and the consistency analysis is rooted in critical pair analysis. However, this approach does not take into account the control flow of activity diagrams. Mehner et al. extend the consistency analysis in [3] where also the control flow is taken into account. In Lambers et al. [4], a similar approach for consistent integration of life sequence charts (LSCs) with graph transformation, applied to service composition modeling, was developed. The formalization based on graph transformation is used to analyze rule sequences. In addition, data flow is modeled textually by name equality for input and output variables.

Other approaches have formalized activity diagrams, e.g. [5–7], but do not consider object flow nor our proposed refinement.

In this paper, we extend refined activity diagrams by object flow. We introduce the notion of partial rule dependencies to formalize the semantics of object flow. Based on the already existing consistency notion of refined activity diagrams in [1], we define and validate the desirable consistency-related properties of refined activity diagrams with object flow.

We illustrate our approach with an example from model-driven development of a service-based web university calendar. In particular the behavior modeling of individual services still lacks advanced support for precise modeling and subsequent consistency analysis. Activity diagrams are an adequate means for modeling individual services, and the use of object flow and pre-/post-conditions can define service behavior more precisely.

This paper is organized as follows. Section 2 presents our motivating example and introduces the syntax and semantics of refined activity diagrams with object flow informally. Section 3 provides the formal background of algebraic graph transformations and the new notion of partial rule dependency. Section 4 presents the graph transformation based semantics and consistency notion of refined activity diagrams and extends it for object flow. Moreover, it analyzes the consistency of the example. Sections 5 and 6 contain related work and concluding remarks.

2 Introduction to Refined Activity Diagrams with Object Flow

This section introduces refined activity diagrams with object flow and illustrates this modeling approach by a small example for a service-based web university calendar. In this example, we model services by activity diagrams with object flow where each activity is refined by pre- and post-conditions, and guards are refined by patterns.

2.1 Domain Model

Our example application manages course parts that are lectures, laboratories, and exercises where a lecture may offer a laboratory and an exercise. Each course part is held by a lecturer and can be located in a room. An appropriate class diagram is presented in Fig. 1. From an abstract class *Object* [4], three classes are derived: *Room*, *Lecturer* and *CoursePart*. The latter is abstract and is specialized by three further classes: *Laboratory*, *Exercise* and *Lecture*. Day and time information for course parts are realized by enumerations *Day* and *Time*.

2.2 Activity Diagrams with Object Flow

We use UML2 activity diagrams with object flow [8] to model services of the university calendar. Three services, *AddLecture*, *AddExercise*, and *AddLaboratory*, are shown exemplarily in Fig. 2.

[4] Italic class names in diagrams indicate abstract classes.

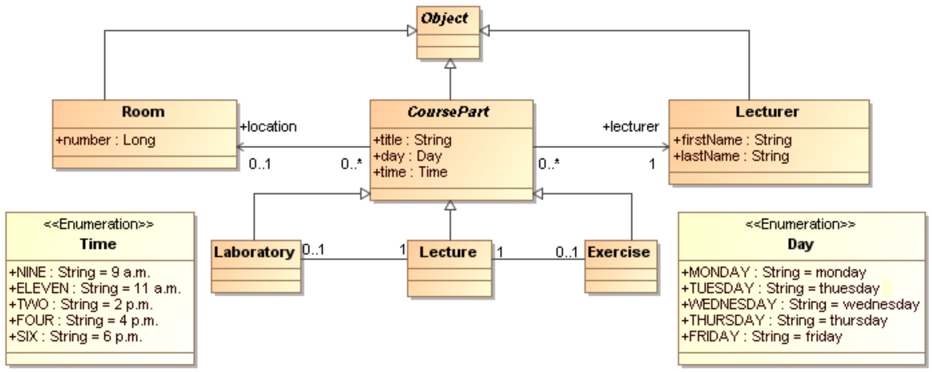

Fig. 1. Domain Class Diagram

Web applications usually contain a number of services. A service provides a clearly defined logical unit of functionality based on data entities. While a basic service might be modeled by one activity only, more complex services might contain a number of different activities. Defining services by the means of hierarchical activity diagrams opens up the possibility to call services from other ones. The usage of other services is depicted by placing a complex activity as representation of the used service into the control flow. In accordance with the UML2 notation, the invocation of a complex activity is indicated by placing a rake-style symbol within the activity node. Hierarchies can be resolved by flattening which of course requires the non-existence of cycles. Our example service *AddLecture* uses two other services. Accordingly, the complex activities modeling used services *AddLaboratory* and *AddExercise* are refined by corresponding activity diagrams (cf. Section 4). This can be done quite intuitively by replacing the complex activities with their activity diagram's content respecting the mapping of the diagram's parameters and the uniqueness of object names.

UML2 provides several object flow notations. The preference for a notation depends on different aspects, e.g. the amount of information, potential ambiguities, and the equality of control and object flow. For example, if object and control flow overlap, related objects may be depicted next to transitions as shown above activity *SetRoom* in Fig. 2. Otherwise an object node with separate object flow edges has to be used as shown for lecture *l*. However, it is desirable to keep the object flow description as simple as possible without leaving out important information. Each object may be named and its identity is expressed by equal names within an activity diagram. E.g. in activity diagram *AddLecture* both lecturer nodes named *l2* depict the same object. Names may also be used to refer to a certain object in terms of parameter values which is explained later on. Please note that in our approach, an object may flow along multiple outgoing edges i.e. object flows, whereas in UML2 one object serves one object flow exclusively.

Objects passed from outside to an activity diagram can be drawn on the diagram boundary in order to show parameters flowing into certain activities. Objects passed out of the diagram itself, may be depicted as boundary objects as well. Consider Fig. 2:

Objects of types *Lecturer* and *Room* are passed to the activity diagram *AddLecture*, while a newly created object of type *Lecture* is passed out of this diagram.

In Fig. 2, service *AddLecture* uses two other services *AddLaboratory* and *AddExercise*. Once a lecture has been created and its attributes have been set, a related laboratory or exercise might be created additionally. Activity diagram *AddLecture* requires three objects as input, activities *CreateLecture*, *AddLaboratory* and *AddExercise*: one object of type *Lecturer* and two of type *Room*. At first, a new lecture is created in activity *CreateLecture*, its attributes are set and it is linked to lecturer *l1*. If, moreover, room *r1* is not null, activity *SetRoom* is used to link this room *r1* to the lecture newly created. If a lecturer is given for a laboratory, the complex activity *AddLaboratory* is used to add a laboratory to the lecture. Therefore, *AddLecture* has to pass the newly created lecture *l*, lecturer *l2*, and room *r2* to the activity. In diagram *AddLaboratory* a new laboratory is created by the first activity *CreateLaboratory*. In the same step, this laboratory is linked to lecture *l* and to lecturer *l2*. Furthermore, the laboratory's attributes are set. In the next activity, the laboratory's location is set to room *r2*, provided that *r2* is given. If a lecturer for a related exercise is given, *AddExercise* is used by *AddLecture* analogously.

Since our activity diagrams model services, we equip each of them with a name and a comma-separated list of *parameters*. The semantics follow the programming concept of parameter passing between operations, i.e. an activity diagram models an operation consisting of a signature and a body. The signature of an activity diagram consists of its name and a list of attribute and object parameters. While object parameters have a type occurring in the domain model, attribute parameters have primitive types in most cases. This signature is an extension of UML2 made by our approach. Please note that all attributes and boundary objects used within the activity diagram are arguments which correspond to the signature of the diagram. Local objects i.e. objects created or selected by contained activities which are not used as output parameters do not occur in the signature. In addition, each parameter declaration has to be enriched with keyword *in*, *out*, or *inout*. This qualification defines the object flow direction. E.g. lecturer *l1* has to be passed to diagram *AddLecture* and is therefore marked *in*. Vice versa, the newly created lecture *l* is passed out of the diagram and is therefore marked by *out*. Parameter objects marked by *inout* are both input and output objects.

2.3 Refined Activities

Activities are used to model specific changes of the current system snapshot i.e. object structure. We propose to refine activities by pre- and post-conditions specifying snapshots before and after the activity respectively. Since the integration of conditions into activity diagrams, e.g. in terms of constraints as provided by UML2, would negatively affect readability and clarity, we refine activities separately by pairs of object diagrams which are typed over the domain model. Figure 3 shows object diagrams refining activities of our example (cf. Fig. 2) where pre-conditions are depicted on the left and post-conditions on the right. Objects and links with equal names on both sides express identity and preservation. Names for links have been omitted for better readability. Objects and links occurring in the left-hand side only will be deleted, while objects and links occurring in the right-hand side only will be created. Conditions on non-existence of patterns are depicted in red dashed outline.

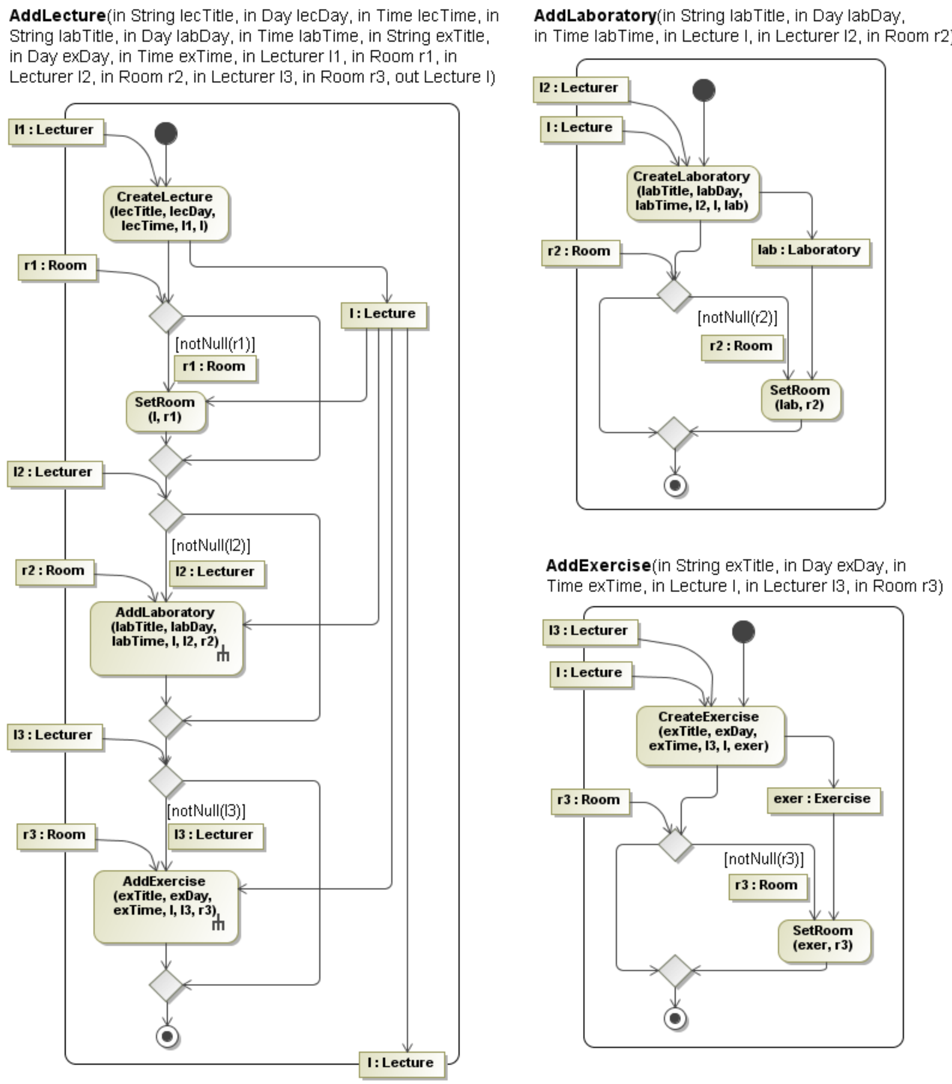

Fig. 2. Activity Diagrams of Services *AddLecture* and *AddLaboratory*

Each pair of conditions exhibits a signature according to the inscription of its refined activity, i.e. it consists of a name (the activity name) and a list of typed parameters qualified with keyword *in*, *out* or *inout*. Parameters can be distinguished into object and attribute parameters, analogously to their usage in activity diagrams. While the former ones are matched to objects, the latter ones are used as attribute values. Keyword *in* requires the occurrence of the related object (if object parameter) on the left-hand side. The object may be used in a read, edit, or delete operation. Keyword *out* declares

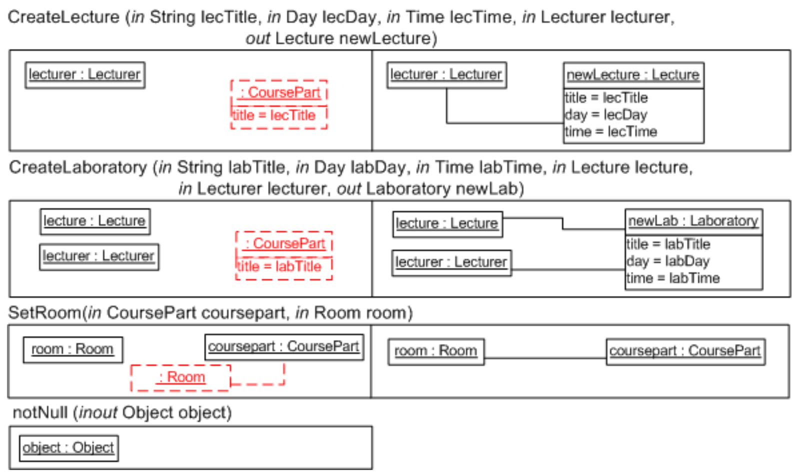

Fig. 3. Refined Activities by Pre- and Post-Conditions

a returned object and requires its presence on the right-hand side. It may be used for a create or select operation. *Inout* declares an object to be given and returned as well, thus requires the given object on both sides which explicitly guarantees its non-deletion. Attribute parameters have to be input parameters and may be used within pre- and post-conditions. If occurring in pre-conditions, attribute parameter values restrict the matching of objects, occurring in post-conditions they are used to assign attribute values. Note that in terms of compatibility with type inheritance, object parameter types must be respected by condition checking, i.e. by pattern matchings. Parameters may be matched, if they are matched with equally typed or sub-typed values only. Analogously, this must hold for attribute types. Note that arrays and collection-like types are not supported by our approach yet.

The first pair of conditions in Fig. 3 refines activity *CreateLecture*. The pre-condition requires the existence of a lecturer in the current system snapshot, otherwise the activity cannot be applied. Also, it requires the non-existence of a *CoursePart* instance (which could be of concrete type *Lecture*, *Exercise*, or *Laboratory*) with a *title* equal to the value of given attribute parameter *lecTitle*. If both conditions hold, the activity is applicable and creates a *Lecture* instance associated with the given *Lecturer* instance and the lecture is returned. The refinement of activity *CreateLaboratory* shown as second pair in Fig. 3 is quite similarly, but it requires two given objects to exist and leads to the creation of an object of type *Laboratory*. Since the conditions of *CreateExercise* are analogous to those of *CreateLaboratory*, they are left out. The refinement of activity *SetRoom* is shown as third pair. It requires two object parameters, one instance of type *Room* and one of type *CoursePart*, and it forbids the *CoursePart* instance to have a room already. No object but a link between the given course part and the new room is created here. Please note, that *CoursePart* is an abstract type. Thus instances of its concrete sub-classes can be used here only The last condition in Fig. 3 refines guard *notNull*. Since guards do not perform model-changing transformations but rather check for existence in the system snapshot, we just define a guard pattern here. Note that we disallow non-existence conditions in guard patterns. Else-guards are predefined by negated guard patterns i.e. it is checked for non-existence of the corresponding guard pattern.

3 Formalization by Graph Transformation

The UML variant presented in the previous section can be equipped with a graph transformation semantics. We start with the theory of graph transformation as presented in [9] and extend it by new concepts. While class diagrams are formalized by type graphs, activities with pre- and post-conditions are mapped to graph rules. The object flow is formalized by a new concept called *partial rule dependencies*. This semantics definition serves as a basis for validating the consistency of refined activities with object flow precisely. First, we recall the basic concepts in a condensed form.

3.1 Graphs and Graph Transformation

Graphs are often used as abstract representation of visual models, e.g. UML models. When formalizing object-oriented models, graphs occur at two levels: the type level (defined by a meta-model) and the instance level. This idea is described by the concept of *typed attributed graphs*, where a fixed *type graph* TG serves as an abstract representation of the meta-model (without constraints). Node types can be structured by an inheritance hierarchy [10] and may be abstract in the sense that they cannot be instantiated. Multiplicities and other annotations have to be expressed by additional graph constraints. Attribute types are formally described by data type algebras. Instances of the type graph are *object graphs* equipped with a structure-preserving mapping to the type graph. Attribute values are given by a concrete data algebra.

Graph transformation is the rule-based modification of graphs. A *rule* is defined by $p = (L \xleftarrow{l} K \xrightarrow{r} R, I, O, NACs)$ where L is the left-hand side (LHS) of the rule representing the pre-condition and R is the right-hand side (RHS) describing the post-condition. l and r are two injective graph morphisms, i.e. functions on nodes and edges which are structure and type-preserving. They specify a partial mapping $r \circ l^{-1}$ from L to R. $L \backslash l(K)$ defines the graph part that is to be deleted, and $R \backslash r(K)$ defines the graph part to be created. All newly created nodes have to be of concrete types. Elements in K are mapped in a type preserving way. All graphs of a rule are attributed by the same algebra being a term algebra with variables. Some of these variables are considered to be rule parameters. Input parameters can be nodes or variables, thus $I = I_N \cup I_V$, whereas output parameters can be nodes only, i.e. $O = O_N$ with $I \subseteq L$ and $O \subseteq R$. A rule is called *node preserving*, if it does not delete nodes.

NACs is a set of negative application conditions, each defined by an injective graph morphism $n : L \rightarrow N$ where $N \setminus n(L)$ defines a forbidden graph part. n allows to refine node types, i.e. a node of a more abstract type is allowed to be mapped to a node with a finer type according to the inheritance hierarchy.

Example 1 (Example rules). Figure 3 shows example graph rules where each pre-condition forms an LHS with one negative application condition and each post-condition describes an RHS. Identifiers given by names indicate the mapping between left- and right-hand sides. The solid parts of a pre-condition indicate the LHS L, while the dashed ones prohibit a certain graph part and represent $N \setminus n(L)$ of the NAC. Input and output parameters are listed on top of each pair of conditions, formally in the head of each rule.

A *graph transformation step* $G \stackrel{p,m}{\Longrightarrow} H$ between two instance graphs G and H is defined by first finding a match $m : L \to G$ of the left-hand side L of rule p into the current instance graph G such that m is an injective type-refining graph morphism. Match m has to fulfill the *dangling condition*, i.e. nodes may be deleted only, if all adjacent edges are mentioned in the LHS. Moreover, each NAC has to be fulfilled, i.e. m satisfies a NAC, if for each $n \in NACs$ there does not exist an injective type-refining morphism $o : N \to G$ such that $o \circ n = m$. Input parameters are instantiated by concrete values being nodes of the instance graph and data type values. Thus, parameter instantiation provides a partial match.

In the second step, graph H is constructed by a double-pushout construction (see [9]). Roughly spoken, the construction is performed in two passes: (1) build a graph D which contains all those elements of G not deleted; (2) construct H as a union of D and all elements of R to be created. To trace the preserved part of a graph transformation step, we define a partial graph morphism $track : G \to H$ by $track = h \circ g^{-1}$. Graph $dom(track)$ is the subgraph of G where $track$ is defined, i.e. the domain of $track$. (See also [11] for a first definition of track morphism.) Morphisms $g : D \to G$ and $h : D \to H$ are constructed by a double-pushout as shown below. Morphism g^{-1} is always well-defined, since l is injective and the pushout construction preserves injectivity, thus g is also injective. Furthermore, a so-called co-match $m' : R \to H$ is defined by the double-pushout construction. Output parameters point to a certain part of this co-match. Output parameters are useful for pointing to nodes which shall be used in further transformation steps.

$$\begin{tikzcd}
I \arrow[r, "\subseteq"] & L \arrow[d, "m"'] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] & R \arrow[d, "m'"] & O \arrow[l, "\subseteq"'] \\
& G \arrow[rr, bend right, "track"'] & D \arrow[l, "g"'] \arrow[r, "h"] & H &
\end{tikzcd}$$

A *graph transformation (sequence)* $t = G_0 \stackrel{p_1,m_1}{\Longrightarrow} G_1 \dots G_{n-1} \stackrel{p_n,m_n}{\Longrightarrow} G_n$ consists of zero or more graph transformation steps. Track morphism $track_{0,n}$ of sequence t is simply the composition of track morphisms $track_{n-1,n} \circ \dots \circ track_{0,1}$ of its steps. For $n = 0$, $track_{0,0} = id_{G_0}$. A set of graph rules P, together with a type graph TG, is called a *graph transformation system* (GTS) $GTS = (TG, P)$. A GTS may show two kinds of non-determinism: Given a graph, (1) several rules can be applicable, and (2) for each rule several matches can exist. There are techniques to restrict both kinds of choices. The choice of rules can be restricted by the definition of control flow (e.g. expressed by activity diagrams), while the choice of matches can be restricted by passing partial matches (e.g. expressed by partial rule dependencies). The tool AGG (Attributed Graph Grammar System) [12] can be used to specify and analyze graph transformation systems.

3.2 Partial Rule Dependencies

To restrict the choice of matches for rules, we introduce the concept of *partial rule dependencies* which may relate output parameter nodes of one rule to input parameter nodes of a (not necessarily direct) subsequent rule in a given rule sequence[5]. Especially, two partial rule dependencies may be composed by a common rule in the middle forming a new transitive dependency. This may lead to a situation where different partial rule dependencies may be defined between the same rules. We say that rule sequences are dependency-compatible, if the transitive closure of all dependencies between each two rules is well-defined, i.e. if all dependencies are unambiguous and conforming to the type hierarchy.

Definition 1 (partial and joint rule dependencies). *Given a GTS* (T, P) *and a rule sequence* $s : p_1, ..., p_n$ *with* $p_1, ..., p_n \in P$*. A* partial rule dependency *between rules* p_i *and* p_j *with* $1 \leq i < j \leq n$ *is defined by an injective partial morphism* $d_{ij} : O_{i_N} \rightarrow I_{j_N}$ *from output parameter nodes of* p_i *to input parameter nodes of* p_j*. If* d_{ij} *is the empty morphism, no rule dependency is defined. For each pair of rules* p_i *and* p_j *in* s*, its* $closure_{ij}$ *is defined as follows:*

(1) d_{ij} *belongs to* $closure_{ij}$
(2) For all d_{ik}*,* d_{kj}*, and rules* p_k *with* $i < k < j$ *add* $d_{kj} \circ r_k \circ l^{-1}_{k|I_k} \circ d_{ik}$ *to* $closure_{ij}$*.*

$$\begin{tikzcd}
O_{i_N} \arrow[r, "d_{ik}"'] \arrow[d, "\subseteq"] & I_{k_N} \arrow[d, "\subseteq"'] & & O_{k_N} \arrow[r, "d_{kj}"'] \arrow[d, "\subseteq"'] & I_{j_N} \arrow[d, "\subseteq"'] \\
R_i & L_k & K_k \arrow[l, "l_k"'] \arrow[r, "r_k"] & R_k & L_j
\end{tikzcd}$$

Rule sequence s *is* dependency-compatible*, if for all closures* $closure_{ij}$ *the following holds:*

(3) For all $d \in closure_{ij}$*:* $type(x)$ *has to be finer or coarser than* $type(d(x))$ *for all* $x \in O_{i_N}$ *wrt. the type inheritance relation defined by type graph* T*.*
(4) Each two dependencies d *and* d' *in* $closure_{ij}$ *are weakly commutative, i.e.* $d(x) = d'(x)$ *for all* $x \in dom(d) \cap dom(d')$*.*

If rule sequence s *is dependency-compatible, we can define a* joint dependency *of a closure. Given* $closure_{ij}$ *we define the* joint dependency $dep_{ij} : O_{i_N} \rightarrow I_{j_N}$ *as follows:*

(5) $dom(dep_{ij}) = \bigcup_{d \in closure_{ij}} dom(d)$
(6) $dep_{ij}(y) = d(y)$ *if* $y \in dom(d)$ *for* $d \in closure_{ij}$

Please note that each closure $closure_{ij}$ is constructed as a set of injective partial morphisms. It can be shown that each dep_{ij} is an injective partial morphism because weak commutativity holds for the elements in the closure and because each element in the closure is an injective partial morphism.

In the following, we discuss several examples for partial rule dependencies.

[5] Note that rule sequences differ from transformation sequences in not providing graphs to which rules are applied.

Example 2 (partial rule dependencies). Considering the rules in Fig. 3, we compose rule sequence $s =$*CreateLecture, SetRoom, CreateLaboratory, SetRoom*. As first step, we define partial rule dependencies taking input and output parameters into account: $d_{12}(newLecture) = coursepart, d_{23} = d_{34} = d_{24} = d_{14} = \emptyset, d_{13}(newLecture) = lecture$. All dependencies are type-compatible, since either the types of mapped nodes are equal or in hierarchy, e.g. $type(newLecture) = Lecture$ is finer than $type(d_{12}(newLecture)) = type(coursepart) = CoursePart$ (see Fig. 1). None of the considered closures contains more than one non-empty partial dependency. Thus, partial rule dependencies are not really composed from each other in this example, e.g. $dep_{13} = d_{13}$.

Example 3 (partial rule dependencies - part 2). We consider rule sequence s again, but slightly modify it. If $coursepart$ were an *inout* parameter of rule *SetRoom*, we could define $d_{23}(coursepart) = lecture$. Thus, $closure_{13}$ would look more interesting: $closure_{13} = \{d_{13}, d_{23} \circ r_2 \circ l_2^{-1} \circ d_{12}\}$ with $d_{23} \circ r_2 \circ l_2^{-1} \circ d_{12}(newLecture) = d_{23} \circ r_2 \circ l_2^{-1}(coursepart) = d_{23}(coursepart) = lecture$. Since d_{13} is equally defined for $newLecture$ and their type is $Lecture$, this closure is dependency-compatible and joint dependency dep_{13} is defined accordingly.

Example 4 (problematic rule dependencies). We consider rule sequence s again and modify it further. If $coursepart$ were an *inout* parameter of rule *SetRoom* and if we enlarged the rule sequence by rule *CreateLaboratory*, we would run into problems as follows. If we defined $d_{34}(newLab) = coursepart$ and $d_{45}(coursepart) = lecture$, then $d_{45} \circ r_4 \circ l_4^{-1} \circ d_{34}(newLab) = lecture$. This morphism is not type-compatible, since $type(newLab) = Laboratory$ and $type(lecture) = Lecture$, two incomparable types. Thus, the rule sequence extended in this way is not dependency-compatible.

Given a dependency-compatible rule sequence s, this sequence is applicable to a graph G_0, if there is a transformation sequence starting at G_0 and applying s such that it relates nodes according to partial dependencies and thus, restricts rule matches. This way certain transformation sequences are ruled out by restricting the choice of matches.

Definition 2 (application of dependency-compatible rule sequences). *A dependency-compatible rule sequence* $s : p_1, ..., p_n$ *is* applicable *to some graph* G_0*, if there is a graph transformation sequence* $G_0 \overset{p_1,m_1}{\Longrightarrow} G_1 \ldots G_{n-1} \overset{p_n,m_n}{\Longrightarrow} G_n$ *such that* $m_j \circ dep_{ij}$ *and* $track_{i,j-1} \circ m'_i(O_{i_N})$ *are weakly commutative, with* $track_{i,j-1}$ *being the track morphism from* G_i *to* G_{j-1} *and* m'_i *being the co-match of rule* p_i *for* $1 \leq i < j \leq n$*.*

$$
\begin{tikzcd}
O_{i_N} \arrow[r, "\subseteq"'] \arrow[rrrr, bend left, "dep_{ij}"] & R_i \arrow[d, "m'_i"] & & L_j \arrow[d, "m_j"] & I_{j_N} \arrow[l, "\supseteq"] \\
& G_i \arrow[rr, "track_{i,j-1}"'] & & G_{j-1} &
\end{tikzcd}
$$

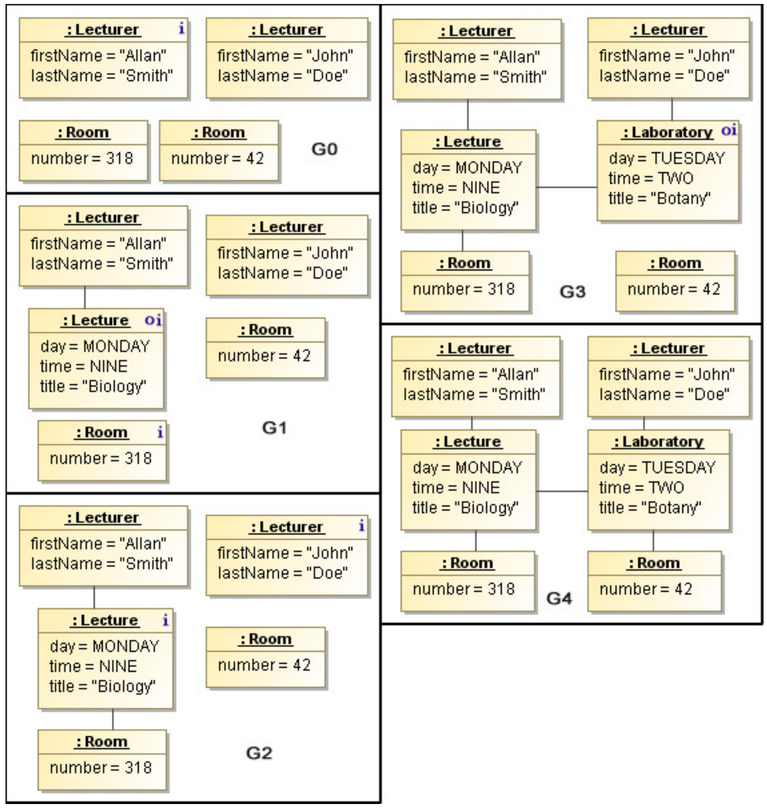

Fig. 4. Example graphs according to an application of sequence s of Example 2

Example 5. Figure 4 depicts a sequence of graphs according to a transformation of the dependency-compatible rule sequence s of Example 2, whereas G_0 is an exemplary start graph. We will concentrate on this sequence only for this example. Attribute parameters used by rules are omitted here. Please note the letters *i* and *o* in the upper right corner of several objects, which shall remind the reader of *i*nput objects to the next rule to apply and *o*utput objects of the rule just applied before.

Applying rule *CreateLecture* to graph G_0 results in graph G_1. One lecturer of the start graph was used as input parameter while the newly created lecture is delivered to the successor rule *SetRoom* as output parameter. Since *SetRoom* requires an input parameter of type *CoursePart*, the given lecture is compatible. The result of applying *SetRoom* is shown in G_1. The following rule *CreateLaboratory* leading to G_3 requires a *Lecture* object and a *Lecturer* object. As depicted in Fig. 2 and explained in Example 2, a partial rule dependency between *CreateLecture* and *CreateLaboratory* exists which provides a *Lecture* object to the latter rule. As lecturer the second object in the graph is used here. The last rule to apply is *SetRoom* which leads to graph G_4. This rule uses the output object *:Laboratory* of *CreateLaboratory* as input parameter. This is valid as *Laboratory* is a subtype of *CoursePart*.

Partial rule dependencies are defined independently of causal dependencies. Causal dependencies between rules can be analyzed by the critical pair analysis (CPA) [9]. The only kind of causal dependencies we are interested in here are *produce/use*-dependencies where the application of one rule produces an element needed by the match of a second rule. If two rules are not causally dependent on each other, the corresponding joint dependency which is defined explicitly must not introduce any produce/use-dependency. If some partial dependency is defined, relating produced objects from some rule with used ones from some successor rule, it has to correspond with at least one produce/use-dependency.

4 Object Flow: Semantics Definition and Properties

In this section, we first specify well-structured refined activity diagrams, refine their activities by graph rules and their guards by graph patterns, and define their semantics and consistency based on graph transformation. Thereafter, this approach is extended to refined activity diagrams with object flow.

From now on, we assume that an activity diagram does not contain any complex activities and that each complex activity has been flattened before, i.e. it has been replaced by its refining activity diagram. During this potentially recursive process, each object which goes in to or comes out from a complex activity is glued with the corresponding boundary object of the refining activity diagram, i.e. the boundary and boundary objects disappear.

4.1 Refined Activity Diagrams

As in [7, 1], we restrict our considerations to well-structured activity diagrams. The building blocks are simple activities, sequences, fork-joins, decision-merge structures, and loops only.

Definition 3 (well-structured activity diagram). *A* well-structured activity diagram *A consists of a start activity s, an activity block B, and an end activity e such that there is a transition between s and B and another one between B and e. An* activity block *is defined as follows:*

(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 decision 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 decision activity as above. Considering this decision 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 is guarded by "else". The whole construct forms an activity block.

(6) Fork/Join: *A fork activity followed by two branches with one activity block each followed by a join activity form an activity block.*

To be able to define object flow to be coherent with control flow we define a control flow relation as prerequisite. Because of potential loops it is not a partial order.

Definition 4 (control flow relation). *The* control flow relation CFR_A *of an activity diagram A contains pairs (x,y) where x, y are activities such that the following holds:*

(1) Pair $(x, y) \in CFR_A$, if x is directly connected via a transition with y.
(2) If $(a, b) \in CFR_A$ and $(b, c) \in CFR_A$, then also $(a, c) \in CFR_A$.

An if- or loop-guard is equipped with a graph pattern which describes an existence condition on graphs. A guard pattern can be interpreted as identical rule (i.e. a rule where the left and the right-hand sides are equal) with arbitrary input and output parameters. Guard pattern g is fulfilled by a graph G, if its corresponding rule p_g is applicable to G. After rule p_g has been performed, the guarded alternative is executed. Otherwise, rule $\bar{p}_g$ which formalizes "else" for given guard g, is applicable to G and the second alternative is performed.

Definition 5 (guard pattern, guard rule and negated guard rule). *A* guard pattern *g is defined by a typed graph being attributed over a term algebra with variables together with input and output parameters. Its* guard rule p_g *is defined by* $(g \xleftarrow{id_g} g \xrightarrow{id_g} g, I, O, \emptyset)$. *Its* negated guard rule $\bar{p}_g$ *is defined by* $(\emptyset \xleftarrow{\emptyset} \emptyset \xrightarrow{\emptyset} \emptyset, \emptyset, \emptyset, \{n : \emptyset \to g\})$.

Note that the negated guard rule should adopt in its negative application condition the input parameter set I given for graph g. As part of future work we plan to formalize guards (resp. negated guards) by the satisfaction (resp. non-satisfaction) of graph constraints instead of by applicability of specific rules.

Lemma 1. *Given a guard pattern g and a graph G. Rule p_g is applicable to G, iff rule $\bar{p}_g$ is non-applicable to G.*

Proof.

$\Rightarrow$: If p_g is applicable to G, there is a match $m : g \to G$ which is injective by definition. For $\bar{p}_g$ we always find a morphism to G, since its LHS is empty. However, this morphism cannot be a match, since there is an injective morphism m from g to G and thus, its NAC is not fulfilled.

$\Leftarrow$: If rule $\bar{p}_g$ is non-applicable to G, its NAC cannot be satisfied, since a morphism from its LHS can be found to any graph. Hence, there must be an injective morphism from g to G. For every identical rule, the dangling condition is always fulfilled. Therefore, this morphism is also a match for p_g which makes this rule applicable to G.

Now we can formally define well-structured refined activity diagrams where the pre- and post-conditions of each activity and where each guard is formalized by graph tranformation rules.

Definition 6 (refined activity diagram). *A* refined activity diagram RA *is a well-structured activity diagram such that each simple activity occurring in* RA *is equipped with a graph transformation rule. Each if- or loop-guard occurring in* RA *is equipped with a guard pattern. We also say that an activity is refined by a transformation rule where decision activities are refined by guard rules deduced from guard patterns which refine guards.*

Next, we give the semantics of well-structured refined activity diagrams. As each activity is formally refined by a rule, we define the semantics as sequences of rules, where each of the sequences is determined by the control flow of the activity diagram. Since loops are guarded by guard patterns, we also provide a restricted semantics assuming a fixed number of loop executions. Note that in [1] we considered only user-guarded loops.

Definition 7 (semantics of refined activity diagrams). *Given an activity block* B *of a refined activity diagram RA, its corresponding set of rule sequences* S_B *is defined as follows.*

(1) If B *is* empty, $S_B = \emptyset$.
(2) If B *consists of a* simple activity a *refined by rule* p_a, $S_B = \{p_a\}$.
(3) If B *is a* sequence *of* X *and* Y, $S_B := S_X\ seq\ S_Y = \{s_x s_y | s_x \in S_X \wedge s_y \in S_Y\}$
(4) If B *is a* decision *block on* X *and* Y *with guard pattern* g *refining its if-guard,* $S_B = (\{p_g\}\ seq\ S_X) \cup (\{\bar{p}_g\}\ seq\ S_Y)$
(5) If B *is a* loop *block on* X *with guard pattern* g *refining its loop-guard,* $S_B := loop(g, S_X) = \bigcup_{i \in I} S_X^i$ *where* $S_X^0 = \{\bar{p}_g\}$, $S_X^1 = \{p_g\}\ seq\ S_X\ seq\ \{\bar{p}_g\}$, $S_X^2 = \{p_g\}\ seq\ S_X\ seq\ S_X^1$ *and* $S_X^i = \{p_g\}\ seq\ S_X\ seq\ S_X^{i-1}$ *for* $i > 2$. $S_B(n) = S_X^n$ *denotes the semantics of loop block* B *with exactly* n *loop executions.*
(6) If B *is a* fork *block on* X *and* Y, $S_B := S_X || S_Y = \bigcup s_x || s_y$ *with* $s_x \in S_X \wedge s_y \in S_Y$ *where* $s_x || \lambda = \{s_x\}$, $\lambda || s_y = \{s_y\}$, *and* $p_x s'_x || p_y s'_y = \{p_x\}\ seq\ (s'_x || p_y s'_y) \cup \{p_y\}\ seq\ (p_x s'_x || s'_y)$.

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

Now, we are ready to check the control flow consistency of activity diagrams. To do so, we consider snapshots of the system, i.e. object models which are formalized as graphs by mapping objects to graph nodes and object links to graph edges. In the following definitions for consistency-related properties, we directly use graphs as abstract syntax representation of object models.

An activity diagram is *consistent*, if there is a set $\mathcal{S}$ of model graphs such that each rule sequence in the diagram semantics is applicable to some of these graphs. If the diagram contains guarded loops, we use the restricted semantics for diagrams (as defined above) which checks for each guarded loop, if a predefined number of loop executions is feasible. As prerequisite for consistency we define that S is complete if for all sequences in the diagram semantics there is a model graph in S to which they are applicable. $\mathcal{S}$ is *without junk*, if each of its model graphs represents a potential snapshot of the system to which a rule sequence in the diagram semantics can be applied.

Definition 8 (completeness). *A set $\mathcal{S}$ of graphs is* complete *wrt. to a refined activity diagram RA, if for all rule sequences s in $Sem(RA)$ there is a graph G in $\mathcal{S}$ such that s is applicable to G. If RA contains k guarded loops, a set $\mathcal{S}$ of graphs is* quasi-complete *wrt. to RA, if for all rule sequences s in $Sem_{n_1,\ldots,n_k}(RA)$ there is a graph G in $\mathcal{S}$ such that s is applicable to G. Set $\mathcal{S}$ is* without junk*, if for each graph in $\mathcal{S}$ at least one applicable rule sequence in $Sem(RA)$ (resp. $Sem_{n_1,\ldots,n_k}(RA)$) exists.*

Definition 9 (consistent activity diagram (without object flow)). *A refined activity diagram RA is* consistent*, if there is a set $\mathcal{S}$ of graphs which is complete wrt. RA. If RA contains k guarded loops, RA is* quasi-consistent*, if there is a set $\mathcal{S}$ of graphs which is quasi-complete wrt. RA.*

4.2 Refined Activity Diagrams with Object Flow

In the following, we extend refined activity diagrams by partial rule dependencies which formalize object flows. We extend the semantics of refined activity diagrams to cover also object flow.

First, we define well-structured activity diagrams with object flow. Object flow has to be coherent with the control flow of an activity diagram.

Definition 10 (well-structured activity diagram with coherent object flow). *A* well-structured activity diagram $A_{OF} = (A, Obj, OFR, I, O)$ with coherent object flow *is a well-structured activity diagram A (as given in Def. 3) equipped with a set of object nodes Obj, an object flow relation OFR for A and Obj, input parameter set I, and output parameter set O, defined as follows:*

(1) Input parameters can be object nodes or values, i.e. $I = I_N \cup I_V$ with $I_N \subseteq Obj$. Output parameters may only be object nodes only, i.e. $O = O_N$ with $O \subseteq Obj$.
(2) Object flow relation *OFR contains triples (x, o, y) where x and y are simple or decision activities and $o \in Obj$. In addition, there is a special tag* null *not used as activity name which is used to define object flow from and to parameter objects, i.e. triples $(null, o, y)$ and $(x, o, null)$ can also be in OFR where $o \in I_N$ or $o \in O_N$, resp. For each object o in I_N (resp. in O_N), there is at least one triple $(null, o, y)$ (resp. $(x, o, null)$) in OFR. For each other object $o \in Obj$, there has to be at least one triple $(x, o, y) \in OFR$.*
(3) OFR is coherent *with control flow relation CFR_A of A (see Def. 4), i.e. for all $(x, o, y) \in OFR$ with $x, y \neq null$ there is $(x, y) \in CFR_A$.*

Please note that OFR contains a triple for each pair of object flows sharing an object and Obj is not allowed to contain objects not involved in object flow.

Example 6. Considering activity diagram *AddLaboratory* shown in Fig. 2 which has a set of four object nodes $Obj = \{l2 : Lecturer, l : Lecture, lab : Laboratory, r2 : Room\}$ and an object flow relation $OFR = \{(null, l2 : Lecturer, CreateLaboratory), (null, l : Lecture, CreateLaboratory), (null, r2 : Room, DecisionActivity1), (CreateLaboratory, lab : Laboratory, SetRoom), (DecisionActivity1, r2 : Room, SetRoom)\}$. Please note that the unnamed decision activity is represented by *DecisionActivity1* here. The considered activity diagram has input parameter sets as follows $I_N = \{l2 : Lecturer, l : Lecture, r2 : Room\}$ and $I_V = \{labTitle : String, labDay : Day, labTime : Time\}$. The output parameter set O is empty as the signature of activity diagram *AddLaboratory* lists no parameter marked with keyword *out*. Coherence of the object flow relation OFR is satisfied as for triples $(DecisionActivity1, r2 : Room, SetRoom)$ and $(CreateLaboratory, lab : Laboratory, SetRoom)$ there are tuples $(DecisionActivity1, SetRoom) \in CFR_A$ and $(CreateLaboratory, SetRoom) \in CFR_A$ i.e. corresponding successive activities can be found in diagram *AddLaboratory*.

Next, we define how refined activity diagrams are extended coherently by object flow. The object flow relation for objects that do not serve as input or output parameters of the activity diagram has to be coherent with the input and output parameters of the rules refining activities. That means, each such object flow has to correspond with exactly one input and output parameter of an activity.

Definition 11 (refined activity diagram with object flow). *A* refined activity diagram RA_{OF} with object flow *is a well-structured activity diagram* $A_{OF} = (A, Obj, OFR, I, O)$ *with coherent object flow such that each simple activity* x *occurring in* A_{OF} *is refined by a graph transformation rule* p_x*. Each decision activity* $x \in A_{OF}$ *has an if- or loop-guard which is equipped with a guard pattern* g*. Its guard rule* p_g *is also denoted by* p_x*. Let* O_{p_x} *be the output parameter set of* p_x *and* I_{p_y} *the input parameter set of* p_y*.* OFR *has to be* coherent with refining rules *which is defined as follows:*

(1) For all $(x, o, y) \in OFR$ *where* $x \neq null$*, an output object parameter exists in* O_{p_x} *which is called* $src(x, o, y)$*. If* $y \neq null$*, an input object parameter exists in* I_{p_y}*, called* $tgt(x, o, y)$*.*
(2) For all triples $(x, o, y), (x, o, y')$ *(resp.* $(x, o, y), (x', o, y)$*) in* OFR *we have* $src(x, o, y) = src(x, o, y')$ *(resp.* $tgt(x, o, y) = tgt(x', o, y)$ *).*
(3) For each two activities x *and* y *and the set of all* $(x, o, y) \in OFR$*, the set of all pairs* $(src(x, o, y), tgt(x, o, y))$ *defines an injective mapping.*
(4) For all triples $(x, o, null), (x, o', null)$ *(resp.* $(null, o, y), (null, o', y)$*) in* OFR *with* $o \neq o'$ *we have* $src(x, o, null) \neq src(x, o', null)$ *(resp.* $tgt(null, o, y) \neq tgt(null, o', y)$*).*

Example 7. We consider activity diagram *AddLaboratory* (cf. Fig. 2) again with OFR as discussed in Example 6. According to the definition one example is shown each for src and tgt.

The evaluation of src with triple $(CreateLaboratory, lab : Laboratory, SetRoom) \in OFR$, with respect to its source activity $CreateLaboratory(labTitle, labDay, labTime, l2, l, lab)$ and its refining rule $CreateLaboratory(in\ String\ labTitle,$

$in\ Day\ labDay, in\ Time\ labTime, in\ Lecture\ lecture, in\ Lecturer\ lecturer,$ $out\ Laboratory\ newLab$) (cf. Fig. 3) results in $newLab : Laboratory$ since lab was assigned to this output object parameter within the parameter list. This evaluation conforms to the definition above. Evaluating tgt with triple $(null, l2 : Lecturer,$ $CreateLaboratory) \in OFR$ the target activity and refining rule is equal to the case before, and is evaluated easily to $lecturer : Lecturer$ as the object $l2$ of type *Lecturer* is distinctively assigned to an input object parameter by its position in parameter list. This evaluation conforms to the definition above. Since there is no other related triple, the injective mapping condition is satisfied as well. Furthermore, for this case each two activities have at most one object of equal type flowing, thus condition (3) is satisfied apparently.

An example case for the second condition occurs in activity diagram *AddLecture* where $l : Lecture$ flows from activity *CreateLecture* to several different activities e.g. *SetRoom* and *CreateLaboratory*[6]. However, since object l is assigned to a distinct parameter function src is evaluated here always to $newLecture : Lecture$.

The semantics of refined activity diagrams can be extended to a semantics for refined activity diagrams with object flow by extending it with partial rule dependencies. Partial rule dependencies are derived from refined activity diagrams with object flow that correspond to the above definition Def. 11.

Definition 12 (semantics of refined activity diagrams with object flow). *The semantics $Sem(RA_{OF})$ of an activity diagram RA_{OF} with object flow being a refined activity diagram of $A_{OF} = (A, Obj, OFR, I, O)$ is equal to $Sem(RA)$, the semantics of the refined activity diagram RA without object flow, where in addition partial rule dependencies (see Def. 1) are defined as follows:*

For each pair of rules (p_i, p_j) in a rule sequence $s : p_1, ..., p_n$ of $Sem(RA)$ with $1 \leq i < j \leq n$, partial rule dependency d_{ij} is defined as follows: Let x (resp. y) be the activity that is refined by rule p_i (resp. p_j) in sequence s, then the partial rule dependency d_{ij} between p_i and p_j consists of all pairs $(src(x, o, y), tgt(x, o, y))$ such that $(x, o, y) \in OFR$ where src and tgt are given by Def. 11.

RA_{OF} is called dependency-compatible*, if all rule sequences in $Sem(RA_{OF})$ are dependency-compatible, as defined in Def. 2.*

Having defined refined activity diagrams with object flow formally, we consider now their consistency. Again, we define completeness as a prerequisite for consistency. We define completeness and consistency based on the completeness and consistency for refined activity diagrams without object flow.

Definition 13 (completeness of refined activity diagrams with object flow). *A set $\mathcal{S}$ of graphs is* complete *wrt. a dependency-compatible refined activity diagram RA_{OF}, if for all dependency-compatible rule sequences s in RA_{OF} there is a graph G in $\mathcal{S}$ such that s is applicable to G in the sense of Def. 2.*

Now we can finally extend the consistency definition from refined activity diagrams to refined activity diagrams with object flow.

[6] Remind that complex activity *AddLaboratory* containing activity *CreateLaboratory* was flattened in our considerations

Definition 14 (consistent activity diagram with object flow). *A refined activity diagram RA_{OF} with object flow is* consistent*, if there is a set $\mathcal{S}$ of graphs which is complete wrt. RA_{OF}.*

Please note that the properties quasi-completeness and quasi-consistency of refined activity diagrams without object flow can be extended to those with object flow accordingly.

Example 8. After having formalized the semantics of refined activity diagrams with object flow by sets of rule sequences where rules are connected by partial rule dependencies, we are ready to check the defined object flow in activity diagrams in Figure 2 according to completeness and consistency. Each simple activity is refined by interrelated object diagrams, i.e. object rules, in Figure 3.

To give the semantics of the activity diagram in Figure 2 by a set $Sem(A)$, we consider all rule sequences (with pattern *NotNull* as identical rule) and use the following acronyms: $NN = NotNull$, $CLec = CreateLecture$, $CLab = CreateLaboratory$, $CEx = CreateExercise$, and $SR = SetRoom$.
$Sem(RA_{OF}) =$
$\{(CLec, \overline{NN}, \overline{NN}, \overline{NN}), (CLec, NN, SR, \overline{NN}, \overline{NN}), (CLec, \overline{NN}, NN, CLab, \overline{NN}, \overline{NN}),$
$(CLec, \overline{NN}, NN, CLab, NN, SR, \overline{NN}), (CLec, NN, SR, NN, CLab, \overline{NN}, \overline{NN}),$
$(CLec, NN, SR, NN, CLab, NN, SR, \overline{NN}), (CLec, \overline{NN}, \overline{NN}, NN, CEx, \overline{NN}),$
$(CLec, \overline{NN}, NN, CEx, NN, SR, \overline{NN}), (CLec, NN, SR, \overline{NN}, NN, CEx, \overline{NN}),$
$(CLec, NN, SR, NN, CEx, NN, SR, \overline{NN}), (CLec, \overline{NN}, NN, CLab, \overline{NN}, NN, CEx, \overline{NN}),$
$(CLec, NN, SR, NN, CLab, \overline{NN}, NN, CEx, \overline{NN}),$
$(CLec, \overline{NN}, NN, CLab, NN, SR, NN, CEx, \overline{NN}),$
$(CLec, NN, SR, NN, CLab, NN, SR, NN, CEx, \overline{NN}),$
$(CLec, \overline{NN}, NN, CLab, \overline{NN}, NN, CEx, NN, SR),$
$(CLec, NN, SR, NN, CLab, \overline{NN}, NN, CEx, NN, SR),$
$(CLec, \overline{NN}, NN, CLab, NN, SR, NN, CEx, NN, SR),$
$(CLec, NN, SR, NN, CLab, NN, SR, NN, CEx, NN, SR)\}$

As partly shown in Example 2, the object flow in our example can be formalized by partial rule dependencies. Moreover, as defined in Definition 12, if all rule sequences in $Sem(RA_{OF})$ are dependency-compatible, RA_{OF} is as well. In the following we will show that all rule sequences are dependency-compatible by examining one after the other with an additional brief discussion.

$(CLec, \overline{NN}, \overline{NN}, \overline{NN}):$
$d_{CLec,\overline{NN}} = d_{\overline{NN},\overline{NN}} = \emptyset$
No rule dependencies occur here.

$(CLec, NN, SR, \overline{NN}, \overline{NN}):$
$d_{CLec,NN} = d_{CLec,\overline{NN}} = \emptyset,$
$d_{CLec,SR}(newLecture) = coursepart,$
$d_{NN,SR}(object) = room,$
$d_{NN,\overline{NN}} = d_{SR,\overline{NN}} = d_{\overline{NN},\overline{NN}} = \emptyset$
Since $type(newLecture) = Lecture$ is finer than $type(coursepart) = CoursePart$

(see Fig. 1), this dependency is type-compatible. Likewise $type(object) = Object$ is coarser than $type(room) = Room$ thus this dependency is type-compatible as well. It follows that this sequence is dependency-compatible.

$(CLec, \overline{NN}, NN, CLab, \overline{NN}, \overline{NN})$:
$$d_{CLec,\overline{NN}} = d_{CLec,NN} = \emptyset,$$
$$d_{CLec,CLab}(newLecture) = lecture,$$
$$d_{NN,CLab}(object) = lecturer,$$
$$d_{CLab,\overline{NN}} = d_{\overline{NN},CLab} = \emptyset,$$
$$d_{NN,\overline{NN}} = d_{\overline{NN},\overline{NN}} = \emptyset$$
Dependency $d_{NN,CLab}(object)$ is type-compatible as $type(object) = Object$ is coarser than $type(lecturer) = Lecturer$. Analogously we can argue that for $d_{CLec,CLab}(newLecture)$, types of $newLecture$ and $lecture$ are identical, thus this sequence is dependency-compatible.

$(CLec, \overline{NN}, NN_1, CLab, NN_2, SR, \overline{NN})$:[7]
$$d_{CLec,\overline{NN}} = d_{CLec,NN_1} = d_{CLec,NN_2} = d_{CLec,SR} = \emptyset,$$
$$d_{CLec,CLab}(newLecture) = lecture,$$
$$d_{CLab,NN_2} = d_{CLab,\overline{NN}} = d_{\overline{NN},CLab} = \emptyset,$$
$$d_{CLab,SR}(newLab) = coursepart,$$
$$d_{NN_1,CLab}(object) = lecturer,$$
$$d_{NN_2,SR}(object) = room,$$
$$d_{SR,\overline{NN}} = d_{\overline{NN},SR} = d_{NN_1,SR} = \emptyset$$
Dependencies $d_{CLec,CLab}(newLecture)$ and $d_{NN_1,CLab}(object)$ are type-compatible as just shown above. Dependency $d_{NN_2,SR}(object)$ was checked for type-compatibiolity before as well. Dependency $d_{CLab,SR}(newLab)$ with $type(newLab) = Laboratory$ and $type(coursepart) = CoursePart$ is type-compatible as well, as $Laboratory$ is a sub-type of $CoursePart$.

Remaining sequences can be easily evaluated the same way. Since the authors believe, that arguing is quite straightforward, in the following the arguing is left out. Furthermore only non-empty dependencies are listed.

$(CLec, NN_1, SR, NN_2, CLab, \overline{NN}, \overline{NN})$:
$$d_{NN_1,SR}(object) = room,$$
$$d_{CLec,SR}(newLecture) = coursepart,$$
$$d_{CLec,CLab}(newLecture) = lecture,$$
$$d_{NN_2,CLab}(object) = lecturer$$

$(CLec, NN_1, SR_1, NN_2, CLab, NN_3, SR_2, \overline{NN})$:
$$d_{CLec,SR_1}(newLecture) = coursepart,$$
$$d_{NN_1,SR_1}(object) = room,$$
$$d_{CLec,CLab}(newLecture) = lecture,$$

[7] Rules NN were numbered for clarity only. Numbering of $\overline{NN}$ is left out as no dependencies occur at all.

$d_{CLab,SR_2}(newLab) = coursepart,$
$d_{NN_2,SR_2}(object) = room,$
$d_{NN_2,CLab}(object) = lecturer$

$(CLec, \overline{NN}, \overline{NN}, NN, CEx, \overline{NN}):$
$d_{CLec,CEx}(newLecture) = lecture,$
$d_{NN,CEx}(object) = lecturer$

$(CLec, \overline{NN}, NN_1, CEx, NN_2, SR, \overline{NN}):$
$d_{CLec,CEx}(newLecture) = lecture,$
$d_{NN_1,CEx}(object) = lecturer,$
$d_{NN_2,SR}(object) = room,$
$d_{CEx,SR}(newExer) = coursepart$

$(CLec, NN_1, SR, \overline{NN}, NN_2, CEx, \overline{NN}):$
$d_{CLec,SR}(newLecture) = coursepart,$
$d_{NN_1,SR}(object) = room,$
$d_{NN_2,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$

$(CLec, NN_1, SR_1, NN_2, CEx, NN_3, SR_2, \overline{NN}):$
$d_{CLec,SR_1}(newLecture) = coursepart,$
$d_{NN_1,SR_1}(object) = room,$
$d_{NN_2,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$
$d_{NN_3,SR_2}(object) = room,$
$d_{CEx,SR_2}(newExer) = coursepart$

$(CLec, \overline{NN}, NN_1, CLab, \overline{NN}, NN_2, CEx, \overline{NN}):$
$d_{NN_1,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$
$d_{NN_2,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$

$(CLec, NN_1, SR, NN_2, CLab, \overline{NN}, NN_3, CEx, \overline{NN}):$
$d_{CLec,SR}(newLecture) = coursepart,$
$d_{NN_1,SR_1}(object) = room,$
$d_{NN_2,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$

$d_{NN_3,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$

$(CLec, \overline{NN}, NN_1, CLab, NN_2, SR, NN_3, CEx, \overline{NN}) :$
$d_{NN_1,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$
$d_{NN_2,SR}(object) = room,$
$d_{CLab,SR}(newLab) = coursepart,$
$d_{NN_3,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$

$(CLec, NN_1, SR_1, NN_2, CLab, NN_3, SR_2, NN_4, CEx, \overline{NN}) :$
$d_{NN_1,SR_1}(object) = room,$
$d_{CLec,SR_1}(newLecture) = coursepart,$
$d_{NN_2,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$
$d_{NN_3,SR_2}(object) = room,$
$d_{CLab,SR_2}(newLab) = coursepart,$
$d_{NN_4,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$

$(CLec, \overline{NN}, NN_1, CLab, \overline{NN}, NN_2, CEx, NN_3, SR) :$
$d_{NN_1,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$
$d_{NN_2,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$
$d_{NN_3,SR_3}(object) = room,$
$d_{CEx,SR}(newExer) = coursepart$

$(CLec, NN_1, SR_1, NN_2, CLab, \overline{NN}, NN_3, CEx, NN_4, SR_2) :$
$d_{NN_1,SR_1}(object) = room,$
$d_{CLec,SR_1}(newLecture) = coursepart,$
$d_{NN_2,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$
$d_{NN_3,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$
$d_{NN_4,SR_2}(object) = room,$
$d_{CEx,SR_2}(newExer) = coursepart$

$(CLec, \overline{NN}, NN_1, CLab, NN_2, SR_1, NN_3, CEx, NN_4, SR_2) :$

$d_{NN_1,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$
$d_{NN_2,SR_1}(object) = room,$
$d_{CLab,SR_1}(newLab) = coursepart,$
$d_{NN_3,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$
$d_{NN_4,SR_2}(object) = room,$
$d_{CEx,SR_2}(newExer) = coursepart$

$(CLec, NN_1, SR_1, NN_2, CLab, NN_3, SR_2, NN_4, CEx, NN_5, SR_3)$:
$d_{NN_1,SR_1}(object) = room,$
$d_{CLec,SR_1}(newLecture) = coursepart,$
$d_{NN_2,CLab}(object) = lecturer,$
$d_{CLec,CLab}(newLecture) = lecture$
$d_{NN_3,SR_2}(object) = room,$
$d_{CLab,SR_2}(newLab) = coursepart,$
$d_{NN_4,CEx}(object) = lecturer,$
$d_{CLec,CEx}(newLecture) = lecture$
$d_{NN_5,SR_3}(object) = room,$
$d_{CEx,SR_3}(newExer) = coursepart$

Please note that the flattened version of activity diagram *AddLecture* in Fig. 2 is consistent according to Def. 14 as all listed rule sequences are applicable to a graph G consisting of one *Lecturer* object and one *Room* object. For example, the minimum sequence shown at first $(CLec, \overline{NN}, \overline{NN}, \overline{NN})$ is applicable, if *CreateLecture* is invoked with parameter value l_1 equal to the *Lecturer* object in G, and parameter values for r_1,l_2 and l_3 different from the *Room* and *Lecturer* object in graph G. This is because rule *CreateLecture* requires only a lecturer (in this case l_1) and $\overline{NN(r_1)}$,$\overline{NN(l_2)}$ and $\overline{NN(l_3)}$ require that r_1 (resp. l_2 and l_3) do not occur in graph G. In contrast, the very long sequence listed at last is applicable as well, as rules *CreateLecture*, *CreateLaboratory* and *CreateExercise* may all use one lecturer which holds the course and one room the course takes place in. The *Lecture* object required by *CreateLaboratory* and *CreateExercise* is created always by the first rule *CreateLecture*. Please note that the set $\mathcal{S}$ consisting of graph G is minimal. However, each set which enlarges this set S by graphs with several lecturers or rooms could also be used to show that the flattened version of activity diagram *AddLecture* is consistent.

5 Related work

This paper is rooted in the research directions of formal semantics and analysis of activity diagrams as well as graph transformation approaches. While a lot of research has been done on semantics and validation of activity diagrams (see e.g. [5–7]), few works exist on the analysis of object flow in activity diagrams such as [15] and [16]. However, these approaches do not consider refinements of activities as we do. For example, [16] adds data flow semantics to activity diagrams by means of colored petri nets. Activities are not refined as in our approach, but objects which are passed between activities have attribute value checks and method calls. Colored Petri nets provide validation like reachability of certain states and quantitative analyses as matching of time bounds. In contrast, we define a semantics for activity diagrams with object flow where activities may be refined by interrelated object diagrams which has not been done before (to the best of our knowledge).

Fujaba [17], VMTS [18], and GReAT [19] are graph transformation tools for specifying and applying object rules along a control flow specified by activity diagrams. Fujaba's story diagrams integrate activity diagrams with object rules, similarly to our approach of refined activity diagrams. Compared to our approach, object flow is not depicted separately, but represented by equal names in activities. Furthermore, rules are not separated from activities. Rules used at different places have to be specified several times. We define object rules independently of activities and can apply them more than once with different arguments. VMTS supports rule application controlled by activity diagrams, similar to Fujaba. In contrast to Fujaba, rules are not directly depicted in activities, but represented by special separate activities. Global variables are used to pass model elements from one rule execution to a later one. This object flow is not checked for consistency with internal causalities of rules. GReAT supports controlled rule application as well. For each rule, input and output parameters are defined which can be connected along the control flow to pass objects from rule to rule. This restricted form of object flow is always consistently defined with the rules. All three approaches are implemented, but do not provide a formal semantics comprising activity refinement and object flow. Moreover, AGG is the only graph transformation tool which supports the applicability checks on rule sequences and also critical pair analysis to detect causal dependencies.

6 Conclusion

In this paper, we have defined refined activity diagrams with object flow where each activity is refined by a set of interrelated object diagrams in addition, describing the pre- and post-conditions of an activity. Pre-conditions can also include non-existence conditions on object patterns. We have formalized the semantics of well-structured refined activity diagrams with coherent object flow using algebraic graph transformation where activity-refining object diagrams are defined by transformation rules. In addition, we have introduced the notion of partial dependencies between rules formalizing object flow between refined activities. To prepare a notion of consistency we define the applicability of rule sequences with partial rule dependencies. The consistency definition is based on comparing rules (stemming from pre- and post-conditions) with partial rule dependencies (defined by object flow).

The graph transformation tool environment AGG can be used to analyze potential causal dependencies between rules. As a next step, facilities for partial rule dependency specification should be added to provide the basis for a consistency analysis of partial rule dependencies with causal dependencies. Our theory needs to be consolidated to enable automatic generation of graph transformation semantics from refined activity diagrams with object flow.

In future, we want to use the formal semantics given by graph transformation to prove the consistency of refined activity diagrams with object flow along sufficient criteria easy to check. We expect that the graph transformation environment AGG can do a good job to support automatic checks.

In this paper we have applied the approach to service modeling. Our example demonstrates how service behavior can be modeled precisely and how the coherence of its object flow can be checked. We expect that modeling of service composition and service orchestration and domains such as work flow design and aspect-oriented modeling can benefit from the application of our concepts as well.

References

1. Lambers, L., Jurack, S., Mehner, K., Taentzer, G.: Sufficient Criteria for Consistent Behavior Modeling with Refined Activity Diagrams. In: Proc. 11th Int. Conf. on Model Driven Engineering Languages and System MoDELS08. Volume 5301 of LNCS., Toulouse, France, Springer (2008) 341–355
2. Hausmann, J., Heckel, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In: Proc. of Int. Conference on Software Engineering 2002, Orlando, USA (2002)
3. Mehner, K., Monga, M., Taentzer, G.: Interaction Analysis in Aspect-Oriented Models. In: International Conference on Requirements Engineering RE 06. (2006)
4. Lambers, L., Mariani, L., Ehrig, H., Pezzè, M.: A formal framework for developing adaptable service-based applications. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE). (2008)
5. Eshuis, R., Wieringa, R.: Tool support for verifying uml activity diagrams. IEEE Trans. on Software Eng. **7**(30) (2004)
6. Störrle, H., Hausmann, J.H.: Towards a Formal Semantics of UML 2.0 Activities. In: Software Engineering 2005. Volume LNI P-64., Gesellschaft f. Informatik (2005) 117–128
7. Engels, G., Soltenborn, C., Wehrheim, H.: Analysis of UML Activities Using Dynamic Meta Modeling. In Bosangue, M.M., Johnsen, E.B., eds.: FMOODS 2007. Volume 4468., Springer, Berlin/Heidelberg (2007) 76–90
8. UML: Unified Modeling Language. `http://www.uml.org` (2008)
9. Ehrig, H.and Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer-Verlag (2006)
10. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In Wermelinger, M., Margaria-Steffens, T., eds.: Proc. Fundamental Aspects of Software Engineering 2004. Volume 2984 of Lecture Notes in Computer Science., Springer-Verlag (2004)
11. Plump, D.: Evaluation of Funtional Expressions by Hypergraph Rewriting. PhD thesis, Universität Bremen, Fachbereich Mathematik und Informatik (1993)
12. AGG: AGG Homepage. (`http://tfs.cs.tu-berlin.de/agg`)
13. Lambers, L., Prange, U.: Efficient Conflict Detection for Graph Transformation with Inheritance using Abstract Critical Pairs. (submitted for FASE 2009)
14. Lambers, L., Ehrig, H., Taentzer, G.: Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences. In Ermel, C., Heckel, R., de Lara, J., eds.: Proc. International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT'08). Volume 10., Electronic Communications of the EASST (2008)

15. Barros, J.P., Gomes, L.: Actions as Activities and Activities as Petri nets. In: Workshop on Critical Systems Development with UML. (2003) 20–24 workshop at 6. Int. Conf. on the Unified Modeling Language (UML 2003), San Francisco, U.S.A.
16. Störrle, H.: Semantics and Verification of Data Flow in UML 2.0 Activities. Electronic Notes in Theoretical Computer Science **117** (2003)
17. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels, G., Rozenberg, G., eds.: Proc. of the 6^{th} Int. Workshop on Theory and Application of Graph Transformation. LNCS 1764, Springer (1998) 296–309
18. Visual Modeling and Transformation System: (2008) `http://vmts.aut.bme.hu/`.
19. GReAT - Graph Rewriting and Transformation: (2008) `http://www.isis.vanderbilt.edu/tools/GReAT`.