A Formal, Graph-Based Semantics for UML Activities

Diploma Thesis

Markus Hornkamp

01.09.2009

Supervisor:	Prof. Dr. Gregor Engels
Assessor:	Prof. Dr. Heike Wehrheim
Co-Supervisor:	Christian Soltenborn

Universität Paderborn
Institut für Informatik
Fakultät für Elektrotechnik, Informatik und Mathematik
Fachgruppe Datenbank- und Informationssysteme

The science of today is the technology of tomorrow.

Edward Teller (1908 - 2003)

Contents

1 Introduction

In the development process of large software systems models play an important role, comparable to blueprints in the architecture of buildings. First, they can be used in discussions with the end-users of a software that is going to be developed to check, if the business functionality is correct according to their wishes.
Second, the architects of such a software can check quality aspects of the architecture itself using models, before the actual implementation starts, which is beneficial, because the later design changes are made, the more expensive it gets.
Models can also help by letting people work at higher levels of abstraction, or focus on certain detailed aspects. Whatever may be required by developers to discuss.

The Unified Modeling Language (UML), developed by the OMG, is the de–facto standard for modeling software systems. The definition of this visual language can be found in the UML superstructure (Obj09). Here the concrete and the abstract syntax are defined together with their semantics. While the syntax is described through metamodels, in the form of UML class diagrams, the semantics is given informally as a textual description in natural language. This informal description of the semantics comes with some problems.

It is almost impossible to phrase such descriptions in a way that no diversity in interpretation is possible. Therefore the application of UML comes with inaccuracies and ambiguities that lead to various problems. First, a variety of interpretations are possible in the modeling process, to an extent where even the understanding of experts may vary. In the field of automatic procession, i.e. execution and analysis of such models, an even more severe problem arises: any form of automatic handling is impossible without a formal semantics that a machine can interpret. Moreover problems arise in the field of UML tool development. Often the UML tools of different manufactures are incompatible with its interpretation of the semantics, so that the interchangeability of models among different tools is seriously hampered.
The UML distinguishes two types of diagrams: Structure diagrams in comparison to behavior diagrams have less complex semantics. Therefore most uml tools only support automatic code generation for structure diagrams. This greatly hinders the model driven development which strongly needs behavior diagrams, too. Without these diagrams it is impossible to describe actual behavior of parts within a software.
Given these facts, the need for a formal semantics description arises. One way to achieve this is through Dynamic Meta Modeling (DMM) (EHHS00).

Dynamic Meta Modeling in short is a technique to formally describe dynamic semantics of modeling languages that use a metamodel to describe their syntax. The key idea behind DMM is the combination of two mechanisms.
First, a DMM specification includes a so called runtime metamodel. The syntax metamodel, which describes the structure of the visual modeling language, will be mapped to this. The runtime metamodel extends the syntax metamodel with elements that are needed to describe the semantics of the model. In the domain of UML2 activities it is for instance necessary to add a

class for the tokens that flow through the activities.
Second, a set of graph transformation rules are created that describe how instances of the runtime metamodel change over time. By doing this, the semantics of the different elements of the language are defined. These rules are called DMM ruleset and are typed over the runtime metamodel, so that they can be applied to instances of this metamodel.
We only give a brief introduction of DMM at this point, so that the reader can understand the integration of this thesis in prior works; a detailed description is given in chapter 3.

1.1 Current State

The idea for DMM was first presented in a paper (EHHS00). These ideas were then elaborated in a dissertation by Jan Hendrick Hausmann (Hau05). In this dissertation, a case study is presented where a formal semantics for UML2 activities using this technique is created. The purpose of this case study is the demonstration of the convenient application of DMM, so it was never intended to cover the complete UML2 semantics for activities. For automatic application the graph transformation rules of this DMM specification have been manually translated into GROOVE rules. GROOVE (Ren09) is a toolset for graph transformation and modelchecking developed at the University of Twente.
Since then tool support for DMM has been developed. In a bachelor thesis (Röh08) a graphical editor for modeling DMM rulesets has been developed with the Graphical Modeling Framework (GMF) (Ecl08) technology. This so called DMM editor provides a great means to create a ruleset conveniently.
Moreover a recent enhancement has been made to DMM itself. In another bachelor thesis (Bau08), the ability to compare and manipulate attributes existing in the runtime metamodel have been added to DMM. This makes DMM much more viable in complex scenarios.
After its release, the DMM editor has been enhanced with a solution to transform the DMM rulesets created with it automatically into a GROOVE graph presentation. Through this addition the editor provides the means to efficiently build and apply DMM rulesets. This is mandatory for automatic application of graph transformations specified in a DMM ruleset.

1.2 Goals for this Diploma Thesis

The goals for this thesis come up from the current state. We want to continue the work of Hausmann on the formal activity semantics case study (Hau05). As mentioned earlier it was never his intention to cover all parts found in the UML2 specification about activities. And one rather unpleasant thing is that the existing rules were manually translated to GROOVE. While this was necessary to apply this ruleset to actual models, it is very inflexible and a source of error.
Fortunately the manual translation step is now obsolete due to the automatic translation the DMM editor provides. With the tools available, the goal is to make a new DMM ruleset using the GMF editor that allows flexible practical application. As the semantics Hausmann presents in (Hau05) does not completely cover all details found in the UML2 specification (Obj09). Especially with the possibilities the attribute enhancement to the DMM system provides, we want to cover all details found in the UML specification as far as possible.

To achieve this, the UML specification has to be analyzed thoroughly and with the insight gained through these studies, the next step consists of two parts: The first counterpart is to build an appropriate runtime metamodel for the ruleset to be typed over. The second counterpart is creating the ruleset itself that describe the semantics of instances of the metamodel exactly.
After creating such a ruleset the final step is to evaluate it to guarantee the quality of it. Therefore appropriate test cases have to be found to validate the specification. Additionally an automatic test framework to speed up the testing process should be implemented.

1.3 Document Structure

The structure of this thesis is as follows: chapter 2 starts with a brief introduction of UML2 activity diagrams, followed by a description of the syntax and informal semantics of single activity elements. This chapter closes with an explanation of the token flow concept in activities that build the core of the semantics.
In chapter 3, Dynamic Meta Modeling is described in detail. In the first section, the basics are introduced, after that the detailed mechanics are explained with an example.
How this technique can finally be used to define a formal semantics for UML2 activities is shown in chapter 4. This is the main chapter of this thesis, starting with an explanation of the case study in the previous work by Jan Hendrik Hausmann (Hau05). In the following section, the runtime metamodel used for the DMM ruleset is explained in detail followed by the description of how to map this to the actual syntax metamodel. After this introduction, the actual DMM ruleset is presented explaining the semantics of UML2 activities.
Thereafter chapter 5 presents a test framework enabling a user of the created ruleset to automatically test the application with activity diagrams. In the final section of this chapter some interesting test cases are presented.
Finally in the last chapter a summary of the work at hand and an outlook are given.

2 Overview of UML2 Activities

This chapter gives an introduction to UML2 activities. Their purpose in the UML2 is pointed out and the informal semantics are described. In section 2.2, the notations and their semantics are described. This is only meant as a brief overview of the matter at hand, directed at people not familiar with activities. The semantics are described in more detail in chapter 4 and in the appendix, where the formal definition via a DMM ruleset is specified. In the last section 2.3 the token concept is illustrated to give a further understanding of the activity semantics.

2.1 Introduction to UML2 Activities

Developed by the OMG, the Unified Modeling Language (UML) is the de facto standard for modeling software systems. One part of this language are activities, nowadays they are used for behavior modeling: from the specification of use cases, over workflow modeling, down to the implementation of operations (Boc03). The visual representation of activities are activity diagrams, which are described in the following sections. Activities have been under quite a development in the past years. In the UML 1.1 they were integrated rather rash for being a popular flow modeling means. While this standing alone really was not related to the object oriented paradigm which is the main scope of the UML, it led to activities being rather poorly integrated with the rest of the UML. In this early version their semantics was based on state machines. The release of the UML2 brought a complete change to activities. Here they were well integrated with the rest of the UML and the semantics was completely redesigned to be "petri-like" (Obj09, p. 324). As mentioned earlier this semantics is only given informally, which will be described in this section. In chapter 4 our approach for a formal semantics will be shown.

2.2 Description of Activity Diagram Elements

This section describes the main elements of UML2 activity diagrams. It is meant as a quick reference, directed to the readers unfamiliar with the theme at hand. An activity diagram is a visual representation of an activity. It is build of a rectangle with rounded corners including the activities' name. The rectangle contains a network consisting of nodes and directed edges. The UML2 makes a distinction between three types of activity nodes: actions as fundamental units of an activity, object nodes as helpers to describe the flow of objects, and control nodes to control the flow of tokens in an activity.
In the UML2 specification the activities are organized into seven packages which are described in the following:

FundamentalActivities This package states that an activity consists of a collection of activity nodes. It also integrates activities into the rest of the UML by stating of which UML elements the contained elements are derived from.

BasicActivities The BasicActivities package adds control sequencing and data flow between actions.

IntermediateActivities Within this package support for concurrency (using fork node and join node) and decisions (through decision node and merge node) is introduced.

CompleteActivities Enhances activities with lower level constructs like edge weights and streaming.

StructuredActivities Adds traditional programming constructs such as sequences, loops, and conditions.

CompleteStructuredActivities Enhances the constructs of the StructuredActivities package with data flow capabilities.

ExtraStructuredActivities Adds higher level constructs such as exception handling and invocation of behavior on sets of values.

In the following we will not use this structure, but use an outline that matches the structure of the DMM specification created in this thesis. This structure is more semantics orientated and groups elements with similar or related semantics together.

2.2.1 Activity Edges

There are two main kinds of edges: Control flow edges and object flow edges. A control flow edge is an edge that starts an activity node after the completion of the predecessor node by passing a control token. An object flow edge is for modeling the flow of actual objects or values flowing between object nodes. Figure 2.1 shows that the visual notation of these two types of edges is always an arrow.

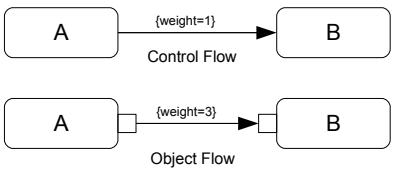

Figure 2.1: Control and object flow

The CompleteActivities package adds the possibility do define how many tokens must traverse an activity edge at the same time. If the denoted number of tokens is reached, all the tokens at the source are offered to the target at once (Obj09, p. 326). If a weight is given for an activity edge, usually the visual notation of this is the actual weight beginning with "weight=" standing with curly brackets as a label at the corresponding edge. In the UML2 specification (Obj09, p. 327) it is stated that this weight is a value specification. "The weight is a value specification, which may be a constant, that evaluates to a non-zero unlimited natural value." [1]

[1] In our DMM ruleset we will support weight with a constant value specification (LiteralInteger). To process non-constant cases it would be necessary to parse the semantics of such definitions and interpret it with an algorithm

2.2.2 Control Nodes

Control nodes were redesigned in the UML2. It is worth noting, that these nodes can not hold tokens if they are not accepted further downstream (Obj09, p. 315). They are used for the routing of tokens. This is achieved by introducing decisions, enabling concurrent flows using forks and joins and enabling the production and consumption of tokens. An overview of all control nodes can be seen in figure 2.2, where all notations are shown. In the following sections they will be described in detail.

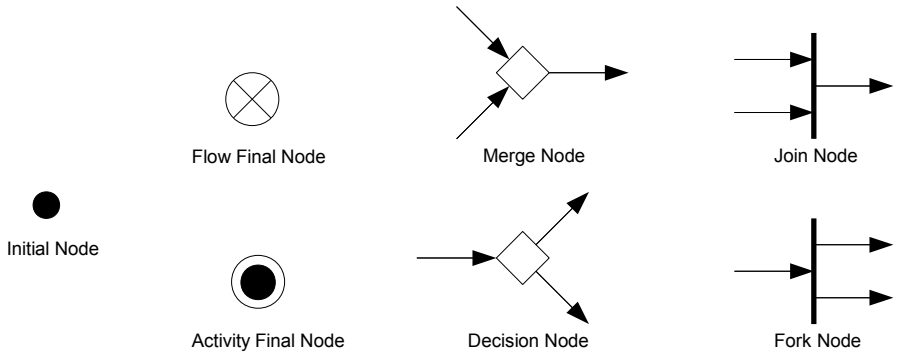

Figure 2.2: Visual representation of control nodes

2.2.2.1 Initial Node

An initial node, like the name implies, is the point where the execution of an activity starts. The visual notation as seen in figure 2.2 is a simple solid circle. When the activity starts, a control token is placed here and offered to all the outgoing edges. An activity can have multiple initial nodes; these are all handled in the same way. Thus upon starting an activity with multiple initial nodes, multiple flows will be started.
Initial nodes are an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream (Obj09, p. 387). The characteristics and the formal semantics of a initial node can be found in section 4.4.1.

2.2.2.2 Decision and Merge Node

The purpose of decision nodes is to guide the flow of tokens into different directions in an XOR like behavior. The visual notation is a diamond with one incoming and multiple outgoing flows, as seen in figure 2.2. To determine which path the incoming token should take, the application of guards are advised. Even if no guards are specified or more than one guard evaluates to true, only one path is taken. Every time a token is offered to more than one outgoing activity edge, there is always only one path taken due to token competition. This concept and the reason behind it is explained in section 2.3.
The notation of merge nodes is the same as the notation of decision nodes except that there are multiple incoming flows and only one outgoing flow. The semantics is to let every incoming token pass along the outgoing flow. This construct does not necessarily imply an predecessing

decision node and can be used independently. The rules for decision and merge nodes are explained in section 4.4.5.

2.2.2.3 Fork and Join Node

As shown in figure 2.2, fork nodes are notated as a bar with one incoming and multiple outgoing flows. Their semantics is to pass copies of incoming tokens to all outgoing flows. This construct is used to indicate that these outgoing flows can be executed concurrently.
Join nodes are the complement to fork nodes. Their visual notation is the same except they have multiple incoming and only one outgoing flow. They are used to synchronize concurrent flows that originated from fork nodes. This is done by taking the information found in incoming token copies that emerged from a Fork Node and putting this information into a new token[1]. What rules have to be considered and how exactly the fork and join process works can be found in section 4.4.2.

2.2.2.4 Final Nodes

Final nodes consume incoming tokens and therefore are means to stop the execution of an activity. As seen in figure 2.2, there are two types of final nodes, the flow final node and the activity final node. Flow final nodes consume every incoming token, by doing so they end a single flow of execution. Other tokens in the same activity remain untouched.
Activity final nodes do not just consume tokens, but terminate the enclosing activity. As soon as one token arrives on an activity final node, all tokens currently being in the activity are destroyed, including all invoked behavior.

2.2.3 Object Nodes

Object nodes exists to represent data or objects in activites. They come in three types: directly attached to an action as pins, they represent input and output data of the corresponding action. Second they can serve as activity parameter nodes for the input and output values of activities.

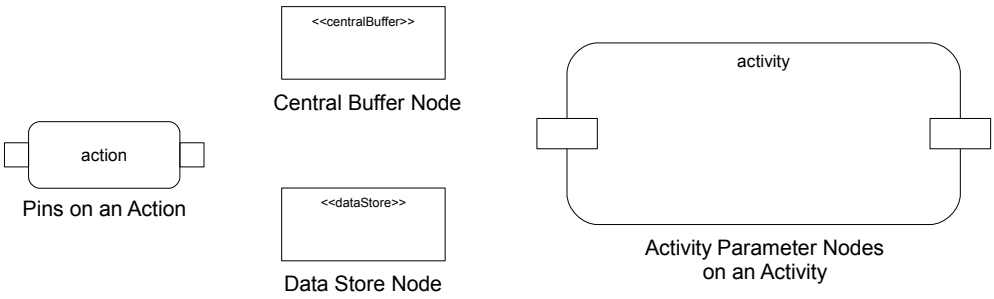

Figure 2.3: Visual representation of object nodes

[1] Object Tokens and Control Tokens, especially when they arrive simultaneously, are treated differently; this is pointed out in detail in section 4.4.2

Finally they can be unattached to other nodes as centralbuffer or datastore node.
All object nodes can buffer more than one object token. The limit can be exactly specified by adding an upperBound ValueSpecification containing the limitation (Obj09, p. 393). Object nodes are usually typed and moreover they restrict the types of the objects they can store as only the same or subtypes of the object node's type itself.

2.2.3.1 Pins

There are two types of pins: Input pins and output pins. Input pins represent the input parameters of an action. An input pin holds the input values that are received from other actions. Output pins are the complement to the output parameters of an action. They hold the output values produced by the action.
Notable are the upper and lower attributes inherited from MultiplicityElement. They control the execution of an action. An action can only start if an input pin has at least as many tokens as the lower multiplicity indicates. The upper value of an input pin dictates the maximum number of tokens that can be consumed by one execution of the action (Obj09, p. 255). For output pins these rules are adapted: if it has fewer tokens than the lower multiplicity the action execution can not terminate. The upper multiplicity indicates the maximum number of tokens a single action execution can put into the output pin (Obj09, p. 263).

2.2.3.2 Activity Parameter Node

Activity parameter nodes provide a means to accept inputs to an activity and provide outputs from an activity, through the activity parameters. Activity parameters inherit support for streaming from Parameter. Streaming activity parameters attached to an activity parameter node allow an action execution to take inputs and provide outputs while it is executing. During one execution, the activity may consume multiple tokens on each streaming input and produce multiple tokens on each streaming output[1]. The concept of streaming is further explained later when the solution for the execution of actions is explained in section 4.4.3.

2.2.3.3 Centralbuffer and Datastore

Centralbuffer nodes are object nodes, not directly connected to other nodes, with the stereotype «centralbuffer», as seen in figure 2.3. They serve as a simple buffer for multiple object flows. A datastore node is used to model data flow in which the data is persistent. They have the stereotype «datastore». Unlike the centralbuffer node, all tokens are copied before leaving so that they are kept until the enclosing activity is terminated.

2.2.4 Action Nodes

An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. (Obj09, p. 217). An action is executed if all input conditions are satisfied; this means that all incoming

[1] Although directly involved in the streaming process a pin itself does not inherit any support for streaming

edges carry at least one control token, and all input pins contain enough tokens to meet the lower multiplicity as stated before. After the execution has finished, a control token will be produced on each outgoing edge, and on the output pins tokens will be produced as specified. In the DMM ruleset specified for this thesis the focus was set on two different types of actions: opaque actions and call behavior actions. Opaque actions have no behavior attached that is invoked and can be seen as a placeholder for an implementation, whereas call behavior actions have an attached behavior that is being invoked on execution. An attached behavior can be once again an activity allowing a nested structuring of activities. Details to the execution of actions can be found in section 4.4.3.

2.3 Token Concept

The UML2 specification describes the semantics of activity diagrams to be "petri-like" (Obj09, p. 324). To put it in a nutshell, tokens and their corresponding offers determine all states of an activity execution. There are two types of tokens, control tokens that simply represent a locus of control, and object tokens, that contain objects or simple data. These two types are sometimes handled differently. While both object and control tokens can be held by object nodes, only control tokens can be held directly by actions and initial nodes. Tokens are always offered to all outgoing edges of the corresponding node. Here token competition comes into play. Although there may exist multiple outgoing edges, there will always be only one path a token will actually take, even though other paths could also have been traversed.
Tokens that have been offered stay put until they are accepted by target nodes. To estimate which path is actually taken, the offers on all outgoing edges traverse through the activity graph until one of them is finally accepted. Upon being accepted by their target nodes, tokens traverse the whole path from the source node to the target node at once. This behavior is called the traverse-to-completion principle (Boc04).
The main reason for the existence of this behavior is to avoid deadlocks to some extend. Figure

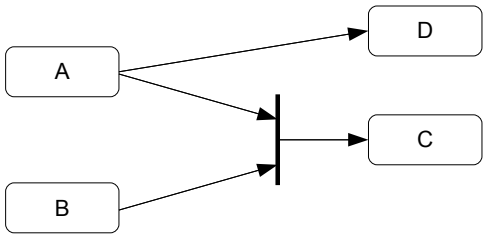

Figure 2.4: Example where traverse to completion prevents deadlocks

2.4 shows an example of traverse to completion preventing a token to get stuck on a control node. A token provided by action B will not flow towards the join node unless action A also provides a token. If the traverse to completion principle would not apply here, it is possible that a token from action B flows to the join and action A does not provide a token and vice versa. In these cases the token would get stuck at the join node.

2.4 An Example Activity

In this section, an example activity is presented that contains parts, where the semantics is not easily understandable or has ambiguous explanations in the specification. It will be used later in chapter 4.4 to explain interesting parts of the DMM ruleset created for this thesis. The example activity seen in figure 2.5 is a snippet activity from a web server providing streaming videos for payment.

2.4.1 Structure of Example

The initial node of the activity leads directly to a fork node, with two outgoing control flows. This fork node serves for the purpose of arranging the "show advertising" action parallel to the rest of the activity. Therefore "show advertising" can be executed at any time beside the other actions of the activity, so that the advertising can be showed all the time to the users. Another point where the execution starts is the action "user buys viewing time". If an user of this web application buys viewing time, this action is executed. Object tokens containing the data of one minute viewing time are stored in the output pin after action execution. The viewing process requires at least 5 minutes of left viewing time for each user's account to start the viewing process, so the outgoing object flow has a weight of 5. Therefore it is required that 5 object tokens traverse this edge at the same time, if less are available then they stay put in the output pin. The object flow leads to a join node, which has one more incoming edge. It is a control flow with an weight of 1 emerging from the fork node mentioned earlier. The action after the join is

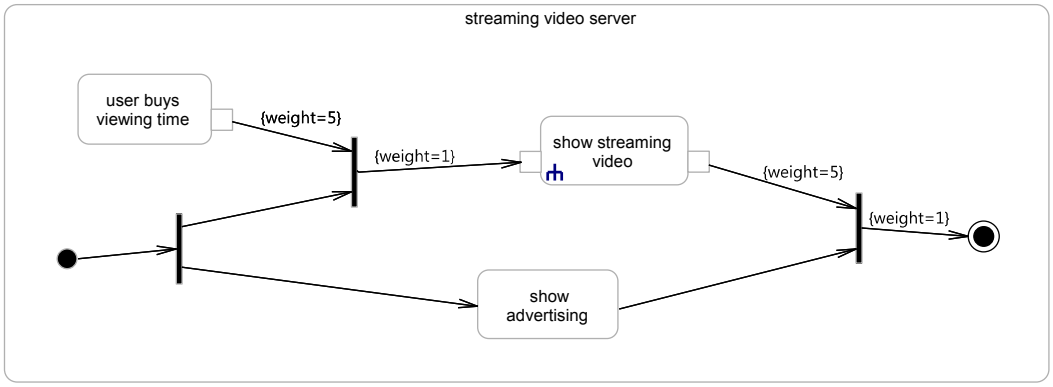

Figure 2.5: Example activity part 1

a call behavior action named "show streaming video". And as stated before the viewing process requires at least 5 minutes of viewing time so the input and output pins of this action have their upper and lower attribute set to 5, meaning that at least 5 tokens have to be consumed and also be created by the execution. Notable is that the parameters of the called behavior are streaming parameters, this means that this action can consume and produce tokens while the execution is running. On execution of this action an behavior is executed that is again an activity named like the action itself. This activity named "show streaming video" can be seen in 2.6.

It has two activity parameter nodes that match the two pins of the calling action. When the data of a token with 1 minute of viewing time gets passed to the activity parameter node this data

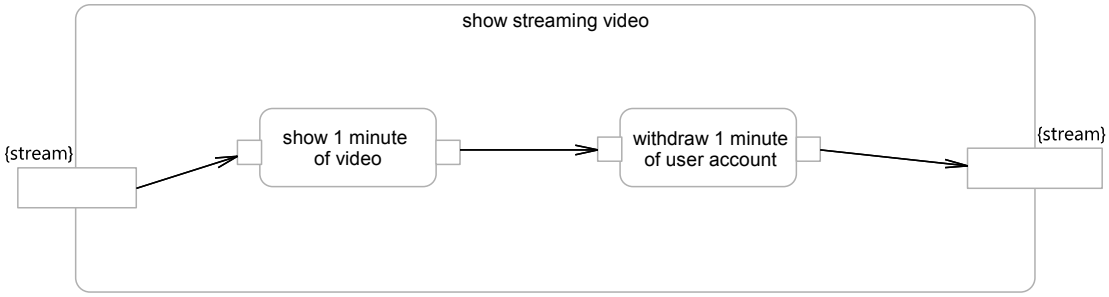

Figure 2.6: Example activity part 2

is available for this new activity. A new object token for this data is created and being passed through this activity. At first it arrives at the action "show 1 minute of video" that is executed and passes the data to another action "withdraw 1 minute of user account". After this action is executed the data is passed with an object token again to the following activity parameter node. Because the associated parameter is streaming the invoking action produces a token for this data and saves is in the corresponding output pin. This token is offered to the outgoing edge, that has a weight of 5 again. So all tokens produced by the action remain in the output pin until there are at least 5. The activity edge leads to another join node where all parallel parts are synchronized leading to an activity final node.

2.4.2 Challenges

The experience gained, from creating the DMM ruleset for UML2 activities, shows that there are some parts in the example above that become challenges for a creator of formal semantics. These challenges are explained in the following:

I. Joining of different types and number of tokens At both join nodes we have a non trivial case of joining flows: we have a mixture of numbers and types of tokens arriving at the same time. While the joining of different types of tokens is described in detail by the specification, the handling of different weights arriving at the same time is only mentioned indirectly. In section 4.4.2 the approach we have taken is described in detail.

II. Implicit FIFO queue at fork node At fork nodes tokens are offered to all outgoing edges and if one of this offers get accepted, duplicates of this token are made and offered to all other outgoing edges. If they can not be accepted are kept in an "implicit FIFO queue" (Obj09, p. 376). Such an FIFO queue is introduced as TokenQueue in the runtime metamodel to accomplish the correct description of this semantics. Another difficulty in describing the semantics of fork nodes is that a token may pass multiple fork nodes before it finally arrives at a join node. This requires a recursive approach in rule definition, because the visited fork nodes have to be remembered to correctly duplicate the tokens. A good way to achieve this was presented in the case study in (Hau05). We further developed this idea for the combination of different types of tokens. The whole thing about the fork and join semantics is presented in section 4.4.2

III. Action execution with streaming parameters Given the option of streaming parameters, defining formal semantics for the execution of actions is non trivial. A lot of possi-

bilities have to be taken into account to say, when exactly an action execution starts, and when this execution has to finish. These possibilities and the approach chosen for this thesis are presented in section 4.4.3.

IV. No direct connection between pins and activity parameter nodes Another hurdle if one is defining formal semantics of call behavior actions or call actions in general is the connection between the pins of the call action and the parameters of the invoked behavior. This is needed to pass the tokens from the pins to parameters and the other way round. In the specification it is explained that this is done by matching type, ordering and multiplicity of the pins to the parameters (Obj09, p. 243). While this is sufficient for modelers, this proves to be a bother in the design of a DMM ruleset for this semantics, where one is forced to constantly find the correct match trough traversal of all possibilities while somehow keeping track of the position of the counterpart. To avoid this problem, a direct connection between pins and their corresponding parameters was introduced in the runtime metamodel shown in section4.2.

V. Exceptions Another interesting challenge that does not appear in the example are exception handlers. An exception handler is an element that specifies another executable node to be executed, if a specified exception occurs during the execution of an executable node marked for protection. For exception handlers to correctly work with actions most of the action rules had to be reworked, as described in section 4.4.4.

3 Dynamic Meta Modeling

This chapter explains the techniques and technology used to create the formal semantics for UML2 activities. At first, a small introduction to the Dynamic Meta Modeling technique is given. Afterwards, the key parts of DMM, the runtime metamodel and the DMM ruleset, are explained in more detail. To convey a better understanding of these two counterparts, examples are given, which are taken from the UML2 activity DMM specification created for this thesis. These are used to explain the methods of DMM, the actual utilization in form of the DMM specification for UML2 activities is presented in the following chapter 4.

3.1 Introduction to DMM

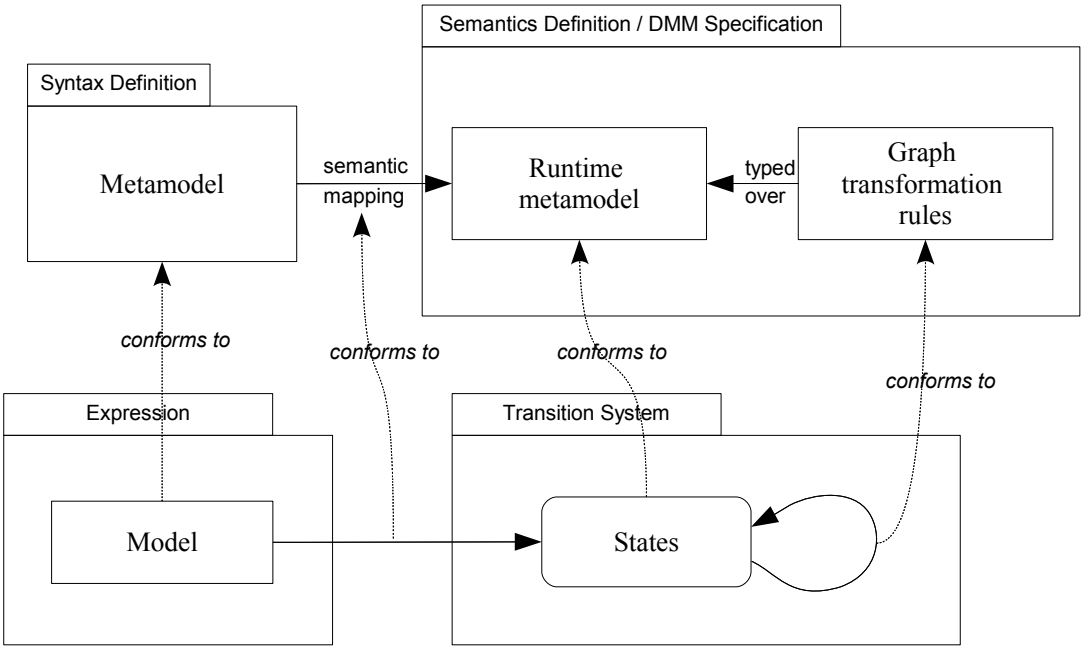

Figure 3.1: Overview of the DMM approach

As mentioned in the introduction, the semantics description is only given informally for the UML. As this prevents automatic procession and sometimes also leads to a diversity in interpretation, Dynamic Meta Modeling (DMM) (Hau05) was developed as a means to create formal semantics for visual modeling languages that have a metamodel to define the syntax[1]. This technique consists of two major counterparts to achieve this aim. Figure 3.1 indicates how these parts work together.
The first part is the runtime metamodel that is mapped to the original metamodel used to define

the syntax of the visual modeling language. Usually this runtime metamodel contains additional elements that are needed for the semantics description. We call an instance of the runtime metamodel the runtime model. In the domain of UML2 activities the runtime model is called a runtime activity.
The second part is the DMM ruleset, a set of graph transformation rules typed over the runtime metamodel. These rules specify all changes an instance of the runtime metamodel receives, by describing what element is changed if a certain condition is met and exactly in which way this happens then. In the domain of UML2 activities this is often a handling of offers and tokens flowing through an currently executing activity.
The mapping of the metamodel to the runtime metamodel makes it possible to get the corresponding runtime model from a model. On such a runtime model the ruleset can be applied resulting in a transition system. A transition system consists of a set of states and transitions between states. In this case every single rule application of the ruleset results in one transition. And all states in the transition system stand for different states of the runtime model. The start state of the transition system for instance stands for the runtime model in the initial state. The two counterparts and how they work together are described more throughly in the following sections.

3.2 Runtime Metamodel

The first part of a DMM specification is the so called runtime metamodel. As already mentioned, in comparison to the metamodel of the visual modeling language, which is going to receive the DMM specification for the semantics, it contains additionally elements that are necessary to describe the semantics.
To get a better idea, what is meant by such additional elements and how exactly a runtime metamodel looks like, figure 3.2 presents an example in the context of UML2 activities. Here

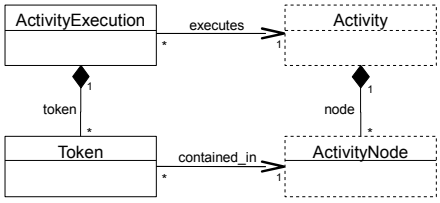

Figure 3.2: Example of a runtime metamodel

we see the class Activity that represents an activity of the original metamodel drawn with a dashed line. Elements that are added to the runtime metamodel and therefore not present in the UML2 metamodel are drawn with a solid line. The first new class is ActivityExecution with the association "executes" to Activity. This is used to indicate whether the execution of an activity is active or not. The other new class that is presented here is Token that has an containment association to Activity. The Token class represents tokens which traverse the activity on execution and is needed to model the actual token flow for the activity semantics. Token has an association to ActivityNode named contained_in to represent the node a token is kept at. These two new elements are only an example for elements that are needed to describe

[1]This technique also works for visual modeling languages other than the UML, as long as a metamodel for the syntax exists.

the activity semantics, the complete runtime metamodel is presented in section 4.2. When it comes to the actual creation of a runtime metamodel, there are two options: to create a complete new metamodel from scratch or to enhance the existing metamodel.

3.2.1 Creating a Runtime Metamodel from Scratch

Creating a new metamodel from scratch is shown with an example in figure 3.3. Here a snippet of a runtime metamodel for UML2 activities is presented and mapped to the original metamodel. Similar to the prior example it contains an ActivityExecution class with an "executes" association to Activity for the execution of an activity. The Activity class is mapped to its original counterpart. Notable here are the two classes contained in the Activity of the runtime metamodel: Node and Edge. Node is mapped to ActivityNode and Edge to ControlFlow and ObjectFlow of the original metamodel. This is a resemblance to the runtime metamodel of the

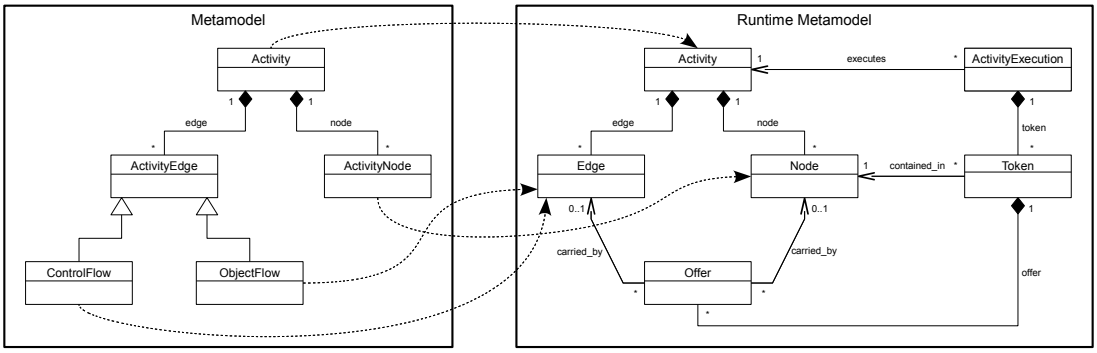

Figure 3.3: Snippet of a runtime metamodel created from Scratch mapped to the original one

case study found in (Hau05), that was created from scratch. In this runtime metamodel ControlFlow and ObjectFlow are not distinguished, because it is irrelevant for the semantic description presented in there. Besides ActivityExecution, the classes Token and Offer have no counterpart on the left side, because these are only needed for the semantics description. Offer has a "carried_by" association to both Node and Edge, whereas Token has a "contained_in" association only to Node, because tokens can not be held by edges. This example shows that there are a lot of elements found in both the runtime metamodel and the original one. Whenever the number of these elements is very big it is not advised to create the runtime metamodel from scratch, but rather by enhancing the original metamodel as shown in the next section.

3.2.2 Creating a Runtime Metamodel by Enhancing the Existing Metamodel

This section illustrates the creation of a runtime metamodel by enhancing the existing one. As the runtime metamodel has some additional elements for the semantics and apart from there are usually minor changes, using the application of the decorator pattern (GHJV95) is advised. This is done by an extension of the existing metamodel that is used to specify the syntax of the modeling language. In figure 3.4 there is the same snippet of the original UML2 metamodel as in the example before, but this time it is referenced by another metamodel containing only the additional elements needed for the semantics. Both these metamodels together form the

runtime metamodel. Crucial in this approach is that the original metamodel remains unchanged, so that only elements from the new metamodel can reference elements from the original one and not the other way round. Due to that fact that the runtime metamodel often only contains

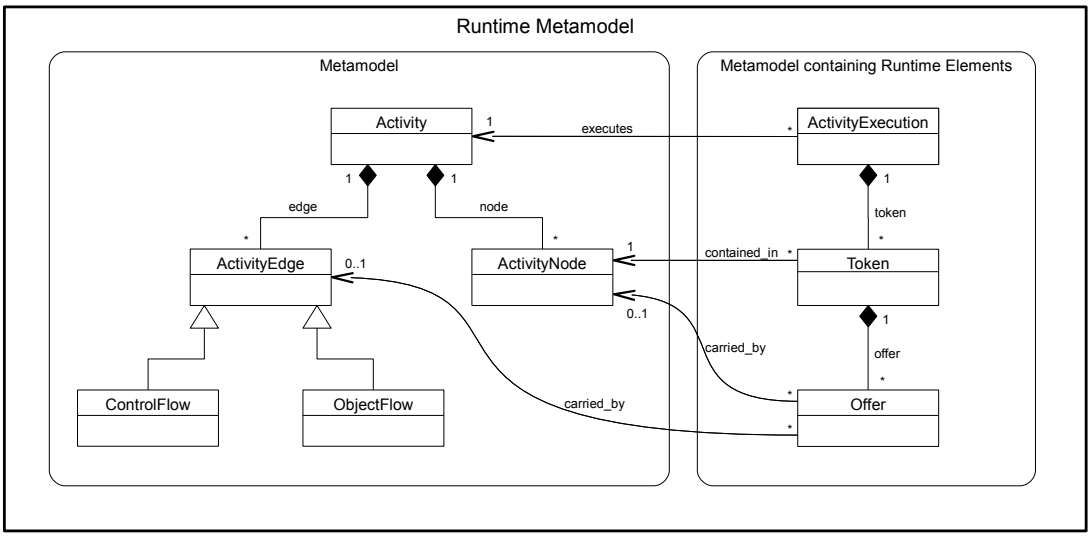

Figure 3.4: Snippet of a runtime metamodel consisting of an extension of the syntax metamodel

some additional elements needed for the execution of the model, while the rest of the models are exactly the same, this is more convenient in most cases. For this reason, we chose this method for the runtime metamodel in the UML2 activity DMM specification as shown in section 4.2. There are of course cases where only few of the original metamodel elements are used to describe the semantics. In such a case a complete new metamodel can save a lot of overhead if most classes are unused.

3.2.3 Mapping between Metamodel and Runtime Metamodel

After this runtime metamodel is created a mapping needs to define the connection between these two metamodels. Such a mapping needs to be executable in some form, so that is is possible to automatically get a runtime model from a given model. In the DMM specification created in this thesis, Java is used to specify the mapping, as explained in section 4.3. A runtime model is needed to apply the DMM ruleset and receive a transition system, which can be seen in the next section 3.3. Basically a runtime model is being viewed as a host graph on which graph transformation rules can be applied, therefore the runtime model is also referred to as host graph when speaking of graph transformation. To put it in a nutshell the application of a graph transformation rule is done by searching for an occurrence of the graph defined by the rule and by replacing this found occurrence with the changes described by the rule. On this host graph the transformation rules contained in the DMM ruleset can be applied. They describe how the model changes at runtime, creating all possible states the execution of the model can take on in a transition system. The graph transformation principles DMM uses are explained in the following section while describing the structure of the rules in the DMM ruleset.

3.3 DMM Ruleset

The second part of a DMM specification is the already mentioned DMM ruleset being a collection of DMM rules. These are basically graph transformation rules that describe how a runtime model changes at runtime. These changes define the semantics of all elements of the corresponding model that is described by the metamodel mapped to the runtime metamodel. To understand all mechanics behind a DMM specification, a brief introduction to graph transformation will be given in the following section.

3.3.1 Introduction to Graph Transformation

Graph transformation is a method to create a new graph out of an original graph, the so called host graph. There are several approaches for graph transformation. These approaches use rules to describe changes in a graph, that are being executed on application. In DMM the runtime model is used as a graph and with a set of graph transformation rules it is described, how this graph changes over time, in order to specify the semantics. In the following we will stick to the single–pushout approach (Löw93), because it is the one used in DMM. More information regarding graph transformation can be found in (Hec06; BH04; CMR+97; EHK+97; Roz97).
In the single–pushout approach each rule has a left-hand side L and a right-hand side R, which are both graphs. A rule matches the host graph if there is a morphism between L and the host graph. This means that the structure of L is searched within the host graph and found. In this case the graph transformation rule can be applied to the host graph. On rule application the changes between L and R are analyzed and applied to the host graph: elements that are only present in L are deleted from the host graph, elements only found in R are added to the host graph, whereas elements that are both in L and R remain unchanged. After such a rule application the host graph is changed as described by the rule. There is also a concept in the single–pushout approach, that allows to specify the absence of elements in L by marking them. These elements must not be in the host graph at the specified spot for the rule to match. This method is called negative application condition (NAC).
The possibility exists to define a type and attributes for every node in a host graph. This is a concept that comes from object oriented programming. Usually a so called type graph that contains all possible types, their attributes and the associations between them is used to achieve this. As the host graphs are then typed over the type graph, they are accordingly called typed attributed graphs. In order to match correctly, the graph transformation rules have to be typed over the same type graph as well.

3.3.2 Graph Transformation Rules in DMM

As mentioned earlier a runtime model can be seen as a host graph for graph transformation. To allow automatic application of the rules, they are typed over the runtime metamodel. This has a great advantage: the semantics are not only formally described with these DMM rules, they can also be applied automatically to a runtime model generated from a given model. Furthermore the rules being typed over the runtime metamodel have almost the same visual representation as object diagrams and are therefore easy to read for the casual user of object orientation concepts.

3.3.2.1 Nodes and Edges

In figure 3.5 an example rule is shown to illustrate the structure of a DMM rule. The elements a DMM rule is build upon are nodes and edges. In the application process all nodes are matched to objects and all edges are matched to links in the runtime model. For the matching process to successfully complete, the host graph objects being typed over the runtime metamodel and the rule nodes also being typed over this metamodel must have either the same type or the object must have a supertype of the node type. After the matching is done, the changes described

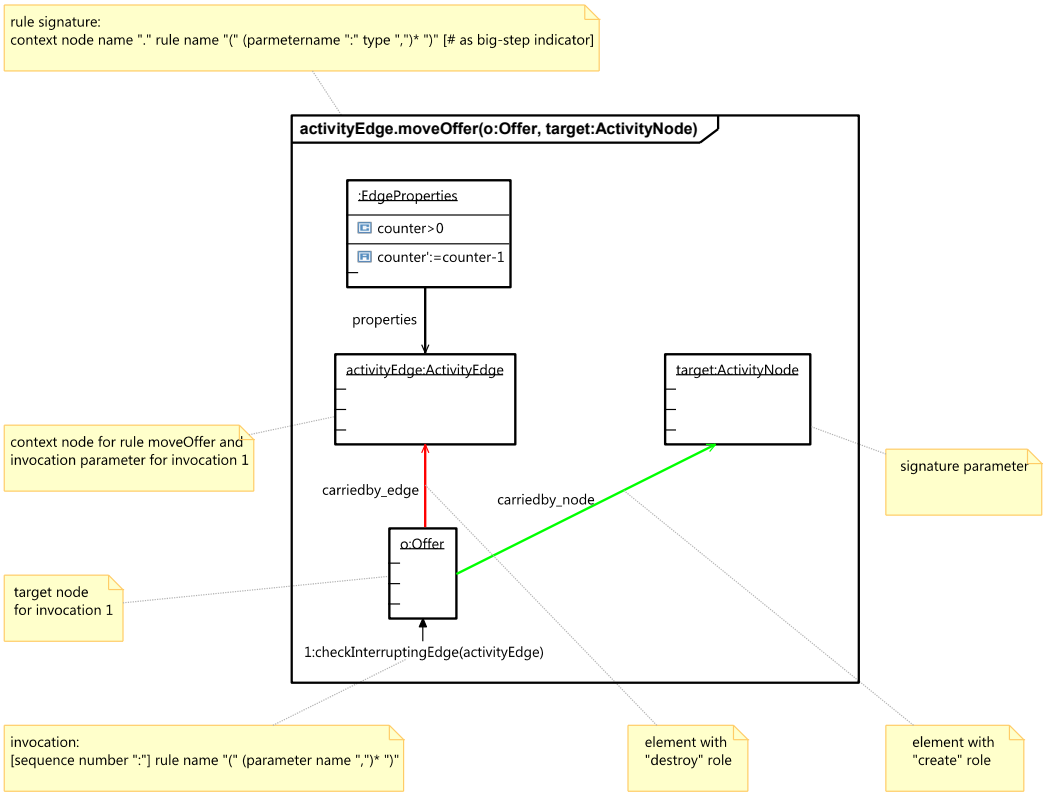

Figure 3.5: Example rule of DMM ruleset

by the rule elements are applied to the corresponding elements in the host graph. The visual representation of a node in a DMM rule is a rectangle with name and type seperated by a colon in the upper part. In the example 3.5 the rectangle with the text target:ActivityNode is a node with the name target and of the type ActivityNode. The name of a must either be unique or empty, because it is used to identify nodes in a DMM rule. The node names have no consequences for the matching process. Edges are visually shown as arrows between two nodes in a DMM rule. Their name and direction are determined by their corresponding association found in the runtime metamodel, because each edge has an association in the runtime metamodel as counterpart. Only if such an association is defined in the runtime metamodel an edge between two nodes can exist in a DMM rule. In the example 3.5 the edge with the name properties refers to an association with the same name between the two types EdgeProperties and ActivityEdge.

3.3.2.2 Context Node and Invocations

Every DMM rule contains exactly one context node that is the anchor point of the rule and determines, like the name implies, the context to which the rule is applied. This can be seen like an object in object oriented programming languages that has a certain behavior, here the behavior is specified by rules for a context node. DMM also has a special mechanism called invocations. An invocation defines which rule has to be invoked on a marked node. This marked node is the so called target node of the invocation and for the invoked rule it becomes the context node. This context node can therefore be compared to an object on which a particular method is invoked within object oriented programming languages.
Rule invocations are applied directly after the application of the rule invoking them. If more than one invocation is used in one rule, a sequence for the application of these invocations can be set with sequence numbers. This concept allows splitting of big rules into more but smaller ones, leading to better readability and reusability of rules. The possibility of passing parameters to invoked rules also exists. Parameters are nodes within the invoking rule that need to match to the parameters of the invoked rule. If the parameters and the target node of the invoking rule are not consistent with the parameters and the context node of the invoked rule, the invocation will fail. This means that the DMM ruleset is invalid and on application of the invocation, no rule could be found matching the signature, resulting in an error.
The visual representation in a rule of an invocation is done with an arrow pointing to the target node with a label including the name and names of parameters of the invoked rule. All nodes being used as parameters must have a name for identification. In figure 3.5 the context node of the actual rule is the node with the label activityNode:ActivityNode. The invocation checkInterruptingEdge(activityEdge) has the target node o:Offer, thus making o:Offer the context node of the invoked rule checkInterruptingEdge(activityEdge). The number at the beginning is the sequence number. In this example it is obsolete, because there is only one invocation in total, but it shows how the visual representation of sequence numbers is handled.

3.3.2.3 Types of Rules

In DMM there are three different types of rules: bigstep, smallstep and premise rules. The differences between these rule types will be explained in the following. Bigstep rules are applied to the host graph directly and unlike smallstep rules, they can not be invoked. On the contrary, smallstep rules can only be applied to the host graph if another rule invokes them. In a nutshell, bigstep rules behave like normal graph transformation rules, meaning that they are applied as soon as they match. There is one addition though: if they invoke one or more smallstep rules other bigstep rules can only start their application if all these invoked rules have been applied. While the smallstep and bigstep rules have no limitations considering their content, Premise rules are not allowed to change the host graph. These rules serve for the purpose of narrowing the matching of their invoking rule. Basically the content of the Premise rule is copied into the invoking rule. This all serves for the purpose of making the rules much more readable. Unlike smallstep rules, Premise rules must have a unique signature[1]. Each rule has one signature shown in the upper left corner of the example 3.5. It consists of the context node followed by a dot and afterwards the name of the rule. Following the rule name come brackets containing the parameters with name and type separated by commas. If the rule is a bigstep rule an # at the

end of the signature indicates this.

3.3.2.4 Rule Overriding

DMM also provides an option to set rules to override another rule. Whenever the type of a rule's context node is a subtype of another rule's context node, it is possible to define that this rule overwrites the other rule. This means in practice that whenever the rule that overwrites another one matches, the overwritten rule does not match and is therefore not applied. This mechanism is a powerful means to prevent a lot of redundant rules and will be discussed in detail in section 4.5.

3.3.2.5 Element Roles

In comparison to the original single–pushout approach there exists only one rule side. In order to successfully describe all changes that a rule application makes in the host graph DMM uses element roles. Every element within a DMM-Rule has a specific element role. They contain more detailed information about the matching of the graph described by the rule and possible changes within the host graph after application. All these rules are explained in the following:

I. exists The most common role for an element is the *exists* role. It simply means that for an element with this role, a corresponding object with the correct type must be present in the host graph for the rule to match correctly. The color black indicates the *exists* role, so that nodes with this border color and edges in it have the *exists* role. For example in figure 3.5 the edge "properties" has this role.

II. not exists The counterpart to the *exists* role is the *not exists* role. If an element has this role, a corresponding object or link must not exist in the host graph at the place indicated by the rule, otherwise the rule would not match. If more than one adjacent elements have the *not exists* role they build a cluster that must not exist in the host graph likewise. The visual representation is also the color black, but additionally the elements have a prohibitory sign.

III. destroy Another role elements can obtain is the *destroy* role. Like the *exists* role the so marked elements must have a corresponding element in the host graph, but additionally an action after the matching process is described for a full application of the rule. Elements marked with destroy, like the name already implies, are being destroyed on rule application. Links to destroyed objects are automatically destroyed, too. This means they do not exist any longer in the host graph after a rule containing such elements was applied. Such elements are distinguished by the color red. In 3.5 the edge with the label "carriedby_edge" has this role.

IV. create The complement to *destroy* is the *create* role. For each element with this role, an corresponding link or object is created in the host graph on rule application. Visually these elements are represented in green color. The example 3.5 shows the edge "carriedby_node" with this role.

[1] in (Hau05) different premise rules with the same signature were supported allowing the same rule invoking these to match at every condition these different premise rules were expressing

3.3.2.6 Quantifications

In addition to the role of an element there exists the possibility to set a quantification for each element. This can enable more than one matching in the host graph of a single element. The most common quantification of an element is 1. This expresses – in the case it is applied to a node – that the node only matches to exactly one object within the host graph. In the case of an edge it means that there is only a direct connection between the two objects corresponding to the nodes connected by it. Elements with the quantification 1 are called non-quantified. If no quantification is given, elements are automatically non-quantified. Non quantified elements have a solid border and in case of a edge a solid line. All elements in the example 3.5 have this quantification. Then there is the option to declare quantifications of 0..* and 1..*. Both

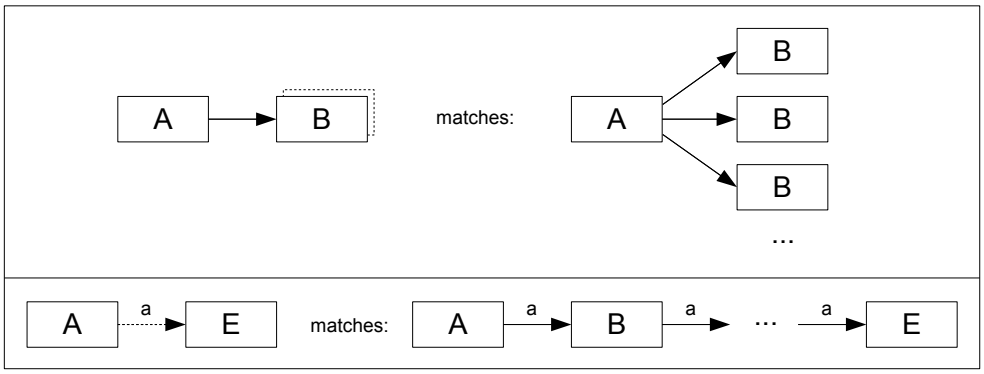

Figure 3.6: Quantifications > 1

mean that a node with this quantification matches any number of objects that occur in the host graph; this behavior is shown in figure 3.6. The 1..* option determines that there must at least be one appearance of a corresponding object in the host graph in the 0..* case a matching is also successful if there are no occurrences in the host graph at all. Elements that have an quantification that matches any number are called universally quantified or simply uqs. To distinguish both uqs types visually, nodes have a double border whereas the outer one has a dashed line. In the 0..* case both borders, the inner and outer one, have dashed lines. Edges are in both cases presented with a dashed line. If more than one uqs node of the same sort are connected with each other, they form a uqs cluster. A uqs cluster must never be connected with other uqs nodes or clusters, otherwise the interpretation will fail.
As an addition to the uqs constructs the quantification for nodes can be set to nested. This is a special case that extends uqs nodes or clusters. While a uqs node or cluster tries to match any possible sub graphs of the host graph, the additional nested node states which structures have to be present, not present, destroyed or created for each matched subgraph. Therefore, every nested node has to be connected to only one uqs cluster or single uqs node. Nested nodes can also be connected to other nested nodes and build clusters that are being treated like single nested nodes. So every uqs construct has exactly one nested cluster possibly an empty one.

3.3.2.7 Attributes

An rather new but very strong enhancement to DMM is the support of attributes introduced by (Bau08). This feature allows the usage of attributes in DMM rules. Classes in the runtime

metamodel can have attributes, so these exist in the objects of a runtime model, too and can be used for a more sophisticated matching of rules and also to compute values by using the concept of conditions and assignments.
Conditions affect the matching of a rule. They are added to nodes and consist of an expression that must evaluate to a boolean value. Conditions are marked with a framed "c" at the beginning directly under the underlined label, containing name and type of the node. In figure 3.5 the EdgeProperties node has the condition "counter>0", so that the rule only matches if the counter attribute in the host graph has a value > 0.
Assignments describe in which way an attribute changes after rule application using an expression. They are presented under the Conditions container marked by a framed "a". This can also be seen in figure 3.5 at the EdgeProperties node. Here the counter attribute, that stores the current quantity of offers residing at an edge, is decreased by one, because one offer moves from the edge to another node after application.

4 DMM Activity Ruleset

This chapter is about the DMM specification that was created for this thesis. First the case study of Jan Hendrik Hausmann (Hau05) will be described and explained how this specification differs from the one created in this thesis. After that the runtime metamodel used for the DMM specification will be described and how it is mapped to the metamodel. Last but not least, the DMM ruleset itself is presented. Because the number of rules grew to over two hundred in the creation process, only the most important ones are explained in detail here. At last some practical guidelines and examples will be given that were found useful during the creation process of this DMM ruleset. This is a great help for those with less experience in this technique.

4.1 Comparison to Prior Work

In (Hau05) Jan Hendrik Hausmann presented a case study where a DMM specification was given for UML2 activities. As mentioned earlier, this was meant to show the convenient application of DMM, it was never meant to cover all elements and packages found in the UML specification for activities. At that point there was no tool support for DMM and everything had to be manually translated into GROOVE rules, which was very error-prone. Also DMM did not support attributes, rule overwriting and quantification of edges.
In this thesis the case study was continued making use of these enhancements to DMM to support more features of UML2 activities. The goal was to support as much packages and elements as possible. There are some rules in the new ruleset that are very similar to the one in (Hau05), others are completely redesigned and of course there are lots of completely new ones. These differences are pointed out in the following.

I. Runtime metamodel The first difference lies in the runtime metamodel and the mapping. The DMM editor uses ecore for runtime metamodels, so we choose to make an extension of the existing ecore UML2 metamodel (Ecl09) as it would have been much more work to create a new one from scratch. Given this fact we could only add new classes and associations for new elements, so there are already a lot of changes in the mapping. For example in (Hau05) all elements of the runtime metamodel were sortable by inheriting from the new class OrderableElement. As this method is impossible to use if you can not alter the original metamodel we introduce the Sorter class in section 4.2 to also achieve an ordering of elements. An ordering of elements is in some cases needed as illustrated in figure 4.5.4.

II. Ruleset With the support of attributes the ruleset supports weight at activity edges, therefore all rules that specify offers or tokens passing edges were adjusted and especially the rules concerning control nodes were redesigned. Guards are supported as true and false value specifications to route tokens at a decision node. With the possibility to use

rule overwriting and attributes there was a big redesign of all rules for object nodes. The new ruleset supports the semantics of upper and lower attributes, upperbounds. The action execution semantics have been redesigned to support weight at activity edges, pin attributes and streaming parameters. New activity elements that are supported include structured activity node, interruptable activity region and exception handlers.

4.2 Runtime Metamodel

The runtime metamodel used for the DMM specification of UML2 activities is an extension of the UML2 metamodel for activities. In the following, these extensions will be described in detail and how they are connected to the original metamodel. Like it was stated in the introduction of DMM in section 3.2.2, an extension adds elements that are needed for the execution of a model by adding runtime information. The key elements that have been added for this purpose can be seen in figure 4.2, where the elements with the dotted line belong to the original metamodel.

4.2.1 BehaviorExecution

The first new element is the class BehaviorExecution, which has an association to the original element Behavior with the name executes. Every time a behavior is executed such an element indicates whether the execution is active through an "executes" link to the Behavior object it is associated to. Additionally it is shown if one behavior execution invoked another by using

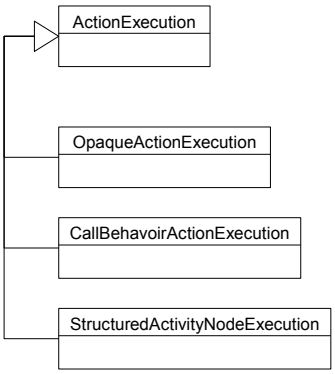

Figure 4.1: Different classes for action executions

an "invoked" link. Two classes inherit from BehaviorExecution: ActivityExecution and ActionExecution. ActionExecution provides an additional association to the Action class named "executes_action". This is necessary for action executions to be treated like behavior executions in most rules like it is suggested in (Hau05), because the Action class inherits from ExecutableNode and not Behavior. By doing this, action executions can be treated like behavior executions in the rules that determine the invocation and termination of behaviors, as a result we receive an ordered tree structure of behavior executions, that tells which execution invoked another. This is a convenient way to manage creation and termination of executions, especially if call behaviors and exception handlers have to be considered as shown in sections 4.4.3 and 4.4.4. For every action there exists a corresponding class for their execution. They all inherit from

ActionExecution and can be seen in figure 4.1. At completion of this thesis the ruleset supports the execution of opaque actions, call behavior actions and structured activitiy nodes.

4.2.2 Tokens and Offers

If an instance of an activity is converted to a runtime activity for rule application, the class ActivityExecution linked with "executes" to the Activity class can be viewed as the root element of this host graph. ActivityExecution is a container for the elements Token, Sorter and Properties. Sorter is a class, as the name implies, to sort certain elements that is sometimes needed to properly traverse element structures or to keep elements in an ordering. The Properties class is used to store additional information about elements, which can be used for example to keep track of the number of tokens or offers on certain elements. This makes it possible to specify

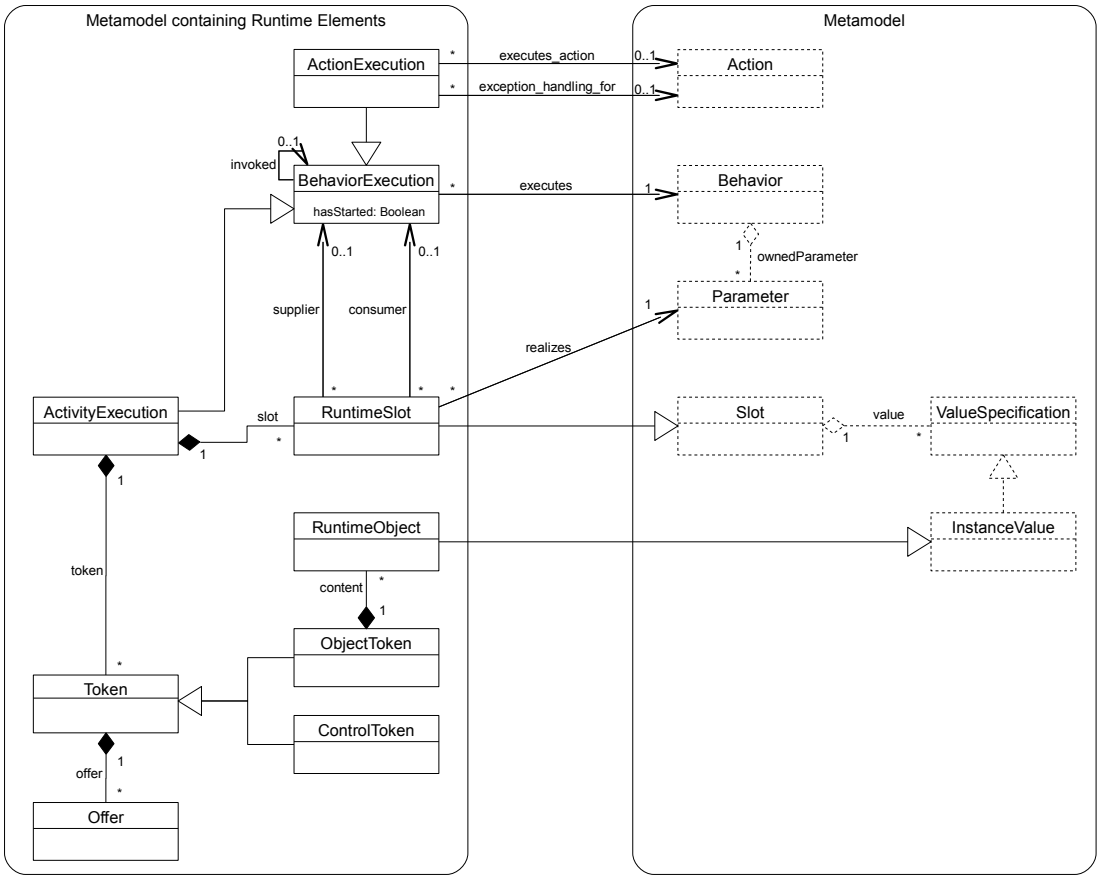

Figure 4.2: Snippet of runtime metamodel

rules for edges with weight or actions that consume more than one token on execution. All cases where properties come in handy are described in 4.2.4, where all classes are explained in detail that are derived from the Properties class. The class Token represents tokens flowing through an activity at runtime. Offer is the class for offers from tokens, they are contained in the corresponding token, so there is a containment association in the metamodel to Token. There are two specializations for tokens. The first is ControlToken for the control flow and the second is ObjectToken. The class RuntimeObject has a containment association to ObjectToken. This

class inherits from InstanceValue and represents the data contained in an ObjectToken. Another new class is RuntimeSlot that inherits from slot. These are used to store runtime objects so actions can consume or supply data from tokens. Whether a consuming or suppling of data is present can be distinguished by the links to the correlating behavior execution.

4.2.3 Sorter

As mentioned earlier, ActivityExecution is a container for the class Sorter as seen in figure 4.3. The purpose of this class is the sorting of activity elements. This is sometimes needed to allow the application of rules on a group of elements in a particular order. To achieve this, the Sorter

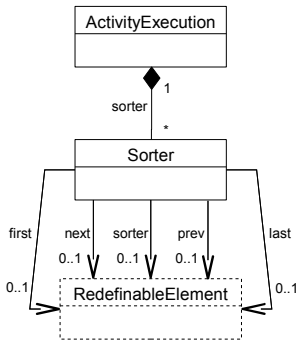

Figure 4.3: Sorter class to sort elements

class has various associations to RedefinableElement, where ActivityNode and ActivityEdge inherit from. There are the associations "first", "next", "prev", and "last" to determine the position of the elements. Finally the "sorter" association specifies for which element the sorting is done. For example, if the incoming edges and input pins of an action are to be sorted, a sorter is added using the sorter association to the action for each such element. While creating these sorters, the first element is tagged with the "first", the last with the "last" and the elements in between with the "next" and "prev" association.

4.2.4 Element Properties

The main purpose of the properties construct is to store additional information for elements to make the definition of rules more convenient. Further some problems can be solved with non local behavior through saving extra information in a properties object. This will become clearer in the following explanation of the attributes. In figure 4.4 it is shown that ActivityExecution contains the Properties class. Properties is a class that has an association with the same name to the class Element. Therefore, a Properties object can be attached to every ActivityElement. The Properties superclass has exactly one attribute "counter". This is mainly used to keep track of the number of offers or tokens residing on elements. This allows rather easy evaluation if weight at edges and the upperbound of object nodes have to be taken into account.
There are four specializations of the Properties class. The first is ActionProperties that has an additional boolean attribute "hasInput" that simply tracks if there is any input on an action. This is needed for the correct matching of the rule that starts the execution of an action, and is further explained in section 4.4.3.

The second is EdgeProperties. It has an additional attribute "weight" that simply stores the weight of the edge. The weight is usually found in a value specification with the link "weight" connected to the corresponding activity edge, but in all rules we use the edge properties and the new attribute to determine the weight. This approach was chosen for convenience, because whenever a weight is not specified, rules that expect it would not match. As a consequence a whole set of extra rules for the case that no weight definition is present would have to be implemented. By setting the default for the weight attribute of the edge properties to 1, this step is not necessary and the rules work as intended for both cases. "offersTested" and "typesMatch" are both boolean and needed for the check if the type of available object tokens is the same or a subtype of an object node so that they can be stored in it.
The third is PinProperties, which contains an integer attribute called "actionTokenCounter" to keep track of tokens consumed or supplied by the action the pin is associated with. This is especially needed for actions that have streaming parameters to correctly determine when the execution of this action finished. The boolean attribute "hasRequiredTokens" has a similar purpose than the "hasInput" attribute of ActionProperties and is also needed for the correct matching of the rule starting an execution of an action. The usage of PinProperties is illustrated in detail in section 4.4.3, where the important rules for the execution of actions are explained.
The last specialization is ParameterProperties. For convenient rule application, it is stored in the

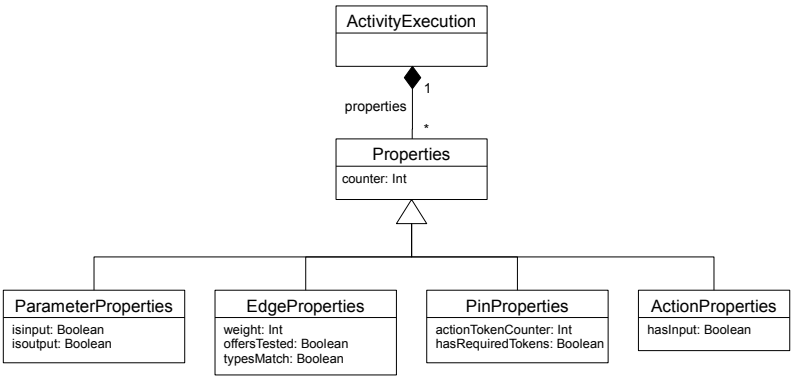

Figure 4.4: The different properties types

attributes "isInput" and "isOutput" whether the direction of the parameter properties is connected to is "in" or "out"[1]. The most notable thing about the ParameterProperties class is the new introduced association "represents_node" to Pin. This is a means to directly know what parameter belongs to which pin on an invoking action and vice versa. The UML2 specification does not designate a direct association to match parameters to pins on call actions. This is simply done by matching type, ordering and multiplicity of the pins to the parameters (Obj09, p. 243). While this definition requires modelers to be rather cautious when creating pins and corresponding parameters, it is unfortunately inconvenient or even almost impossible to define rules for the flow of tokens into and out of call actions without having a direct connection between these two corresponding elements.

[1]Note that the specification also distinguishes between return and inout (Obj09, p. 121). If the direction kind is return, "isoutput" is set to true. In case of inout "isoutput" and "isinput" are both set to true.

4.3 Converting an UML2 Activity to a Runtime Activity

After the creation of the runtime metamodel, the semantic mapping has to be specified. This section describes how the runtime metamodel is mapped to the original metamodel of the syntax definition. This is needed to acquire a runtime activity from a given model on which the graph transformation rules of the DMM ruleset can be applied to. In this thesis the mapping was

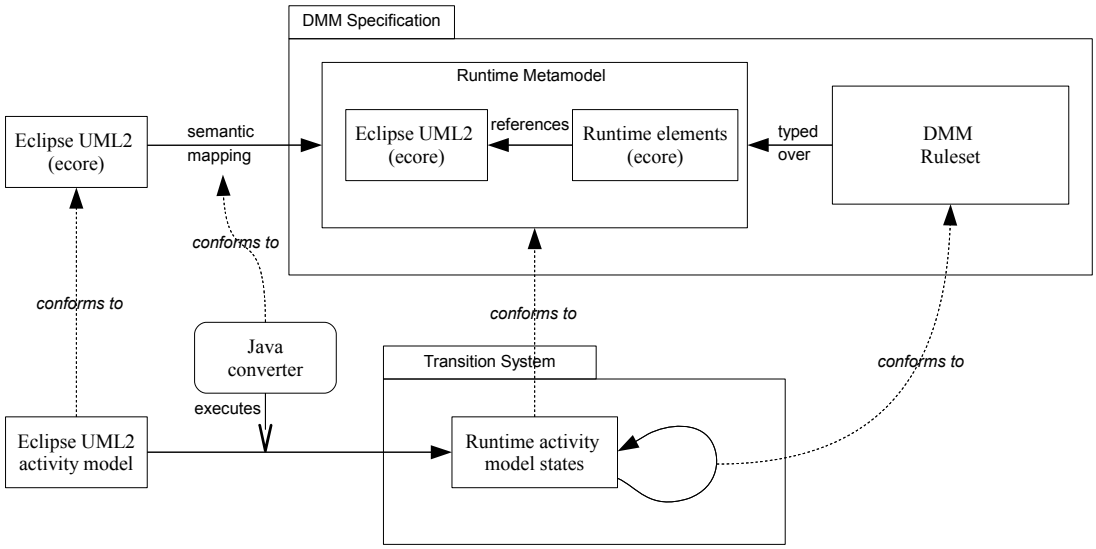

Figure 4.5: Utilization of DMM for UML2 activities

specified using Java. This method was chosen to directly apply the ruleset on activites that were created in the same environment than the DMM editor, Eclipse. In figure 4.5 the DMM environment used in this thesis is presented. There is an implementation of the UML2 metamodel for the metamodel that comes with the modeling framework of Eclipse, ecore (Bud03). This comes with a set of graphical UML2 editors for Eclipse that use this metamodel (Ecl09). Therefore the creation of UML2 models using this metamodel is no problem. The runtime metamodel described in section 4.2 is an extension of the UML2 ecore metamodel. This runtime metamodel is used by the DMM editor for typing of the ruleset. There is an API for UML2 ecore, that is used by the Java converter to parse a model and create the runtime activity according to the mapping described in the following.
At the end of writing this thesis it was still subject of research to find a more generic and convenient way for the mapping. This will be addressed in the outlook in section 6.2.

I. Creating an activity execution At first an ActivityExecution object for the activity that is going to be executed is created and connected with an "executes" link to it. The "hasStarted" attribute is set to the default "false". After the activity execution is created the activity is traversed when certain elements are found within it, sorter and properties objects are created and added to the activity execution.

II. Sorters After the activity execution object has been created, some sorter objects have to be added. There are three elements that need to be sorted for the ruleset to work properly: The outgoing edges of fork nodes, the incoming edges of join nodes and all pins and

edges attached to actions. This procedure makes it possible to create rules that process these elements in a settled order.

III. Properties Finally properties objects are added to the activity execution. The elements within the attached activity that receive these properties objects are as follows: Every parameter gets a parameter properties object. The "isinput" and "isoutput" attributes are set according to the direction kind described in 4.2. If there exists an corresponding pin to this parameter of the invoking action, an "represents_node" link to this pin is created. By doing this we get a direct connection between pins and parameters making the rule definition for call actions that consume and produce tokens possible.
Each pin gets a pin properties object attached. If an input pin has a corresponding parameter, it must be checked whether it is a streaming parameter or not. In the case a streaming parameter is present, the attribute "hasRequiredTokens" is set to true. The rule that starts the execution of an action requires this attribute to be true for every input pin. For streaming parameters, this is already set to true, because here no tokens are required for the action to start as long as there exists another incoming edge or input pin that has a token.
As mentioned earlier, every activity edge receives an edge properties object where the "weight" attribute is set to the actual weight of the edge or – if no weight is specified – it is set to 1. By doing this all rules defined for traversal of edges with weight apply to edges without specified weight as well. All object nodes within the activity get a Properties object to keep track of the number of tokens within them while the execution is running through the counter attribute.
Moreover all actions get an action properties object to provide a simple means to check whether the action has at least one incoming offer or not.

4.4 Creating the Activity Ruleset

This section will describe the created DMM ruleset for UML2 activities. With the ruleset containing more than two hundred rules, displaying every rule would be confusing. Therefore only the most important and interesting ones are explained in detail here, with special focus on those important for the example shown in the beginning in section 2.4. The complete ruleset can be viewed on the CD-ROM delivered as part of this thesis.

4.4.1 Starting an Activity Execution

In the previous section it was described how a runtime activity is created from an activity. Such a runtime activity has one ActivityExecution object with the attribute "hasStarted" set to false. To actually start the execution of the whole activity the bigstep rule *activityExecution.startActivity()#* as seen in figure 4.6 matches on this ActivityExecution and invokes the smallstep rule *activityExecution.start()* shown in section 4.7. It sets the "hasStarted" attribute to true to indicate that the execution process has started and to prevent the matching of *activityExecution.startActivity()#* after the rule and its invoked smallstep rules have been applied, otherwise the rule would continue to match and produce an infinite number of activity executions. There are three nodes connected with an "activity" link to the Activity in this rule with

a quantification of 0..* each having an invocation. The first is InitialNode with the invocation named createToken with the ActivityExecution as parameter. The quantification of the node serves for the purpose that the invocation is applied to all initial nodes belonging to the Activity that has been started.
The invoked rule can be seen in figure 4.8. Each InitialNode of the Activity receives a new crea-

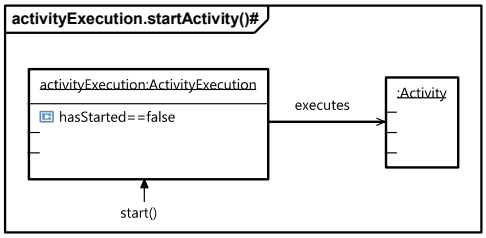

Figure 4.6: Bigstep rule to start the execution of an activity

ted ControlToken and a corresponding Offer. ActivityExecution as the parameter was necessary to correctly create the containment link of the new token. This semantics is described at (Obj09, p. 378): "A control token is placed at the initial node when the activity starts, but not in initial nodes in structured nodes contained by the activity. Tokens in an initial node are offered to all outgoing edges."
The other invocations in *activityExecution.start()* serve for a similar purpose. If the activity is an invoked behavior that has activity parameter nodes, the invocation *activityParameterNode.createToken()*, like the name implies, creates a token at an activity parameter node. The

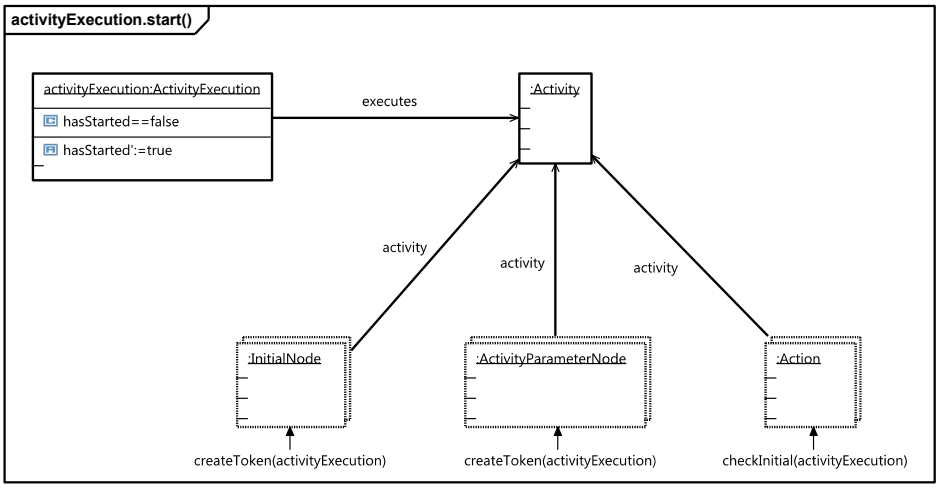

Figure 4.7: Smallstep rule for the start of the execution of an activity

details to this are stated in (Obj09, p. 337). If values have been passed into such an invoked activity, tokens are created for these values, otherwise an empty object token, the so called null token, is created (Obj09, p. 319). There is another version of this smallstep rule for ActivityParameterNodes that are connected to output parameters and have no outgoing edges. At these activity parameter nodes, no tokens are generated when the activity is started. So this alteration of the rule matches in these cases but unlike the other it makes no changes to the runtime activity.

The last of the three invocations covers the special case that actions can also be treated as initial nodes. This can be found in the specification in the initial node section, where it is stated: "when an activity starts, a control token is placed at each action or structured node that has no incoming edges, except if it is a handler body" (Obj09, p. 378).
There exists more than one version of the rule *action.checkInitial(ae:ActivityExecution)*. They

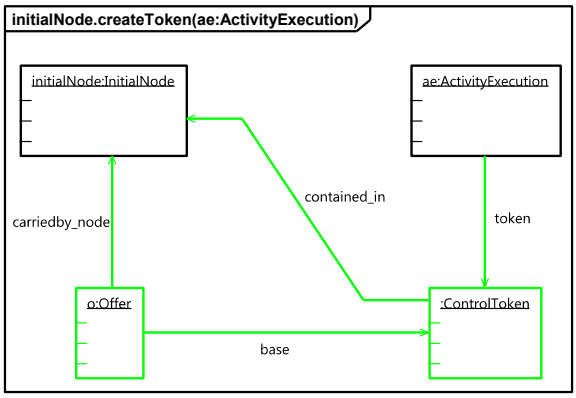

Figure 4.8: Rule to create a control token at an initial node

simple match if there are incoming edges, input pins or the action is a handler body and no changes are made. One of these rules only matches in the opposite case, if there are no incoming edges, input pins and the action is no handler body. In this case a control token is created and placed at this action, so that it can execute.

4.4.2 Fork Node and Join Node

This section covers all significant rules for fork and join nodes. Before the actual rules are described, there is always a short description of the semantics given for the reader unfamiliar with the theme at hand. To keep the logical order, we start with the fork node. Here the focus is set on when and how the copy process of tokens is done. Afterwards, the semantics of the join node is given. Before our approach is presented in form of the rules, the challenges that were suggested in the example 2.4 are explained in detail.

4.4.2.1 Triggering the Fork Process

This section specifies the details about the fork node. We chose to stay close to the offer semantics presented in (Hau05) that slightly differs from what is presented in the specification about fork nodes(Obj09). The difference in the specification to these semantics lies in the copying of tokens at a fork node after an offer of a token was accepted downstream. In the specification it is stated that a token arriving at a fork node is copied as soon as one outgoing edge accepts the token (Obj09, p. 376). The approach presented in (Hau05) is driven by the idea that tokens only traverse edges and control nodes when their offers are accepted downstream and therefore a much more strict realization of the idea that tokens can not be held by control nodes. To explain how a fork node works a small example is used, shown in figures 4.9, 4.11 and 4.13 that are similar to the one found in (Hau05, p. 139).

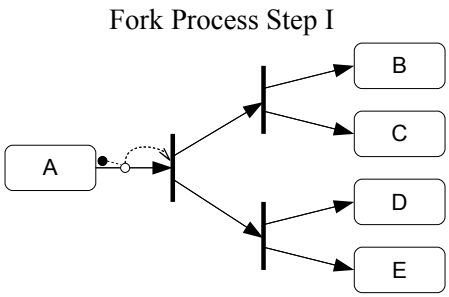

Figure 4.9: Offer from token arrives at a fork node

In figure 4.9, we see a token created by an action named "A" presented as a filled black dot. From it an offer, presented as a white dot of the same size, is put on the outgoing edge of the action. The token is therefore offered to the edges target, a fork node. This triggers the fork process in which the offer is copied to all of the fork node's outgoing edges.
The most significant rule for the semantics of fork nodes is the bigstep rule *forkNode.getOffers()#*.

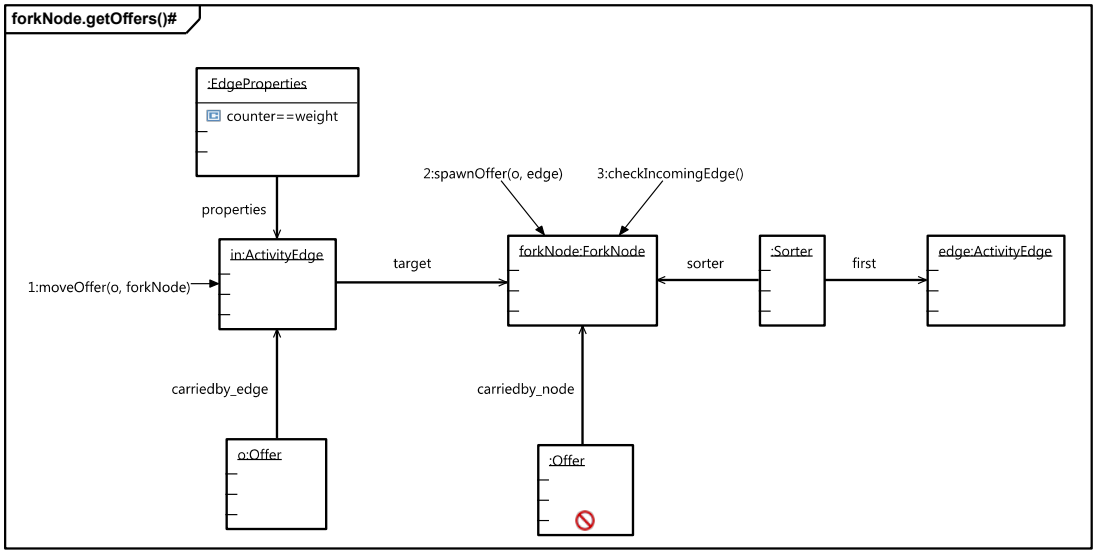

Figure 4.10: Rule that starts the process of copying offers after fork node

It can be seen in figure 4.10. As soon as the incoming edge has enough offers to match the weight specification, the rule matches and the fork process is started.

4.4.2.2 Copying Offers at Fork Nodes

After the fork process hast started, the offer that started it is moved from the incoming edge to the fork node itself using the invocation moveOffer, with the sequence number one. After an offer arrives at a fork node, it is duplicated and offered to all outgoing edges. The original offer is destroyed in the process, but information is being stored in the copies, on which fork node the copy is made and also which path is taken. This can be seen in figure 4.11. All outgoing edges of the first fork node have one offer of the token that is still residing at the action "A". In the right part of the figure it is shown that the copying continues, as the offers arrive once

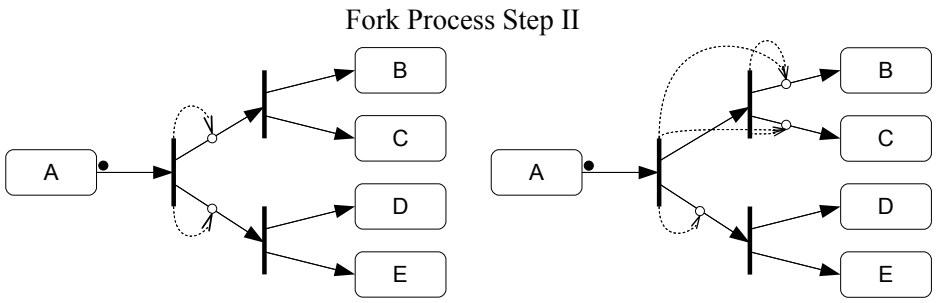

Figure 4.11: Offer gets copied at all passed fork nodes

again at fork nodes. The offer that resides at the upper outgoing edge is once again copied and destroyed. After that the resulting offers have information about two fork nodes and paths that were taken there. After *forkNode.getOffers()#* triggered the fork process its second invocation, *spawnOffer*, starts the copying of the offers. This rule has the fork node as context node and two parameters: the offer that moved to the fork node in advance and the first outgoing activity edge. The rule is shown in figure 4.12. All outgoing edges from a fork node are sorted using sorter objects, therefore there is one outgoing marked as the first one. The invoked rule creates

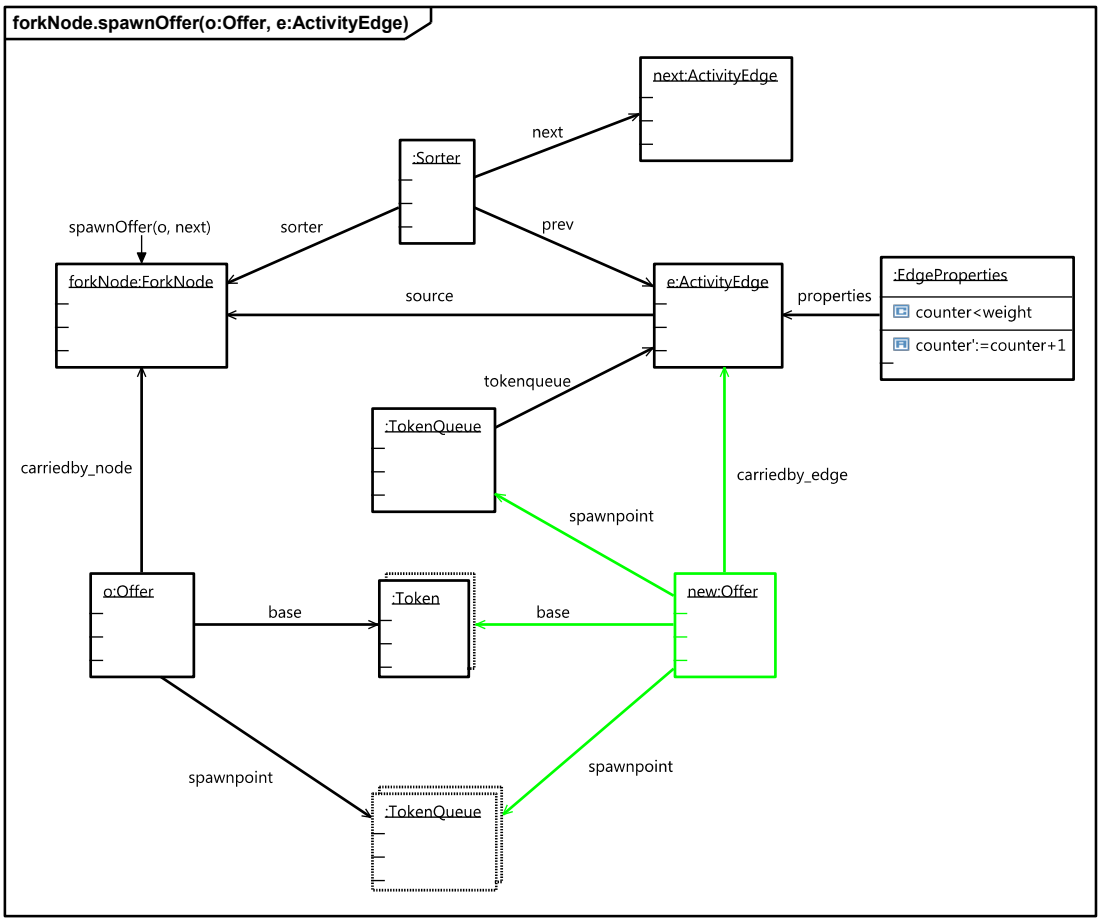

Figure 4.12: Rule that copies offer to outgoing edges of fork node

a new offer object on this edge, raising the counter on the corresponding EdgeProperties object by one. The offer gets a connection to all tokens of the original offer. This can be seen in figure 4.12 and additionally the TokenQueue of the corresponding ActivityEdge is marked with a "spawnpoint" link and if such links of the original offer exist, these are marked for the new offer, too. By marking all passed token queues, the necessary information is saved in the offer which fork nodes it was copied at and what path was taken. There exists one invocation in this rule that serves for the purpose of invoking the same rule for all outgoing edges by always passing the next activity edge as a parameter. If no next edge exists, a variation of the rule will match without an invocation, thus the iteration will stop. This iteration over the sorted outgoing edges is the same rule pattern explained later in section 4.5.4. It is also worth mentioning that the offer on the fork node will be destroyed in this variation, because the containing information was copied to the new offers of all outgoing edges.

4.4.2.3 Enqueuing Token Copies after Offer Acception

If an offer gets accepted downstream, copies of the actual token are made. In figure 4.13 action "B" accepted the offer first; therefore copies of the token are made at the fork nodes the offer itself got copied at. These copies are placed into implicit FIFO queues at all outgoing edges the

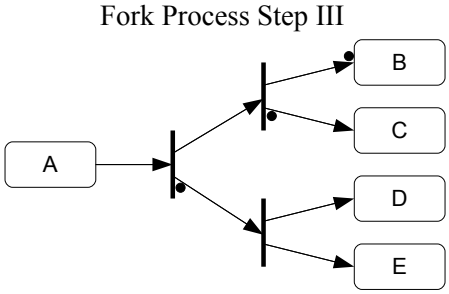

Figure 4.13: Offer gets accepted at action B so token copies are made

offer itself did not traverse. An implicit FIFO queue after a fork node is an exception to the rule that control nodes cannot hold tokens (Obj09, p. 376).
The key rule that specifies the actual copying of tokens, not the offers, at fork nodes is *offer.notifySpawnpoints()* seen in figure 4.14. This smallstep rule is invoked whenever an offer gets accepted downstream. It makes use of token queues connected to each outgoing edge of a fork node via the "spawnpoint" link. There exists a version that just matches if there is no connection to such a token queue, but whenever such a connection is found, the version seen in 4.14 matches and is therefore invoked. This rule first destroys the link to the token queue and marks it by setting the attribute "isSpawnpoint" to true. Marking the TokenQueues is necessary for the first invocation *ForkNode.offerAccepted(t:Token)* 4.15 that iterates over all these token queues at a fork node. For each marked token queue a copy of the token the offer is based on is created and enqueued using the invocation *enqueueCopies*. Although this is invoked on all adjunct token queues using uqs, there exists an extra version of the rule that simply matches and does not enqueue a copy of the token if the queue is not marked. After the application of this smallstep rule, the second invocation named resetSpawnpoints on the context node TokenQueue is handled. This rule simply serves for the purpose of unmarking a prior marked token queue

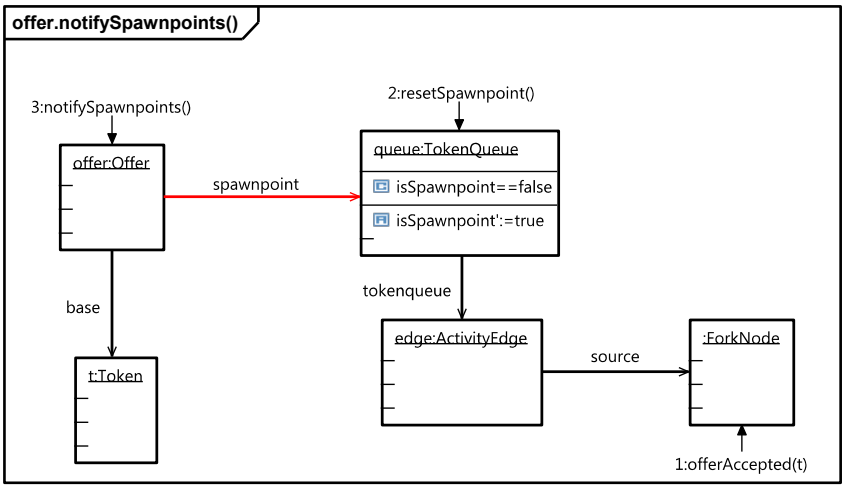

Figure 4.14: Rule that starts the process of enqueuing token copies

by resetting the attribute to its default value.
In the end this rule invokes itself, until all links to token queues have been processed. After these smallstep rules have been handled, the last invocation of the bigstep rule, *checkIncomingEdge()*, is processed.
This rule checks whether there are still offers left on the incoming edge, that have yet to be multiplied by the fork[1]. So if there is at least one offer left on the incoming edge, this last smallstep rule serves for the purpose of starting the whole copy process again.

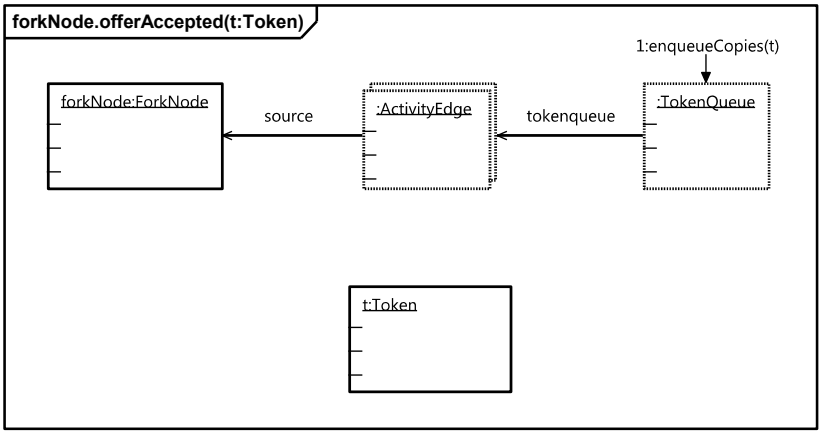

Figure 4.15: Rule that invokes copying of tokes

4.4.2.4 Triggering the Join Process

After the description of the important rules for a fork node, its complement, the join node, will be the subject in the following. The central rule is the bigstep rule *joinNode.flowIn()#*. It

[1]This only applies if the weight at the incoming edge is bigger than one, so that multiple offers arrive at a fork node at the same time.

matches as soon as every incoming edge has enough offers to match the weight specification and if there is no joined offer at the join node itself. To determine the offer requirements for the weight, there is a nested extension using an EdgeProperties object of the uqs used for the incoming activity edges.
Notable here is that the rule was created for the usage of a join specification seen in figure 4.16 as

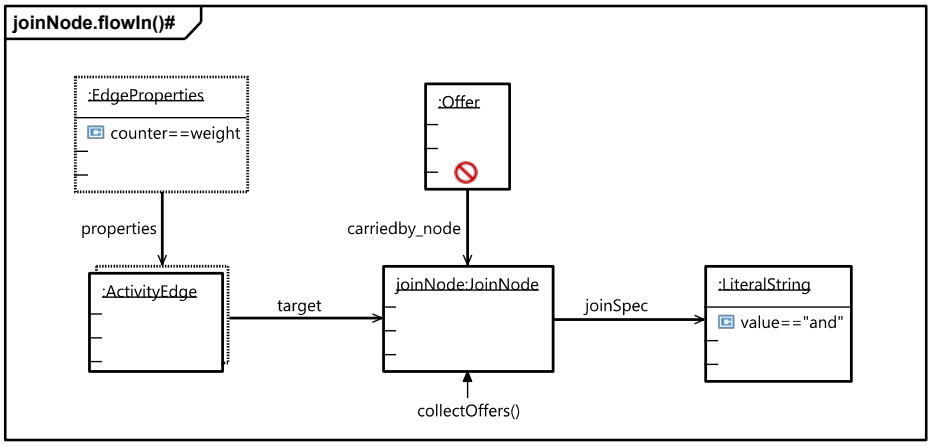

Figure 4.16: Rule to trigger the join process

the LiteralString object linked with "joinspec". Such a join specification states what conditions have to be met for the join node to emit tokens. In this rule the object contains the value "and"[1] that is the default for a join specification (Obj09, p. 382), meaning that all incoming edges must carry tokens.

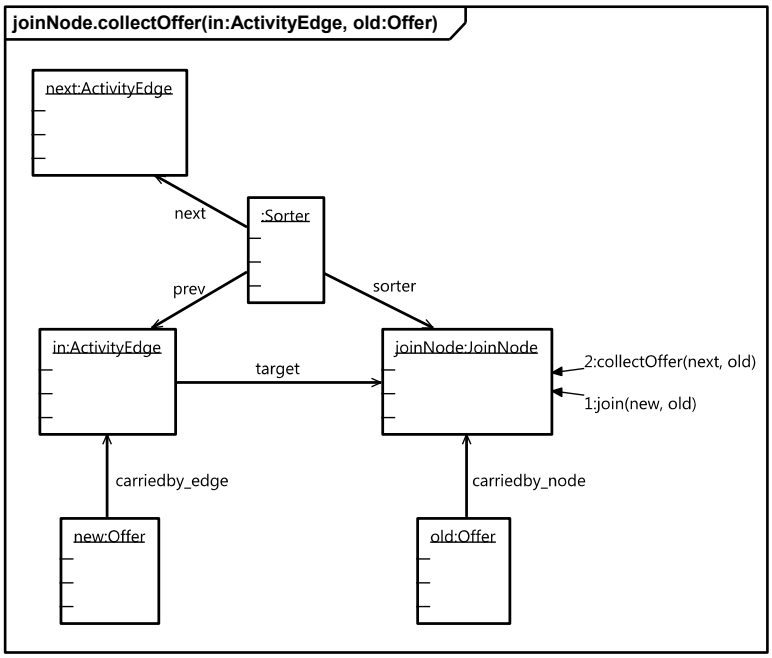

Figure 4.17: Rule that fetches an offer from an incoming edge and invokes joining

If the conditions are met and the rule matches, *joinNode.collectOffers()* is invoked. This rule is used for a similar iteration of the incoming edges as the smallstep rule named "spawnOffer" for the fork node was used to iterate over the outgoing edges. One offer is placed directly at the join node and for each of the remaining offers that are still on the incoming edges, the rule *joinNode.collectOffer(in:ActivityEdge, old:Offer)* seen in figure 4.17 is invoked. If there is a new offer on the incoming edge that is given as a parameter a rule named join is invoked. This rule is responsible for the actual joining of offers at a join node, therefore it has two offers as parameters.

4.4.2.5 Joining Offers at a Join Node

When it comes to the actual joining process, some problems have to be faced within the specification. One minor problem arises from our strict approach that follows the idea that only offers and not tokens arrive at control nodes directly. This approach serves for the purpose that tokens that get stuck downstream, because they are not accepted anywhere, do not remain on control nodes or have to travel backwards. Therefore the offers of tokens are evaluated at control nodes, such as join, fork, decision and merge, instead of directly evaluating the tokens itself. This has the advantage that tokens only pass these control nodes, if accepted downstream, and are not temporary saved there for evaluation. This is a strict realization of the traverse to completion principle explained in section 2.3.
So whenever offers of two tokens are joined, the included information whether they emerged

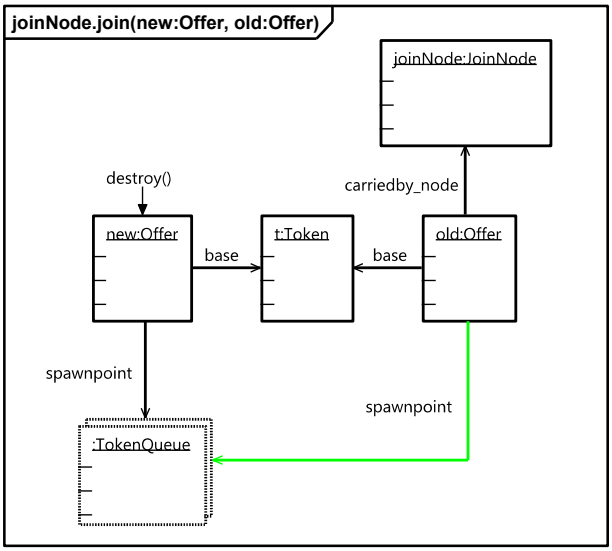

Figure 4.18: Rule to join two offers based on the same token

from fork nodes and what paths have been taken, in form of links to the corresponding token queues, have to be considered. This information is saved in the final offer that leaves the join node after the incoming offers have been processed. It is necessary because if such a joined offer gets accepted downstream, token copies have to be put at all fork nodes the original offers got

[1]The ruleset for this thesis only supports the cases of no given join specification and the default "and", for more join specifications the ruleset can easily be enhanced

copied at as seen in figure 4.13. The means used to save this data is through the "spawnpoint" and "control_spawnpoint" links to the corresponding token queues. How exactly this joining of token queues links is handled is explained later in this section, where the different cases that can occur are explained in detail.
Another problem arises if due to weight definitions more than one token arrives at the same time on the same edge, it gets even more complicated if there are different numbers of tokens on each incoming edge like it is in the example 2.4. There is not much concrete in the specification for these cases, the whole section is kept rather simple only describing how to correctly join one token arriving on each incoming edge. In this case there are two rules (Obj09, p. 382): If there are only control tokens on the incoming edges, one control token is offered on the outgoing edge. If besides the control tokens there are also object tokens then only those are offered. To make these two rules sufficient for all cases it is also stated that multiple control tokens offered on the same incoming edge are combined into one before applying these two rules(Obj09, p. 382). While this provides a means to join multiple incoming control tokens, it was chosen to not support this behavior in the approach for the ruleset. Instead, multiple control tokens arriving at the same edge due to weight specification bigger than one are joined individually. For one thing this is equivalent to the behavior of the fork node, where multiple control tokens arriving at the same time would result in the copying of each incoming token.
For another thing it has to be considered that although it often makes no sense for more than one control tokens to traverse an edge at the same time, it is possible to model it this way (Obj09, p. 325,355) and so a join node should be able to process this accordingly, too. In the following it is presented how all cases are handled within the ruleset by listing all variations of *joinNode.join(old:Offer, new:Offer)*.

I. Joining Offers of Control Tokens The first case seen in 4.18 is the most simple one and describes the case that the new offer that is found at an incoming edge belongs to exactly the same token as the offer that is already on the join node. In this situation if there are any links to a *TokenQueue* at the new offer these are created for the old offer as well and after that the new offer is simply destroyed using an invocation.

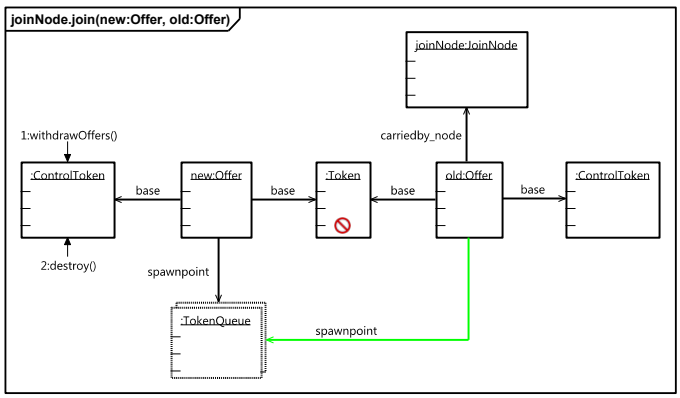

Figure 4.19: Rule to join two offers based on control tokens

The figure 4.19 shows the slight variation that is encountered if both offers belong to different control tokens. The procedure remains roughly the same: At first all found links

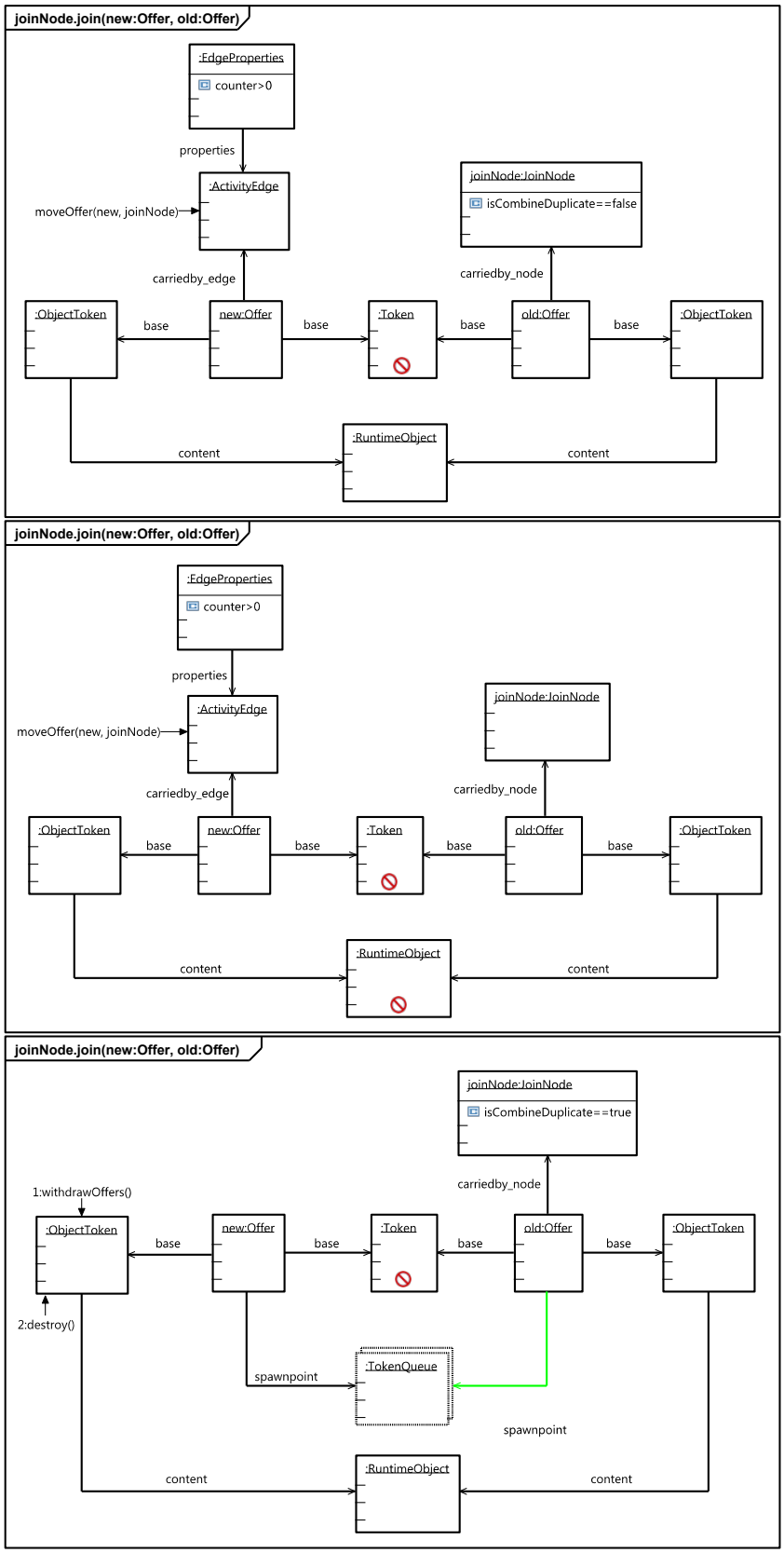

Figure 4.20: Rules to join two offers based on object tokens

to a *TokenQueue* object are created for the old offer, after that not only the new offer, but the corresponding token is destroyed with all its offers.

II. Joining Offers of Object Tokens Figure 4.20 shows how offers of object tokens are joined. The biggest difference here is that offers from object tokens are only joined if they contain the same objects and the attribute "isCombineDuplicate" of the join node is set to true (Obj09, p.383). This case is presented in the last rule below the others. Here once again the old offer receives the links the new one has to TokenQueue objects and after that the new offer is destroyed. The rules above present the cases where "isCombineDuplicate" is set to false or the object of the tokens is not the same. In these cases the new offer is moved to the join node without joining using the invocation *activityEdge.moveOffer(o:Offer, target:activityNode)*. After that the iteration continues with the old offer.

III. Joining Offers of different types of Tokens At last the cases where offers from object tokens and control tokens are joined can be viewed in 4.21. If there is a mixture of control and object tokens on the incoming edges, only object tokens are offered on the outgoing edge of a join node. Therefore the control tokens and its corresponding offers are destroyed and only the offer of the object token remains.
This comes with a problem if the "spawnpoint" links of the two offers are joined in the

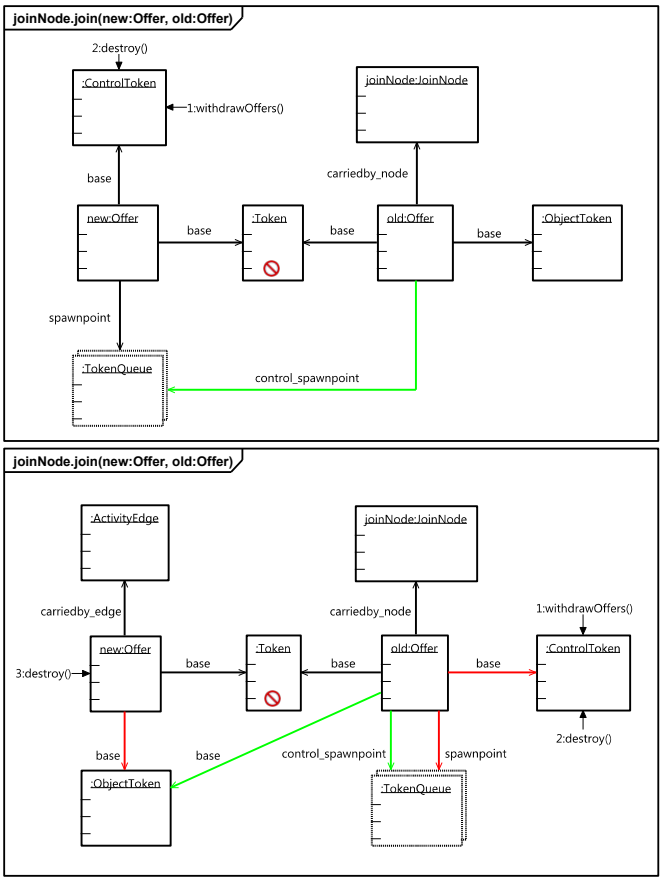

Figure 4.21: Rules to join two offers from different token types

same way as explained in the cases before: As the joined offer gets accepted downstream, only copies of the corresponding object token will be enqueued at the marked fork nodes. This is wrong, because at the fork nodes the offer of the control token got copied, a control token has to be enqueued. Therefore all these token queues receive a "control_spawnpoint" links to distinguish the fork nodes, that enqueue control tokens and those that enqueue the corresponding object token copies if the offer is accepted. There are two variations for the join node for these cases.
The first rule is for the case that the offer residing on the join node has a corresponding object token. Then the offer receives "control_spawnpoint" links for every "spawnpoint" link of the new offer from a control token. After that the control token with the corresponding offers are destroyed. The second rule covers the case that the offer residing on the join node is from a control token and the new offer is based on an object token. This rule destroys the new offer and the control token, after that it attaches the object token to the offer that previously belonged to the control token. Like in the other rule all "spawnpoint" links of the offer for the control token are converted to "control_spawnpoint" links to distinguish on acceptation of the offer whether to enqueue copies of the object token or control tokens.

4.4.3 Action Execution

Before we go into detail and present the most important rules for actions and their execution, a small introduction will be given describing all important steps for such an execution. The standard case for the execution of an action is presented in 4.22. Before the actual execution process starts, offers need to arrive on all incoming edges of the action itself or at the input pins. If all these offers are accepted, the tokens travel to the input pins. In case a control flow is directly connected to the action, control tokens get directly accepted by the action and travel there. All the input pins containing tokens and all directly connected control flows having control tokens can be seen as a prerequisite for the execution of an action. As soon as these requirements are

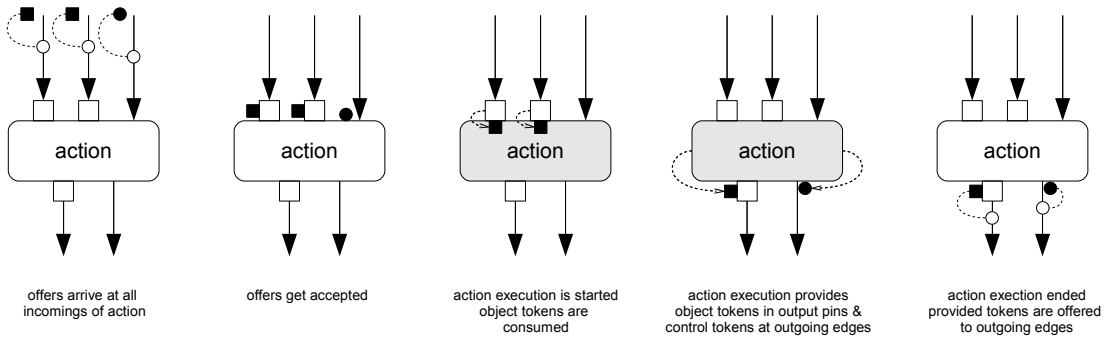

Figure 4.22: Execution process of an action

met, the actual action execution is created and the incoming tokens are processed. While control tokens are simply destroyed, the object tokens are consumed by the execution. This consumption is done differently for each action type. In the case of call behavior actions, the data they contain is passed to the invoked behavior. Whereas in case of opaque actions, the data is simply destroyed. After the execution process has finished, tokens for the output pins and outgoing edges are created. If the output pins contain data supplied from the execution, the object tokens

for this data are created. If there is no data empty object tokens, so called null tokens (Obj09, p. 319) are created. Finally all outgoing control flows receive a control token. These tokens emit offers and the activity execution process continues.
Now that the action execution process has been described, the important steps and the related rules are described in detail in the following.

4.4.3.1 Start of Action Execution

An action execution is started using the bigstep rules with the signature *action.start()#*. For an action execution to start certain conditions have to be met. Especially if streaming parameters have to be taken into account this is a little tricky. An action usually starts if all incoming activity edges have a control token and every input pin has as much tokens as its lower value indicates (Obj09, p. 312, 255). If a pin has a corresponding streaming parameter it does not need a token for the action execution to start, only the so called "non-stream inputs" must be present. There is one exception though: an action execution needs at least one input through a token, whether a control or object token does not matter. So if there are only input pins with corresponding streaming parameters at least one of them must have a token for the action to execute(Obj09, p. 397).
To cover all these cases there is a total number of eight *action.start()#* rules in the ruleset. The

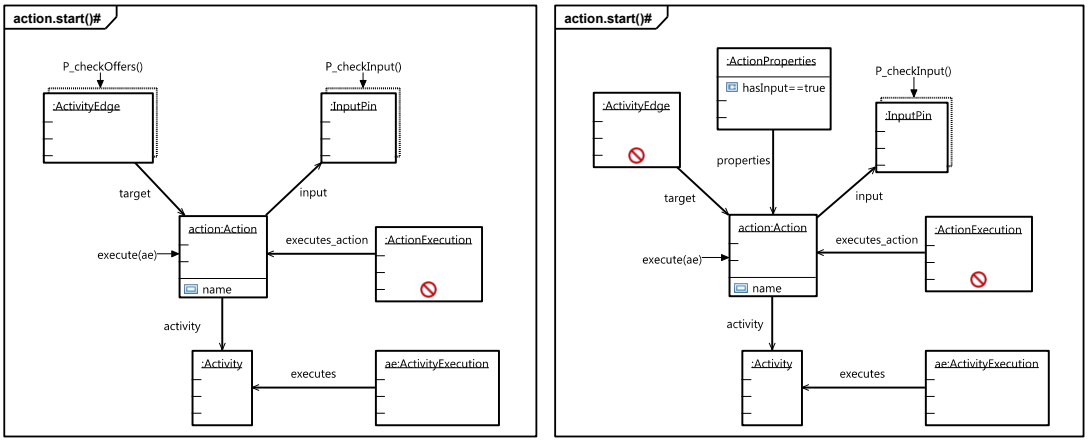

Figure 4.23: Rules to start action execution

two most significant ones are shown in figure 4.23 and are explained in the following. The case where at all inputs, being incoming control flows and input pins, tokens have to arrive for the action to execute, is handled by the left rule in figure 4.23. To check these conditions, two premise rules are used that are explained in the following.
For one there is the rule *P_checkOffers()* invoked on all incoming activity edges, using an uqs. This premise rule can be seen in figure 4.24. Here it is checked, whether there are enough offers on the edge to match the weight condition, so that control tokens can pass the edge and be accepted at the action.
The other premise named *P_checkInput()* rule is invoked using an uqs on all input pins of the action. In this rule it is checked if the number of tokens of an input pin has reached the lower value, this can be seen in 4.25. Interesting here is that the rule does not compare the lower value with the number of tokens directly but only matches if the attribute "hasRequiredTokens"

is set to true. This method was chosen, because if an input pin has a corresponding streaming parameter, the action can already start without the input pin containing any tokens as long as there is at least one token of any kind as input. As mentioned earlier, for corresponding streaming parameters this attribute is always set to true 4.3 and in all other cases it is set to true whenever the number of tokens that are being saved in the pin reaches the lower value. This

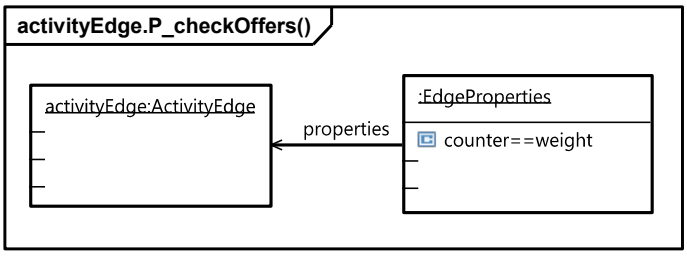

Figure 4.24: Premise rule to check offers on activity edge

can be seen in chapter 4.5.2, where the rule overwriting principle is explained and it is shown that unlike for other object nodes, this attribute is checked and set for input pins. The advantage in matching against the "hasRequiredTokens" attribute is that the same premise rule can match in both cases. Because of these premise rules, the big step rule only matches and invokes the actual execution of the action, if all incoming edges have enough offers for control tokens to be accepted and all input pins contain enough tokens.
The other *action.start()#* rule presented here covers the case that an action has no incoming control flow. Here every input pin needs to have a corresponding streaming parameter or enough tokens to match its lower value. This is simply done like it was in the left rule, using the premise rule *P_checkInput()*. But this alone is not sufficient here, because it can happen that there are corresponding streaming parameters on all input pins. So it also has to be checked whether there is at least one token available as input or not. This is done with the attribute "hasInput"[1] of

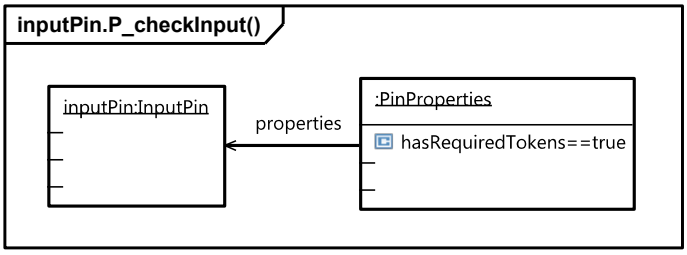

Figure 4.25: Premise rule to check tokens in input pin

the action properties attached to the action itself. This attribute is set to true as soon as a token is put into one of the input pins. This process can also be seen in section 4.5.2, where the rule overwriting principle is explained on this example.
If *action.start()#* matches the invocation *action.execute(ae:ActivityExecution)* is invoked to start the execution process.

[1]This attribute does not need to be checked in the rule on the left, because there is always input through the incoming activity edges, otherwise the rule would not match in the first place.

4.4.3.2 Consuming Input

The first step after an action has been started is creating an action execution object. After that the inputs are being collected. For opaque actions this is rather uninteresting, because here the tokens are destroyed after they have been accepted. So in the following we will concentrate on call behavior actions, where the data in the tokens is passed to invoked behaviors.

To collect all input tokens all incoming control flows and input pins are traversed. The rules

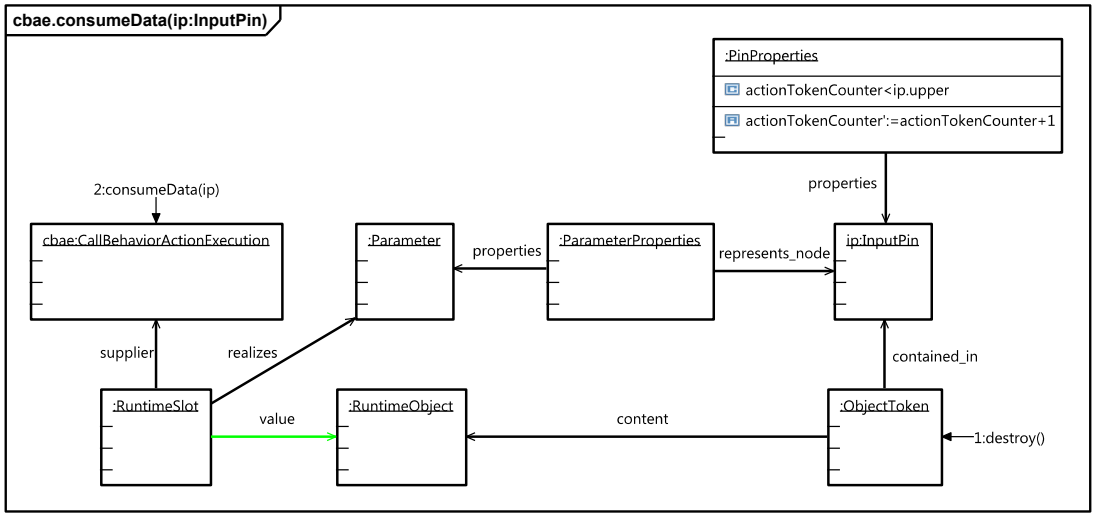

Figure 4.26: Rule to put data from a token into a runtime slot

for this loop are explained later in section 4.5.4. Whenever there is an control flow the attached offers are accepted and in case of an input pin the tokens are consumed. To achieve this, a runtime slot is created for every parameter before the behavior of a call behavior action is actually executed. Using these runtime slots the data is passed to the invoked behavior and vice versa, in case of output pins. The rule *cbae.consumeData(ip:InputPin)* in figure 4.26 shows the rule to put a runtime object from an object token that is contained in an input pin to the corresponding slot. This is done by creating a new "value" link from the runtime slot to the runtime object and

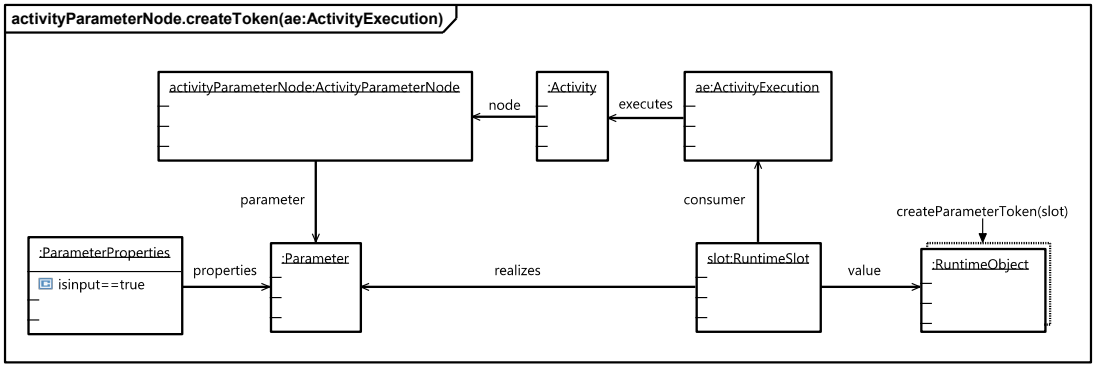

Figure 4.27: Rule to create tokens at an activity parameter node

after that the object token that carried this object is destroyed using the *token.destroy()* invocation.

This is done by invoking the rule in itself as long as the "actionTokenCounter" attribute of the pin properties that is attached to the input pin is lower than the "upper" attribute of the input pin. "actionTokenCounter" is used to count the tokens that were consumed by the action. This is needed for streaming parameters, where the tokens are not consumed all at once. Therefore the need arises to always remember how many tokens have already been consumed by the action. So "actionTokenCounter" is always compared with "upper" before a token is consumed and if it is lower, it is incremented and the token is consumed.
After the runtime objects are put in the slots the invoked behavior is started. In case this is again an activity the rule *activityExecution.start()* seen earlier in figure 4.7 is invoked. Now to all pins

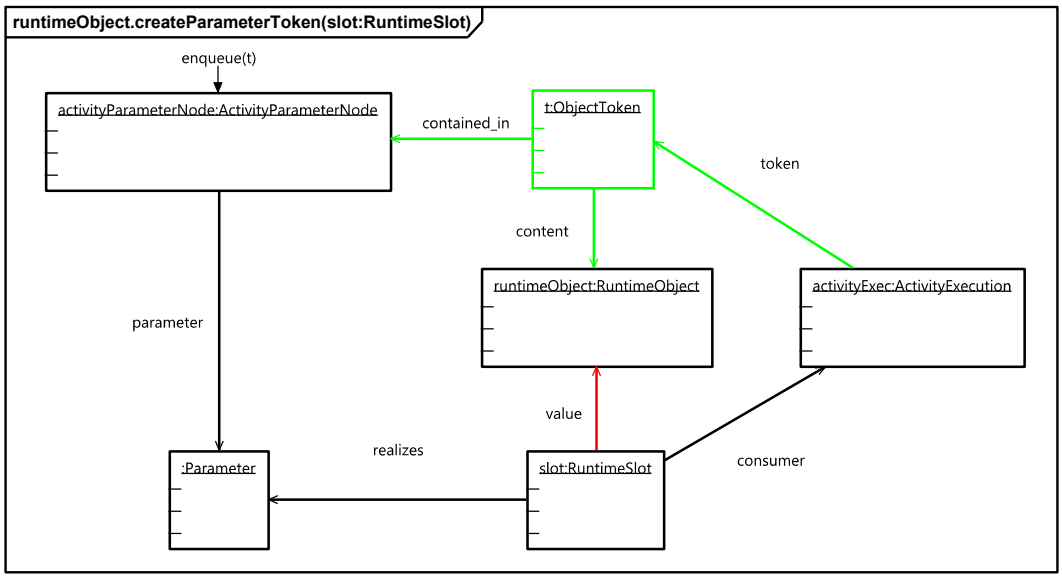

Figure 4.28: Rule to create a token for a runtime object

there are corresponding activity parameter nodes in this invoked activity and so the rule *activityParameterNode.createToken(ae:ActivityExecution)* is invoked for all of these. There are two versions of this rule: one for parameters corresponding to output pins and one for input pins. The one for the output pins simply matches and does no changes. The rule for the activity parameter nodes corresponding to input pins is shown in figure 4.27. Here for every runtime object that is contained in the runtime slot *runtimeObject.createParameterToken(slot:RuntimeSlot)* is invoked. This rule is shown in 4.28. Here the "value" link to the runtime slot is destroyed and the runtime object is attached to a new created object token. That is enqueued in the activity parameter node.

4.4.3.3 Providing Output

After an action execution has finished executing, the outputs have to be created. This means control tokens have to be placed at the outgoing control flows and tokens have to be put in the output pins. In case the action is a call behavior action and there is data in the slots from the invoked behavior execution that has finished its execution this data has to be supplied to the output pins first.
For this all outgoing edges and output pins are traversed with the same method as it is done

with the inputs. On every outgoing control flow a control token is created and placed. For every output pin that is found the rule *cbae.supplyData(out:OutputPin)* is invoked, to create the necessary tokens. This rule is shown in 4.29, where two other smallstep rules are in-

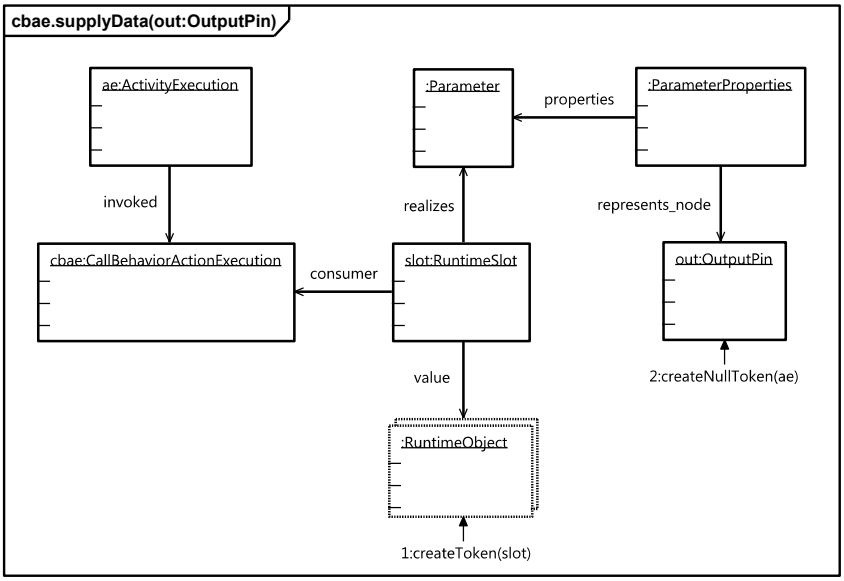

Figure 4.29: Rule to put data from a slot into tokens

voked, that handle the actual creation of the tokens. The first one that is invoked is *runtimeObject.createToken(slot:RuntimeSlot)* on every runtime object found in a runtime slot using an uqs. Here the "value" link from the runtime object, that is the context node, to the runtime slot is destroyed. Then a new object token for the runtime object is created and being enqueued in the output pin using an invocation. After that the "actionTokenCounter" of the pin properties attached to the output pin is incremented. This attribute counts all created tokens after one action execution. A token is created as long as this counter is not greater than the upper value of the output pin. After all the runtime objects have been handled *output-*

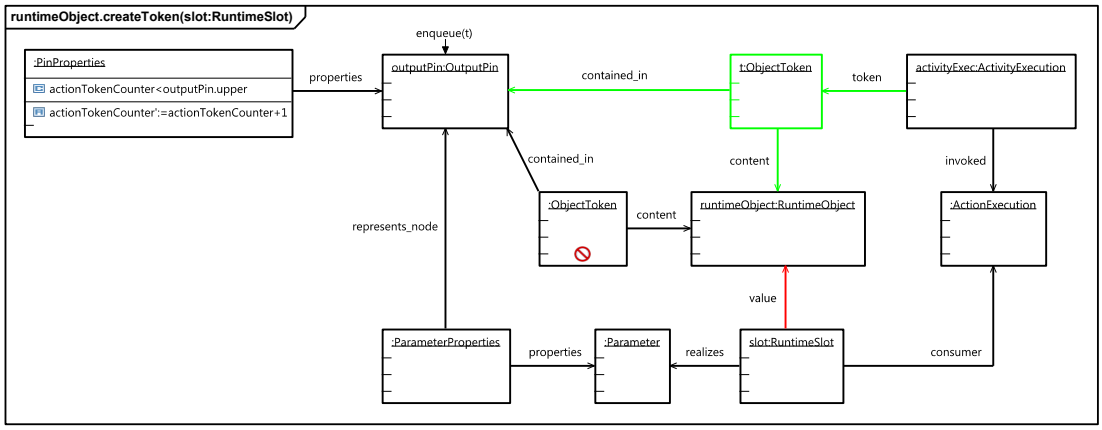

Figure 4.30: Rule to create tokens for a runtime object

Pin.createNullToken(activityExec:ActivityExecution) is invoked. This is done, because there are cases, where an action execution must provide more tokens than there are runtime objects

in an output pin. In these cases an empty object token, a so called null token is created (Obj09, p. 397, 315).
The rule for this can be seen in figure 4.31. Here a object token is created with a runtime object

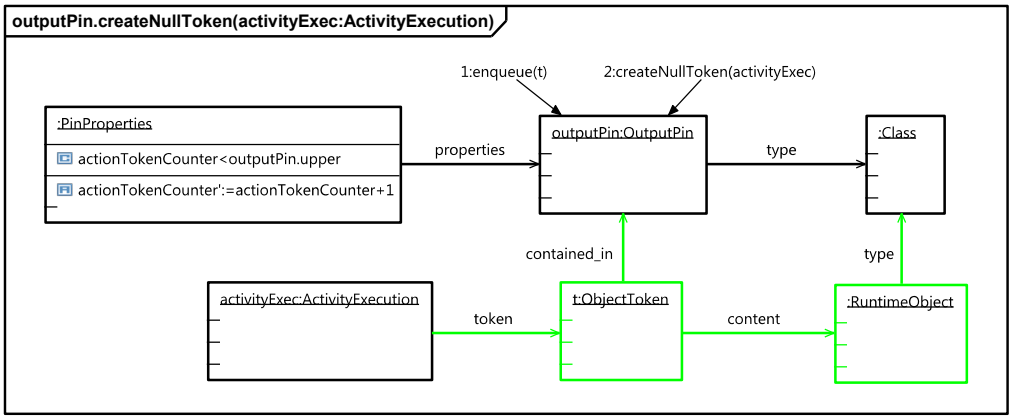

Figure 4.31: Rule to create a null token at an output pin

that contains no value, but has the same type than the output pin. After the creation, this token is enqueued in the output pin using an invocation. This process is repeated by invoking the rule in itself after this until the "actionTokenCounter" attribute is not smaller than the "upper" attribute of the output pin. If this happens, a variation of this rule matches that does not create a new token and does not invoke other rules, so that this process is stopped.

4.4.3.4 Streaming Parameters

If pins have corresponding streaming parameters, tokens can be passed in and out of a running call action. This means that an action can start its execution already without all required tokens available as input. Also tokens can be provided at output pins before the action execution has

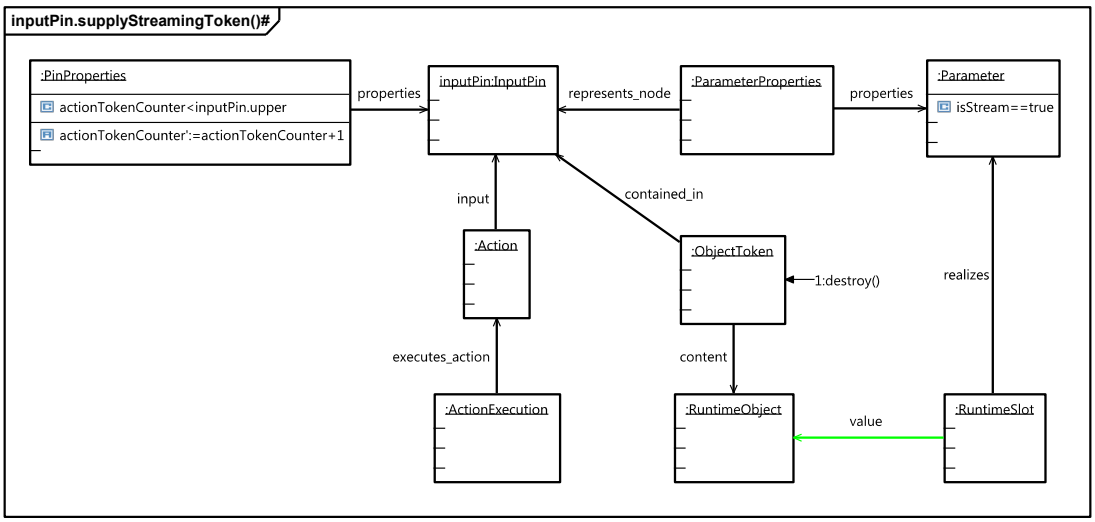

Figure 4.32: Rule for tokens arriving at input pin with streaming parameter

come to an end. To describe these semantics, bigstep rules are needed, that can match during the execution of an invoked behavior.
In our ruleset two of these exist. One for tokens arriving at input pins and one for data arriving at the parameters corresponding to the output pins of an action. The one for input pins can be seen in figure 4.32. This rule matches as soon as the action execution is running and there is an object token enqueued in an input pin that has a corresponding streaming parameter. So whenever this situation occurs the steps to supply the runtime object described in section 4.4.3.2 are taken. First a "value" link from the runtime slot to the runtime object is created. After that the token is destroyed using an invocation, therefore the value can be retrieved in the invoked behavior.
For activities there is another big step rule in the ruleset that simply invokes the smallstep

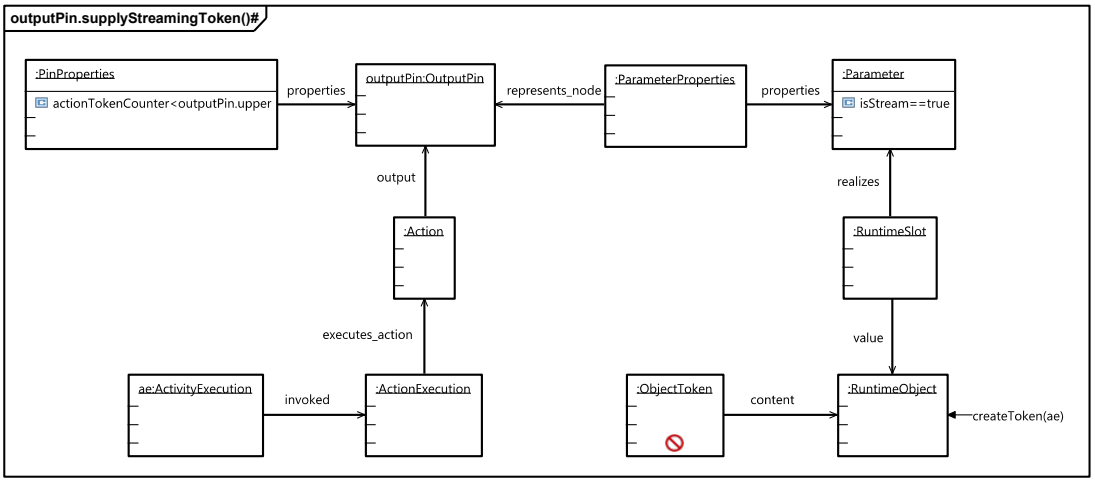

Figure 4.33: Rule for data arriving at output pin with streaming parameter

rule *runtimeObject.createParameterToken(slot:RuntimeSlot)* for all runtime objects without a token, that are in a runtime slot for a streaming parameter. Like the other rules that handle token production and consumption for actions, this big step rule only matches as long as the "actionTokenCounter" has not reached the limit of the "upper" value of the input pin.
There is also a rule for output pin that is shown in figure 4.33. This is simply the complement to the input pin rule: If a runtime object arrives without a token in a runtime slot for a streaming parameter, a token is created using the invocation *runtimeObject.createToken(slot:RuntimeSlot)* presented in 4.30.

4.4.4 Exception Handler

This section explains, how exception handlers are handled by the ruleset. After the conditions for an action to execute are met, the actual execution process is started with the invocation *action.execute(ae:ActivityExecution)*, as described in section 4.4.3. For each type of action there is an adapted rule, the one for opaque actions can be seen in figure 4.34. As mentioned earlier, this rule creates an execution object for the corresponding action. After that the next steps of the execution process are executed using invocations.

If during this execution an exception occurs, it can be caught using exception handlers (Obj09, p. 363). An exception handler defines a type of exception it catches and has a "protectedNode" link

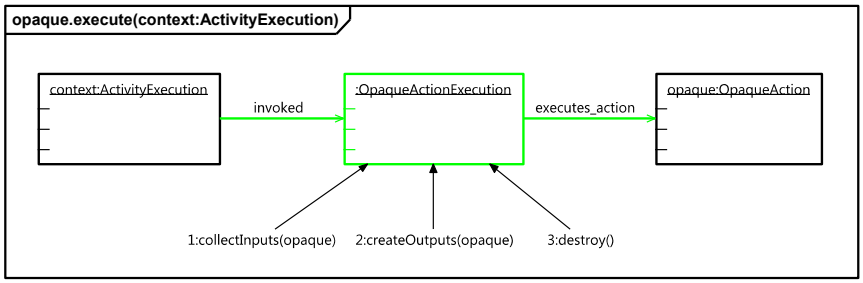

Figure 4.34: Rule starting the execution of an opaque action

to the node it protects. Whenever an exception of the defined type occurs within the protected node the handler body is executed. The handler body is another executable node attached to the exception handler using an "handlerBody" link.

We have no generic method of parsing and evaluating strings in DMM. This would be needed for catching defined exception types within an action execution. Therefore we decided to execute the handler body, whenever a protected node is executed as well, to receive an upper estimation of the transition system. So the transition system always shows the possible path of an occurred exception, even though it might never be taken.
This is achieved by the rule seen in figure 4.35. It has the same signature as the rule seen before

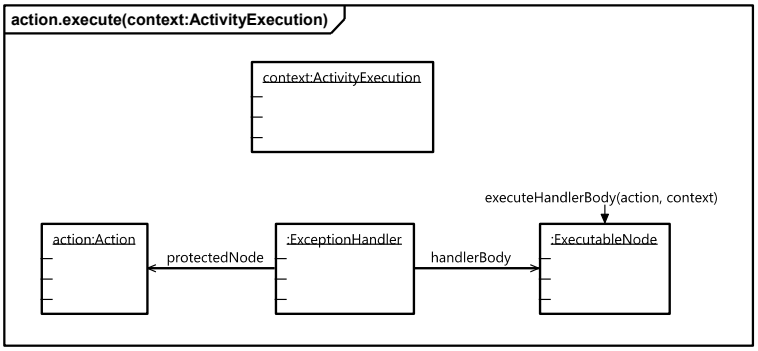

Figure 4.35: Rule that matches if an exception handler is attached to an action that is being started

in figure 4.34 that is invoked on the execution of an action. Whenever an exception handler is attached to the action that is executed this rule matches, too and is applied. On application this rule invokes a rule named executeHandlerBody, seen in figure 4.36, that starts the execution of the handler body. This rule is very similar to the standard rule for executing an action shown in figure 4.34. The only thing that differs is, that the protected node can be found in this rule. On creation of the execution object for the handler body it is connected to the protected node by a link named "exception_handling_for". This is needed to put the output created by the handler body to the protected node as stated in (Obj09, p. 363).

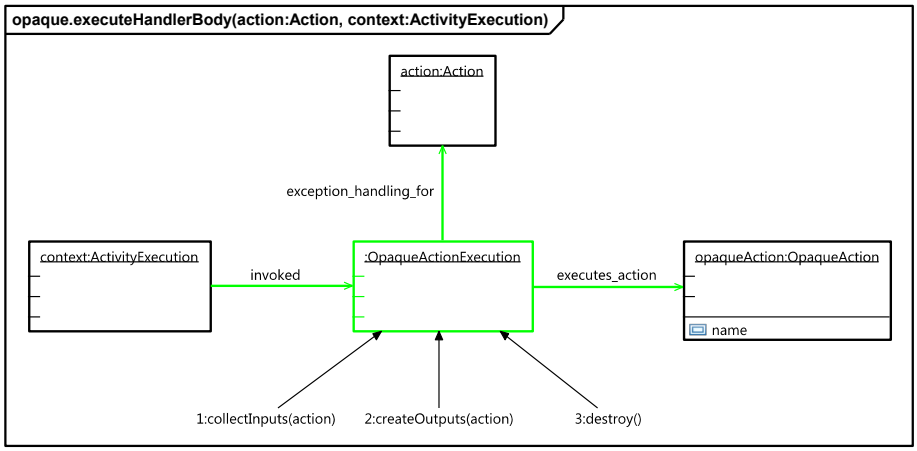

Figure 4.36: Rule to start the execution of a exception handler body

4.4.5 Decision and Merge

In this section the significant rules for decision and merge nodes are explained. We start with the explanation of the decision node. Whenever a token arrives at a decision node it is offered to all outgoing edges. Which of these edges is actually traversed depends on the evaluation of the corresponding guards (Obj09, p. 360).
The specification of this behavior is triggered by the matching of the bigstep rule *decisionNode.flow()#* seen in figure 4.37. Notable here is the used premise rule *activityEdge.P_checkGuard()*

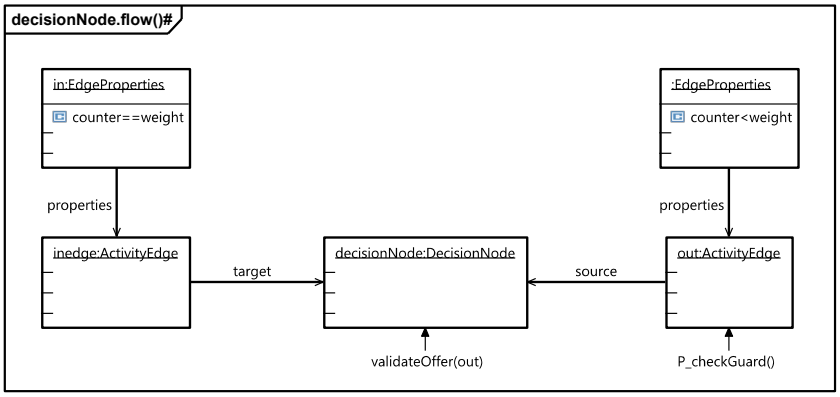

Figure 4.37: Bigstep rule to trigger process at decision node

that is used for a basic guard evaluation. It can be seen in figure 4.38. It only contains a link to a value specification with the value "true", so that the big step rule only matches if the guard of an outgoing edge is set to true[1].

As seen in figure 4.37, there are two other conditions that have to be met for the bigstep rule to finally match: The counter of the incoming edge also has to match its weight specification and

[1]There is also an alternative version of the bigstep rule for the case that no guard is specified, so that tokens also traverse outgoing edges of decision nodes without guards.

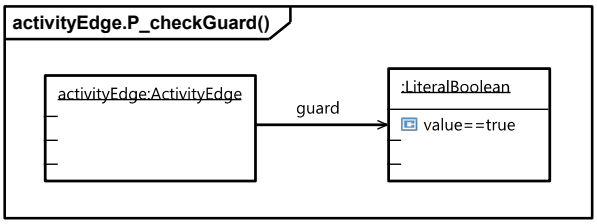

Figure 4.38: Premise rule to check guards

one outgoing edge that has its guard set to true must be able to carry at least one more offer. If these conditions are fulfilled, it means that there are enough tokens offered on the incoming edge that tokens can really arrive at the decision. So the offers of these tokens are passed to the outgoing edges, that are able to carry the tokens and have its guard set to true. This is done, using the invocation *validateOffer*. It has an outgoing edge fulfilling the conditions as a parameter and passes one offer from the incoming edge to it. After that, another rule is invoked that starts this evaluation process again for the remaining offers at the incoming edge. This is repeated until all offers have been passed to the outgoing edges.

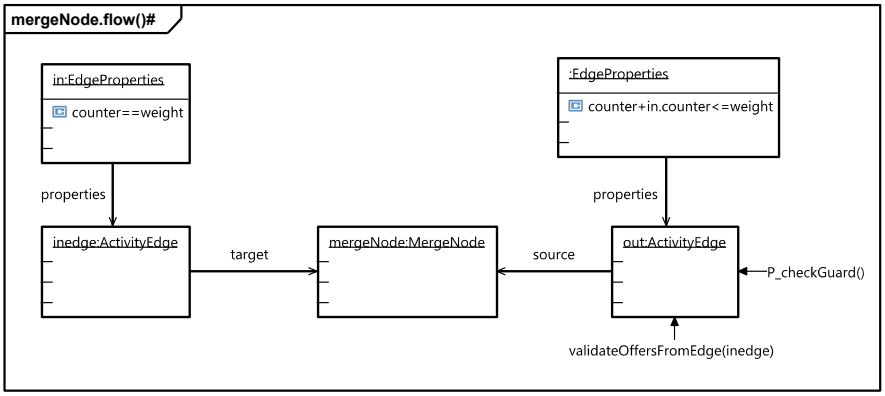

Figure 4.39: Bigstep rule to trigger process at merge node

The complement to trigger the process at a merge node is the bigstep rule *mergeNode.flow()#* shown in figure 4.39. The difference to the rule for the decision node presented in figure 4.37, lies in the fact that the outgoing edge must be able to carry all incoming offers at one incoming edge. Therefore it is checked using the condition counter+in.counter<=weight as seen in the upper right corner of figure 4.39. If this condition is fulfilled, the rule matches and another invocation named *validateOffersFromEdge* passes them all to the outgoing edge.

4.4.6 Coverage of Language Elements

This section is meant to give an overview of all language elements covered by the DMM specification for UML2 activities. It is explained what parts of the semantics found in the specification have been specified and what was left out. The following description is ordered by element type.

Activity Edges The semantics of both activity edge types, control and object flow are specified by the ruleset. Besides the basic semantics the correct handling of weight definitions is supported as well.

Control Nodes All control nodes found in the UML2 specification have its semantics covered by the DMM ruleset. In all the rules for control nodes, weight definitions at activity edges were considered.
The rules that specify the semantics for initial nodes can be found in section 4.4.1. The ruleset also covers the case that actions can be treated as initial nodes.
As a complement to initial nodes, the semantics for the two final nodes, activity final and flow final node, are also specified by the ruleset.
Decision and merge nodes are covered as seen in section 4.4.5 with a simplification for guards. Guards are only supported consisting of "true" and "false" value specifications. If an activity edge has its guard set to false, tokens can not traverse it. This simplification was chosen due to the fact that there is no method to parse and interpret the actual semantics of string value specifications using DMM. To achieve this, a number of rules have to be elaborated, which parse a string and invoke rules according to the semantics found in there. This is of course only realistically realizable, if the form of such string values is known and the semantics described by it is in some way restricted, otherwise the number of rules needed to parse and interpret it would be infinite. Such a method is also called semantic parsing and was not implemented in the ruleset due to time constraints.
For the very same reason, when it comes to the semantics specification of fork and join nodes, the join specification is only supported in its default value "and", shown in section 4.4.2. For every other value, a number of extra rules have to be added.

Object Nodes For all object node types the ruleset supports the check of the upperbound value specification and the type of tokens, whenever a token is placed in it. At completion of this thesis there was only one object node ordering kind implemented: FIFO. Additional ordering types like LIFO could easily be added, but were left out in this first version because of time constraints.
When it comes to activity parameter nodes, the associated parameters are worth mentioning. The ruleset covers streaming parameters as seen in section 4.4.3, but parameter sets are not included. Parameter sets provide alternative sets of inputs or outputs that a behavior may use. It is decided which alternative is taken, by evaluating conditions. To specify such an evaluation, semantic parsing would be required, which was explained earlier. Therefore it was not included in the ruleset.
Pins have its upper and lower attributes evaluated to control the execution of an action, as explained in section 4.4.3.
The only object node that was not fully supported, was the data store node. The permanent saving of tokens in such a node was left out, because of time constraints, therefore it is treated exactly like a central buffer node.

Structured Activity Nodes The ruleset specifies semantics for the standard structured activity node, but all its subclasses are not supported. A structured activity node is a means to add structured activities in activity diagrams. All nodes and edges that belong to such a structured activity are not shared with other activities, including the activity containing the structured activity node. If a structured activity node is invoked, it is treated similar to a call behavior action, with the difference that the invoked behavior is always an activity, specified by the elements in the node itself.

The nodes that inherit from structured activity node are conditional node, sequence node, loop node and expansion region node. To specify the semantics of these nodes, semantic parsing would be needed. This is due to the fact, that these semantics include complex evaluation of values in the execution process, to support loops and conditions. Therefore they were not supported by our ruleset.

Activity Groups There are two activity groups to be found in the UML2 specification: Activity partitions and interruptible activity regions. An activity partition is only used to group similar actions, so that a reader can identify actions that have some characteristic in common more easily. It has no semantics for the execution of activities and therefore does not need to be supported by the ruleset.
In contrast to that, the semantics of an interruptible activity region is relevant for the execution of an activity. If a token leaves an interruptible activity region via an edge designated by the region as interrupting edges, all tokens and behaviors in the region are terminated. This semantics was specified by the ruleset.

Exception Handler As described in section 4.4.4, due to the missing possibility of evaluating whether an exception occurred or not, whenever exception handlers are added to catch an exception, the handler body is always executed, together with the protected node, even though no exception is caught to have this possible execution path in the transition system.

Actions To this end, the DMM ruleset for UML2 activities supports opaque and call behavior actions. The other actions found in the specification were left out mostly due to time constraints.
Without the support of semantic parsing, we had no means to properly check for the occurrence of events. This was needed to specify the correct semantics for accept event actions, therefore these were also left out.

4.5 Guidelines and Patterns for DMM Rules

This section is meant to give beginners an introduction on how to create a ruleset. In (Hau05, p. 153) there is already a chapter about general guidelines, that are not be discussed here. Besides some basic knowledge the focus here is set on patterns that were found during the creation of the ruleset for this thesis. Most of these are explained with concrete examples so that their usage will be clear.

4.5.1 Using invocations to keep rules small

It is already mentioned in (Hau05) that rules should be kept small. This results in rules that can be read more easily. Bigger rules can be kept small by putting functionality of the rule in smallstep rules that are then invoked by the rule. Besides readability there are other reasons, where putting functionality into smallstep rules proves to be worthwhile. If the same functionality can be found in more than one rule, decomposition should be used to put this in a new smallstep rule, for reusability and maintainability.
A good example for such a rule in the DMM ruleset for activities is *activityEdge.moveOffer(o:Offer, target:ActivityNode)* that was shown earlier in figure 3.5. This rule is invoked by a lot of rules, whenever an offer is moved from an edge to a node. There was a point in the creation process

of the ruleset, where a method was needed that checks, as soon as an offer moves over an edge, whether this is an interrupting edge or not. This is done using the invocation also seen in figure 3.5. How this invocation actually works is explained later in section 4.5.3, where this rule is used to explain the marking of nodes using edges. If the functionality of an offer flowing from an edge to a node would not have been outsourced in a smallstep rule, the invocation that checks whether an interrupting edge is passed, would have to be inserted in a lot of rules. By outsourcing of this functionality, this change had to be done at only one place.

4.5.2 Using Rule Overriding

Usually the semantics of objects that share the same supertype are similar in some aspects, therefore leading to the same rules besides the context node. Whenever rules for specialized objects are the same, it is a good idea to specify the rules for a common supertype, to avoid unnecessary rules. If there are one or more derived objects where the rules are not exactly the same it is still possible to specifiy a rule for the common supertype and use rule overwriting for those derived objects, that require the rule to differ. Without rule overwriting the ruleset designer would be forded to create an own rule for every subtype even in cases where the rule for every subtype but one is the same. In the following this method is explained with an example found in the ruleset.

The example is about the rules that describe how tokens are enqueued in object nodes. This semantics are the same for all object nodes, but for input pins we need to alter some additional attributes, that are necessary for the action execution rules. In this case rule overwriting comes in handy. In spite of specifying the same rule for all subtypes of object node it is sufficient

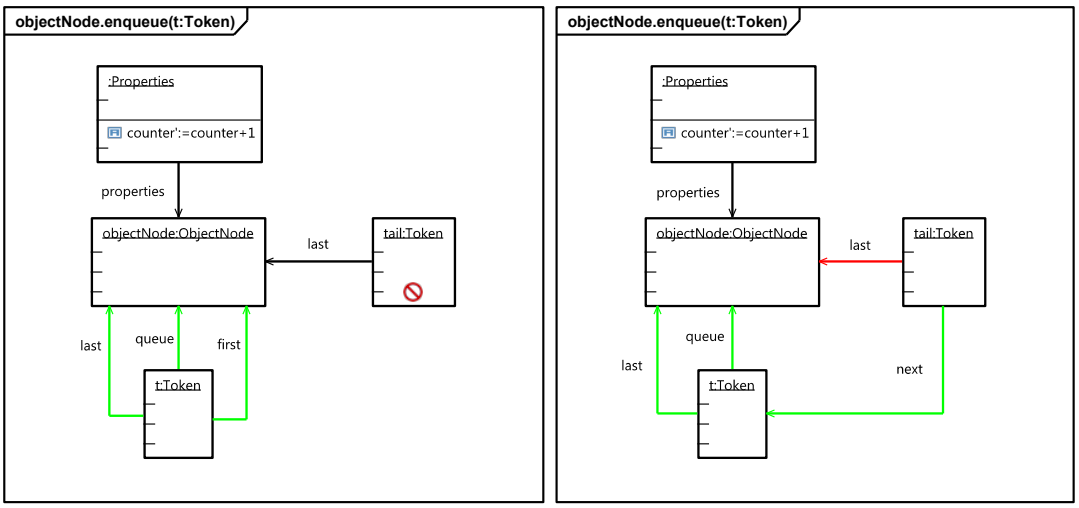

Figure 4.40: Rules to enqueue a token at an object node

to create one rule for object nodes in general. There are two rules for this with the signature *objectnode.enqueue(t:Token)*, shown in figure 4.40. The first rule covers the case if there are no tokens enqueued in an object node. This case is checked using the NAC build of the token named tail on the object node. If the rule matches the new token is placed at the first position

of the queue at the object node and the counter attribute of the pin properties is incremented. The second rule matches in the case if there are already tokens enqueued in an object node. Unlike the first rule here the token receives a position after the tokens already in the object node. Therefore the new element is set to be the last element using the "last" link to the object node and the "last" link of the token that was the last before is destroyed, and a next "link" to the new token is set. These two rules are now used for every object node, whenever it is invoked. As already mentioned we need these rule for input pins to differ a little. Because of that we create new rules especially for input pins. These rules can be seen in figure 4.41. They

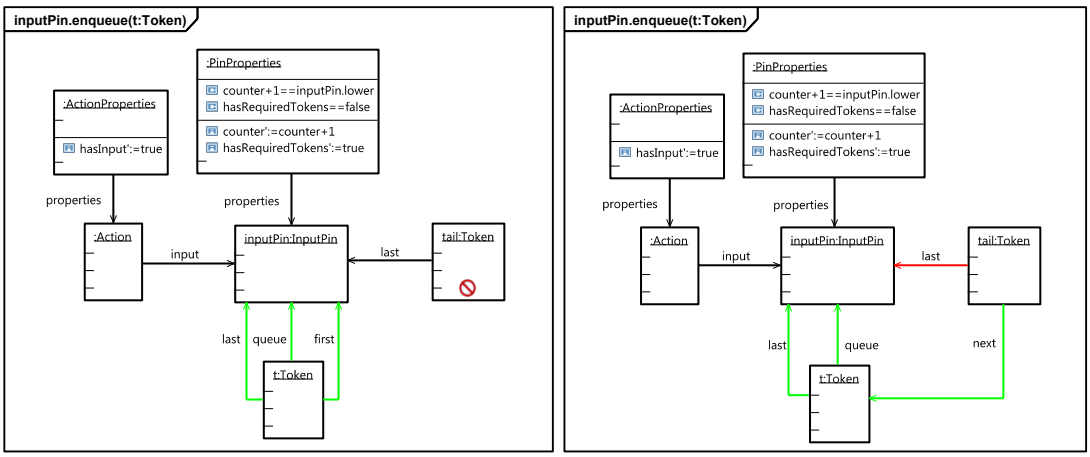

Figure 4.41: Rules to enqueue a token at an inputpin

are very similar to the one specified for object node, but alter some additional attributes. In the pin properties the counter is incremented as before, but additionally if this counter reaches the lower attribute of the pin and therefore the pin contains enough tokens for the action to execute, "hasRequiredTokens" is set to true. Also the attribute "hasInput" of the corresponding action properties is set to true once a token is enqueued in the pin.

With the creation of these two different rules we are not done yet. The problem is that whenever the rule to enqueue tokens at input pins matches, the one for the supertype object node also matches. This is of course unwanted and without the option to set rules to override another, a ruleset designer would be forded to create an own rule for every subtype of object node in this case, even though only the one rule for input pins differs.

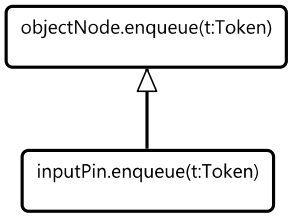

Figure 4.42: The rule inputpin.enqueue(t:Token) overwrites objectnode.enqueue(t:Token)

Therefore after these two rules have been created, it is possible to set the rule for the input pin to override the one with the supertype object node as the context node. This overriding is indicated like a inheritance hierarchy as seen in figure 4.42. This construct servers for the purpose that

whenever this structure is found within the graph only the rule with the input pin as a context node matches.
Technically this is done by inserting a NAC into the overwritten rule in the translation process into GROOVE, where it is stated, that the rule only matches if there is no input pin present. Otherwise both rules would match, which is not wanted. This method saves a lot of unnecessary rules, because if this would not be possible, the ruleset designer would be forced to create an enqueue rule for every object node, even though most are the same.

4.5.3 Marking using Links

The technique of marking nodes using links is a good technique to get over problems that arise while formalizing semantics that are not local. In the DMM ruleset for this thesis this is used two times. First it is used as seen in 4.4.2 to associate all token queues one offer passes to it, so that token copies can be put in the other token queues at the corresponding fork nodes, as soon as the offer gets accepted. Then it is used to remember if an offer passed an interrupting

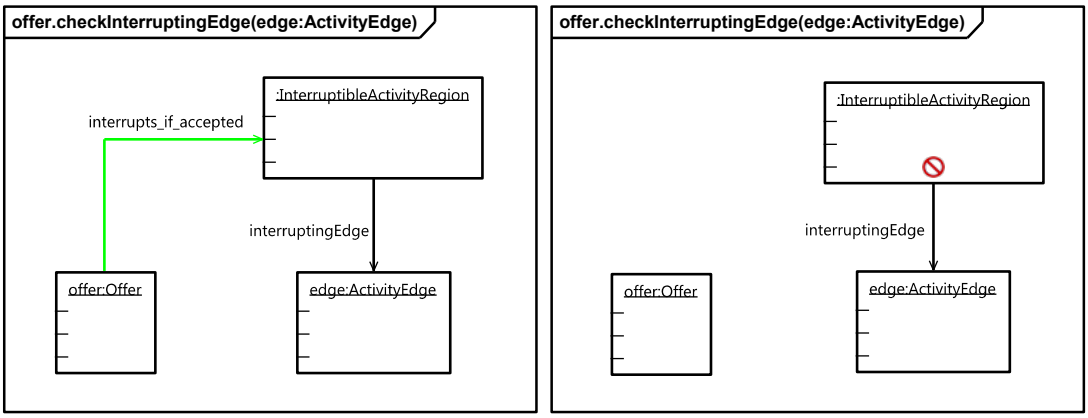

Figure 4.43: Rule that marks interruptible activity region

edge of an interruptible activity region. Whenever a token passes such an edge, the region is interrupted. This means that all tokens in the region are destroyed and all invoked behaviors in it are terminated (Obj09, p. 380). To achieve this, every time an offer passes an edge it is checked by invoking the rule seen in figure 4.43, that receives the passed edge as a parameter. There are two versions of this rule: The one on the right side matches if the passed edge is no interrupting edge and does no changes to the host graph, if this edge actually is an interrupting edge the left rule matches. This rule creates a link named "interrupts_if_accepted" from the offer to the interruptible activity region. If the offer gets accepted, which means that the token itself actually passes this edge and moves to the node where it was accepted, it is checked whether such a link to an interruptible activity region exists or not. If it exists a small step rule is invoked that interrupts this region.

4.5.4 Iteration over Elements

Sometimes the need arises to invoke a rule on all elements of the same type at a certain place without using uqs. This may be because a particular order of rule application is needed or the

nodes are needed as a parameter in another invocation of the same rule, which is impossible with uqs. Notable is that a particular order of rule application results in less interleaving in the resulting transition system. Therefore the transition system becomes smaller, but this is usually payed for by a bigger ruleset. So in this situations the ruleset creator might want to trade off the size of the resulting transition system against the size of the ruleset.

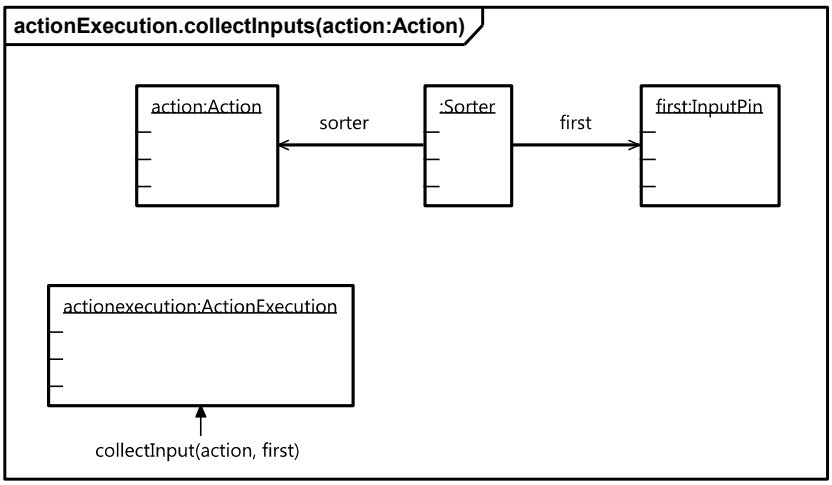

Figure 4.44: Rule to start the collection of inputs at an action

To be able to refrain from using uqs in such a case, sorter objects come into play. If all the elements of interest are sorted, they can be processed in a particular order. In the following this procedure is explained using the rules responsible for the specification of the collection process of tokens from input pins for an action[1].
The process starts with the invocation of the rule *actionExecution.collectInputs(action:Action)*, shown in figure 4.44. This rule invokes *actionExecution.collectInput(action:Action, input-pin:InputPin)*. This is the rule responsible for the collection of the tokens in the input pins. The important fact here is that it is invoked with the first input pin that is sorted. There are two versions of this invoked rule that can be seen in figure 4.45. For the input pin that is given as a

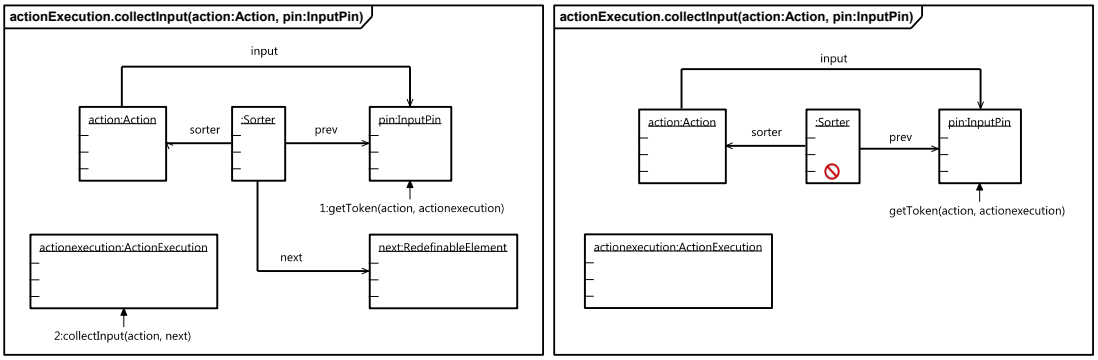

Figure 4.45: Rule to continue the collection of inputs

[1]Note that tokens are also collected from incoming control flows. The additional rules for this are not shown here to keep the focus on the iteration technique.

parameter the tokens are collected using another invocation. After that the rule invokes itself if there is a next input pin marked by a sorter object. This can be seen in the rule on the left side of figure 4.45. If the end of this sorted structure is reached and there is no next input pin to be found, the rule on the right side matches, where the rule does not invoke itself after the tokens are collected, thus stopping the iteration process.

5 Test Framework

After the creation of the DMM ruleset for UML2 activities, this formal semantics can for example be used to test the quality of models using modelcheckers. But if this semantics would not be correct, the results of such tests can not be trusted. This is comparable to the testing of Java programs, when the Java virtual machine that executes it is corrupted.
Therefore the need arises to check the quality of the created semantics. This is done pragmatically following the approach in (SE09), where a test driven approach for DMM specifications is introduced. In this chapter this approach is explained and its application is shown using examples.

In figure 5.1 found in (SE09) a comparison is shown between the testing of software systems in general and the approach to test semantic specifications. Software systems are tested by using some input and the expected result for it. Such a test succeeds if the actual output of the software system is the same as the prior expected result.
By analogy we want to proceed with the testing of semantics specifications. In this domain, a

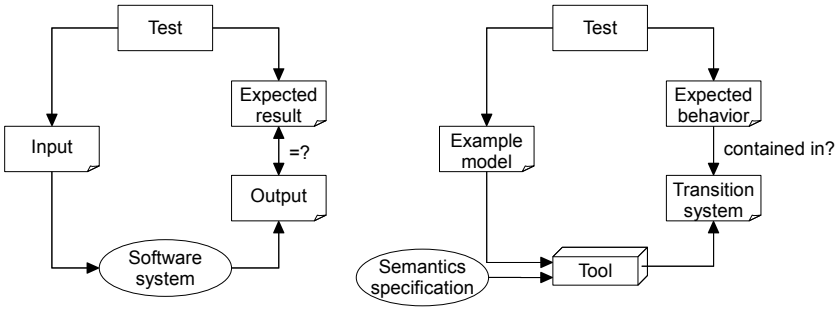

Figure 5.1: Comparison of testing of software systems (left) and semantics specifications (right); the test subject is depicted as an oval.

test consists of an example model ad its expected behavior. From this model and the semantics specification using DMM, a transition system can be computed which represents the model's behavior as explained in section 5.1.3. The test succeeds if the actual behavior found in the transition system conforms to the expected behavior.

5.1 Test Procedure

In figure 5.2 it is shown, how the idea for testing semantic descriptions is put into practice for the DMM specification of UML2 activities. The expected behavior of an activity is described as traces of action execution events. This is shown in the upper left corner of figure 5.2 by "execution sequences" linked to the activity. All the possible traces are kept in a text file in a special format explained in section 5.1.1. These traces are translated into temporal logic

by an algorithm described in section 5.1.2. This algorithm is executed by the so called CTL generator as seen in figure 5.2. The resulting temporal logic formulas are then checked against

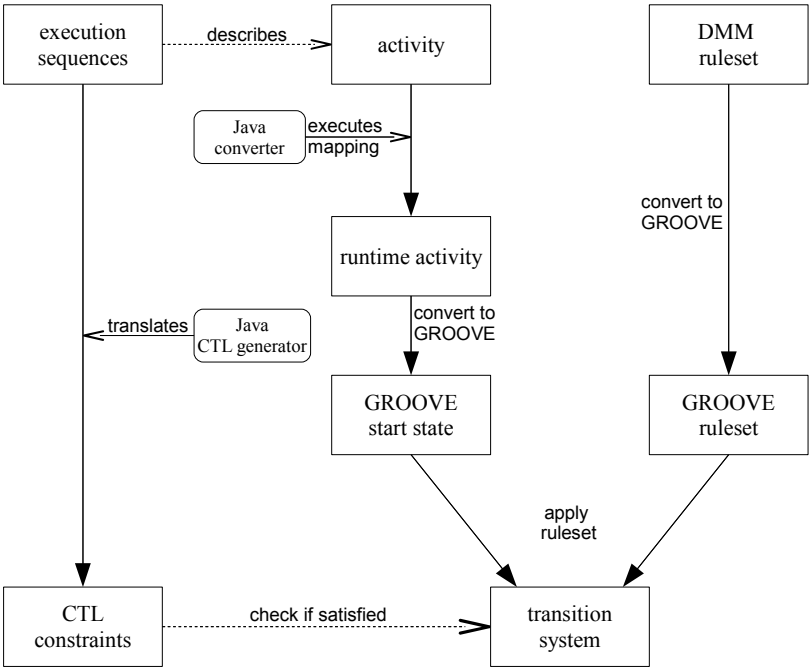

Figure 5.2: Overview of test procedure

the transition system that results from applying the DMM ruleset for UML2 activities on the activity. In the following the steps of the test process are explained in detail.

5.1.1 Traces of Action Execution Events

As mentioned earlier, in a test the activity comes with a corresponding description of its action execution sequences. Such a description is put in a text file in a special format that was elaborated for this thesis. A description of action executions consists of the actual action names of the activity in the correct execution order, split by ->. So *a->b* stands for action "b" is executed after action "a"; this can be seen in the upper example of figure 5.3.
If there are alternative paths in the activity that can be taken, due to control nodes, a new trace has to be specified in a new line. Whenever actions are arranged parallel using fork and join, there are a lot of possible traces. To provide some comfort to the creator of such trace descriptions, all actions that are arranged parallel can be put into square brackets, i.e. *[a,b]* stands for the parallel arrangement of action a and b. This case is shown in the second example of figure 5.3. Whenever actions are arranged parallel, but in these parallel parts some of the actions are sequentially ordered to another, this has once again to be marked by using ->. Therefore *[a->b,d]* as seen in the last example of figure 5.3 stands for action "d" being parallel arranged to action "a" and "b".

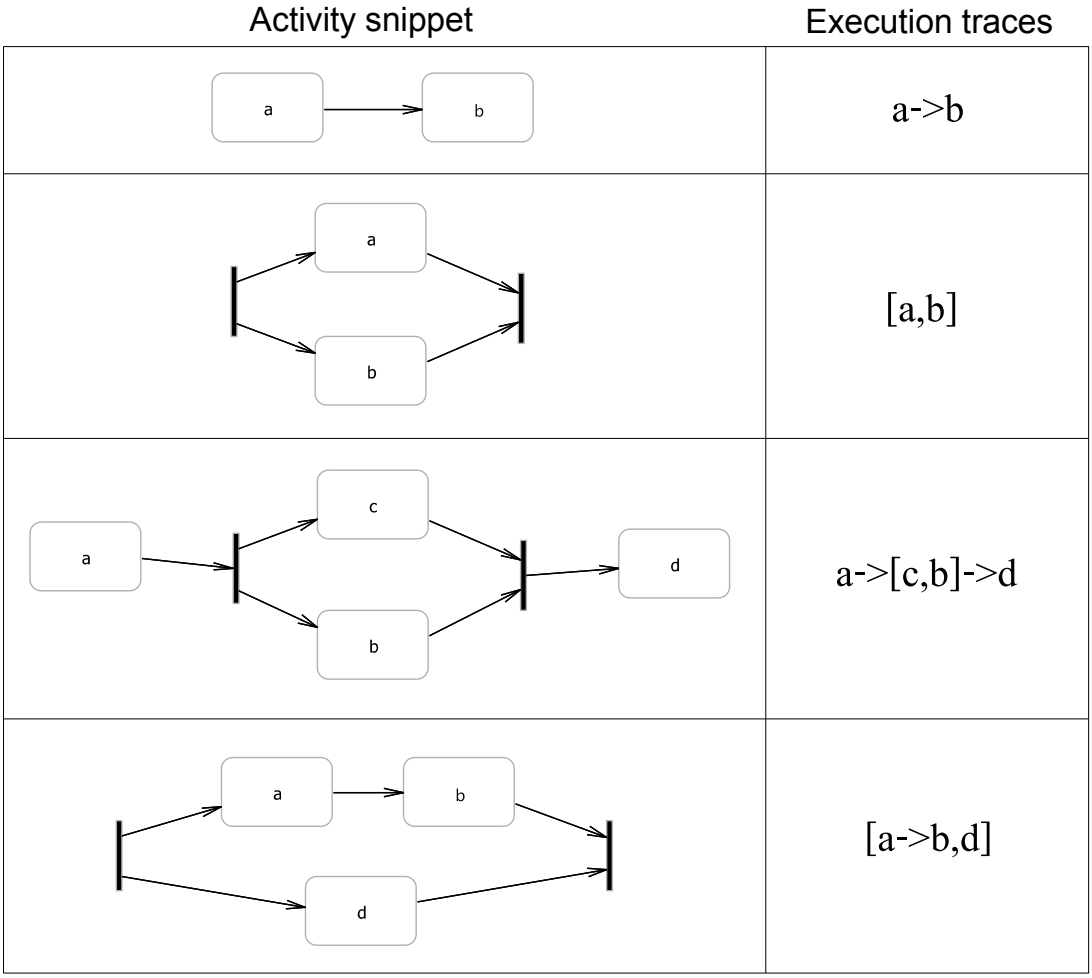

Figure 5.3: Traces of action executions in an activity

5.1.2 CTL Generator

To generate temporal logic formulas that can be used to check the expected behavior against the transition system. A small tool was written in Java shown as "CTL generator" in figure 5.2, that parses such a text file and creates corresponding temporal formulas for all traces found. This is done by using the temporal logic dialect CTL (KM08), that can express using the formula $EF(r)$, whether there exists one path in the transition system where a rule named r is applied. For the execution of actions the ruleset contains the rules named action.start()# explained in section 4.4.3. To distinguish the execution of different actions – which is necessary for the correct testing of execution sequences – DMM provides a feature to put attributes of nodes into parameters of a rule in the transition system. We can put the name attribute of the action node in the rule into a parameter, therefore the following formula can check if a path exists, where the action named "a" is executed.

$$EF(action.start()\#(\text{"}a\text{"}))$$

To check a complete trace the following formula P_1 can be used:

$$P_1 := EF(action.start()\#(\text{"}a\text{"}) \wedge EF(action.start()\#(\text{"}b\text{"})))$$

This checks whether there is a path to the execution of action a, from where once again exists a path to the execution of action b. Such a formula is build of every trace found in the description using an recursive algorithm to handle the correct bracketing. For every parallel description part marked by square brackets all alternative traces are calculated beforehand, i.e. *[a,b]* results in the traces *a->b* and *b->a*. All these traces are checked individually in case one of them fails the tester knows exactly what expected trace is violated by the transition system. After that all these traces are tested at the same time using the following construct:

$$AF(P_1 \vee P_2)$$

The above CTL formula evaluates to true, if for all traces through the transition system, either P_1 or P_2 hold. So whenever there exist different traces in the transition system, meaning that it contains unexpected behavior, the verification will fail.

5.1.3 Creating and Checking a Transition System using GROOVE

In the first step the converter is used to execute the mapping on the activity and a runtime activity is received. On this runtime activity the DMM ruleset can be applied using the GROOVE frame work (Ren09). Therefore the ruleset and the runtime activity are converted to the internal GROOVE format. After that using the GROOVE toolset to apply the ruleset, a transition system is generated that describes the complete behavior of the model.

Every state of the transition system is an actual state of the runtime activity: The start state of the transition system is also the runtime activity in its start state. Every rule of the DMM specification that can be applied leads to a new state. Every new state is different from another, for example a new execution is created or the location of offers and tokens have changed. The transition to such a new state is labeled with the applied rules name. For every reached state, the application process starts again until there are no rules found that match and therefore no new states are created. Such a transition system describes the complete behavior of a given activity, which is used for the actual testing. GROOVE has a build in model checker for computation tree logic (CTL) formulas (KM08). This model checker is used through a Java API to validate the generated CTL formulas on a transition system.

5.2 Examples

For the last section of this chapter two examples are given to show the testing of activities. The first example is a simple activity containing parallel arranged actions to illustrate the techniques described in section 5.1.2. The second one is already known running example activity from section 2.4 that describes proceedings of a web application for streaming video.

5.2.1 Simple Fork Example

In figure 5.4 an activity is seen that has one inital node leading to an action named "a". This action has an outgoing control flow to a fork node, after that two actions named "b" and "c" are arranged parallel. Further downstream the control flow is synchronized by a join node again leading to another action named "d". Finally after "d" an activity final node is reached.

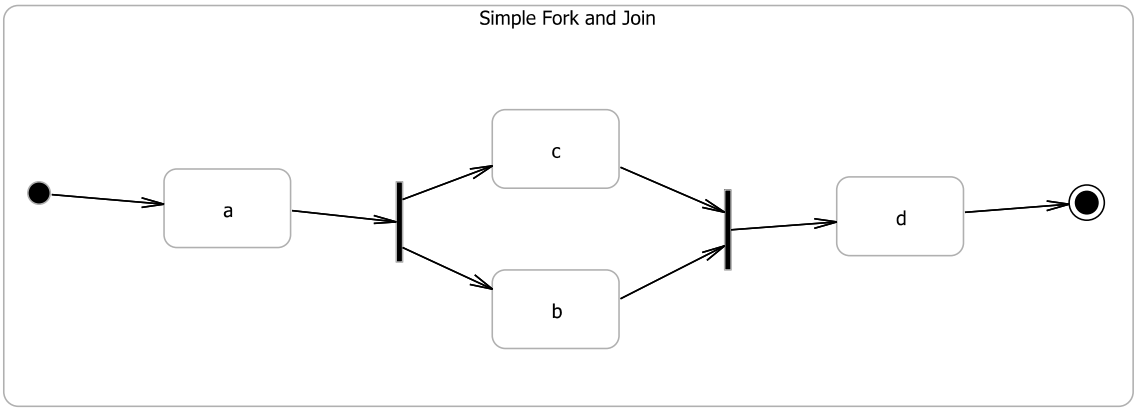

Figure 5.4: Simple fork activity

The description of the action execution traces is already shown in section 5.1.1.

a->[b,c]->d

The CTL generator parses this and creates two traces of this description, because of the parallel part. The first is *a->b->c->d* and the other *a->c->b->d*. For these traces the corresponding CTL formulas are created. Two formulas to check whether paths with these execution sequences exists and another one to check if no other exection paths exist as explained in section 5.1.2. If these formulas are tested against the transition system they all evaluate to true.
If another trace is given to the converter, that represents behavior that is not expected, like *a->d->b->c* the corresponding CTL formula evaluates to false.

5.2.2 Running Example

The example activity shown in section 2.4 can of course also be used to test the created ruleset. The description of the traces for it is a little tricky. This is due to the fact, that "show advertising" is parallel arranged to all other actions, whereas "show streaming video" can only be executed after "user buys viewing time". The two actions from the invoked activity "show 1 minute of video" and "withdraw 1 minute of user account" are arranged sequentially and of course can only be executed after the invoking call behavior action has started its execution. Therefore the resulting description of the traces is *[user buys viewing time->show streaming video->show 1 minute of video->withdraw 1 minute of user account,show advertising,]*.
If the corresponding CTL formulas are tested against the transition system, they all evaluate to true.

6 Conclusion and Outlook

This chapter gives an overview of all the things done for this thesis: The main part is the development of a DMM specification for UML2 activities, but also the test framework to ensure the quality of this work is worth mentioning. It is stated to what extent the goals described in 1.2 were achieved, which problems remain, and what aspects of this work could be improved in the future.

6.1 Conclusion

The main goal in this thesis was the creation of a DMM specification for UML2 activities. With the DMM editor and tool support that was available, we took up the work of Jan Hendrik Hausmann's case study (Hau05) again and tried to cover more details and elements found in the UML2 specification. A DMM specification consists of two counterparts: the runtime metamodel and the DMM ruleset as seen in section 3.1.
The first part, the runtime metamodel for UML2 activities, was elaborated and finally created as an extension of the UML2 ecore model of Eclipse. This could be used in the DMM editor for the DMM ruleset to be typed over. The runtime metamodel extends the original metamodel with classes needed for the description of activity semantics, i.e. classes for action execution and the flow of tokens. The details of this runtime metamodel are explained in section 4.2.
To execute the mapping between the original metamodel and the runtime metamodel, a converter was written in Java. Executing this mapping is a means to automatically get an instance of the runtime meta model – in our case the so called runtime activity – from a given activity. This allowed the user to automatically apply the ruleset to UML2 activities. In section 4.3 the mapping and its execution is described in detail.
After this runtime metamodel for activities was created, the work on the DMM ruleset began. In the beginning the focus was set on supporting very simple activities and we stayed close to the original case study. After a basic activity semantic was working, the specification was reviewed and rules were elaborated to support more details of it.
The second part of the DMM specification, the DMM ruleset was developed with the goal in mind to support more details of the UML2 specification than the case study in (Hau05). So there were a lot of rules added for new elements and most of the existing rules were redesigned, i.e. the rules that specify the semantics for control nodes in order to support weight as it is explained in section 4.4.
To test the created DMM ruleset, an automatic test framework was implemented described in section 5.1. A single testcase consists of an activity and a description of all its possible action execution traces. The ruleset is applied to the activity to receive the resulting transition system, that describes the complete behavior of the activity according to the specified semantics in the ruleset. To test whether this behavior described by the transition system conforms to the expected behavior, we implemented a Java tool to generate temporal logic formulas for all action

executions traces that are given in the description. These can be tested against the transition system to test the ruleset to guarantee an acceptable amount of quality.
All in all, the goals described in section 1.2 were achieved: a DMM specification for UML2 activities was created that covers most of the details found in the UML2 specification. Using the DMM tools, no manual translation into GROOVE for application is necessary. Therefore there is less error–prone in the translation of the ruleset. Including the converter to execute the mapping, it is even possible to automatically apply the ruleset to activities created in Eclipse using the UML2 ecore metamodel. Therefore the ruleset can be used by other tools using that metamodel for validation or execution of activities. We made use of this by building a test framework, to demonstrate the quality of the ruleset shown in section 5.1.

6.2 Outlook

Even though the DMM specification for UML2 activities supports most features found in the UML2 specification and a test framework was implemented for it, there is still room for improvement. Possible future enhancements are listed in the following.

Missing elements The ruleset right now supports only opaque and call behavior actions, so other actions like send signal actions could be supported in a future version, too. All missing elements are listed in the section 4.4.6, where the coverage of the UML2 specification is explained.

String evaluation In terms of automatic execution and testing of activities it would be nice to find a generic method of parsing and evaluating strings in DMM. This would be needed for a real evaluation of value specifications like guards or a joinspec. Right now for every single "string" value specification that is supported in the DMM ruleset, a number of own rules has to be created. This becomes especially difficult whenever alternative choices are described that have to be evaluated, i.e. whether or not an exception really occurred in an action execution. This is the reason why whenever there is an exception handler attached to an action, the body is executed so that the transition system shows this possible path, even though it might never be taken as described in section 4.4.4. This might lead to false negative test results whenever there are transitions on such a path that conflict with the expected behavior.

Mapping Apart from the ruleset, there are other parts of this work that could need improvement. Right now a Java converter is used to execute the mapping, so that a runtime activity can be automatically received from a given activity. To get a runtime metamodel instance from a given model, a more accessible and generic method is advised. Using DMM itself to define this mapping could be a good way to achieve this, because it is easily understood by DMM users.

Testing Another thing that could use improvement is the test framework explained in section 5.1. Right now one test case only tests for the action execution sequences. Other CTL expressions could be found to test the quality of the ruleset through the resulting transition system more profoundly. There is also a problem with the existing CTL expressions. The formula $EF(a \wedge EF(b))$ evaluates to true if b is reached at some point after a, not necessarily directly after a. Therefore if another action is executed between a and b the

formula would also evaluate to true. So the CTL expressions should be enhanced for a stricter check of the action execution sequences.

7 Appendix

CD-ROM containing the following items:

- diploma thesis in pdf format
- DMM ruleset for UML2 activites
- Java converter to execute mapping
- JUnit test framework

List of Figures

Bibliography

[Bal05] BALZERT, H.: *UML 2 kompakt*. Spektrum Akademischer Verlag, 2005

[Bau08] BAUER, Eduard: *Enhancing the Dynamic Meta Modeling Formalism and its Eclipse-based Tool Support with Attributes, Bachelor thesis*. University of Paderborn, 2008

[BH04] BARESI, L. ; HECKEL, R.: *Tutorial introduction to Graph Transformation: A Software Engineering Perspective*. ICGT, S. 431-433, 2004

[Boc03] BOCK, C.: *UML without Pictures*. IEEE Software, 20(5), 2003

[Boc04] BOCK, C.: *UML 2 Activity and Action Models: Object Nodes*. Object Nodes. J. Object Technology available at www.jot.fm, 2004

[Bud03] BUDINSKY, F.: *Eclipse Modelling Framework*. Addison-Wesley Professional, 2003

[CMR+97] CORRADINI, A. ; MONTANARI, U. ; ROSSI, F. ; EHRIG, H. ; HECKEL, R. ; LÖWE, M.: Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In: *Handbook of Graph Grammars*. World Scientific Pub Co, 1997, S. 163–246

[Ecl08] ECLIPSE FOUNDATION: *Graphical Modelling Framework*. http://wiki.eclipse.org/GMF, 25.05.2008

[Ecl09] ECLIPSE FOUNDATION: *The UML2 Project*. http://www.eclipse.org/uml2/, 2009

[EEPT06] EHRIG, H. ; EHRIG, K. ; PRANGE, U. ; TAENTZER, G.: *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag GmbH, 2006

[EHHS00] ENGELS, G. ; HAUSMANN, J. H. ; HECKEL, R. ; SAUER, S.: Dynamic Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: *A. Evans, S. Kent, and B. Selic UML, 2000 - The Unified Modeling Language*. Springer-Verlag Berlin, 2000

[EHK+97] EHRIG, H. ; HECKEL, R. ; KORFF, M. ; LOEWE, M. ; RIBEIRO, L. ; WAGNER, A. ; CORRADINI, A.: Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In: *Handbook of Graph Grammars*. World Scientific Pub Co, 1997, S. 247–312

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[Gro05] GROUP, Object M.: *OMG XML Metadata Interchange (XMI) 2.1 Specification*. http://www.omg.org/cgi-bin/doc?formal/2005-09-01, 2005

[Hau05] HAUSMANN, J.H.: *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Techniques*, Universität Paderborn, Diss., 2005

[Hec06] HECKEL, R.: *Graph transformation in a nutshell*. Electronic Notes in Theoretical Computer Science 148 (1 SPEC. ISS.), S. 187-198, 2006

[HS05] HAUSMANN, J. H. ; STÖRRLE, H.: *Towards a Formal Semantics of UML 2.0 Activities*. Software Engineering 2005, P-64:117-128, 2005

[Hus06] HUSSEY, K.: *Getting Started with UML2*. www.eclipse.org/uml2/, 2006

[Kec06] KECHER, C.: *UML 2.0 Das umfassende Handbuch*. Galileo Press, 2006

[KM08] KRÖGER, Fred ; MERZ, Stephan: *Temporal Logic and State Systems*. Springer Berlin, 2008

[Löw93] LÖWE, M.: *Algebraic approach to single-pushout graph transformation*. Theor. Comput. Sci. 109, 1993

[Meh84] MEHLHORN, K.: *Data Structures and Algorithms 2. Graph Algorithms and NP Completeness*. Springer-Verlag GmbH, 1984

[Obj04] OBJECT MANAGEMENT GROUP: *Meta Object Facility (MOF) 2.0 Core Specification*. http://www.omg.org/cgi-bin/doc?ptc/2004-10-15, 2004

[Obj09] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language Superstructure*. http://www.omg.org/cgi-bin/doc?formal/09-02-02, 2009

[Ren04] RENSINK, A.: The GROOVE Simulator: A Tool for State Space Generation. In: *Applications of Graph Transformations with Industrial Relevance (AGTIVE) volume 3062 of Lecture Notes in Computer Science*. Springer-Verlag GmbH, 2004, S. 479–485

[Ren09] RENSINK, A.: *GROOVE: A Graph Transformation Tool Set for the Simulation and Analysis of Graph Grammars*. http://groove.sourceforge.net, 2009

[Röh08] RÖHS, Malte: *A Visual Editor for Semantics Specifications Using the Eclipse Graphical Modeling Framework, Bachelor thesis*. University of Paderborn, 2008

[Rhe08] RHEKER, Thomas: *A Bidirectional Transformation between EMF Models and Typed Graphs, Bachelor thesis*. University of Paderborn, 2008

[Roz97] ROZENBERG, G.: *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997

[SE09] SOLTENBORN, C. ; ENGELS, G.: Towards Test-Driven Semantics Specification / University of Paderborn. 2009. – Forschungsbericht

[TR99] TAENTZER, G. ; RUDOLF, M.: *Introduction to the Language Concepts of Agg*. TU Berlin, 1999

[Wes05] WESTPHAL, Frank: *Testgetriebene Entwicklung mit JUnit & FIT*. dpunkt, 2005

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig und nur unter Zuhilfenahme der angegebenen Quellen und Hilfsmittel angefertigt habe. Alle wörtlich oder inhaltlich zitierten Stellen sind als solche kenntlich gemacht.

Paderborn, den 1. September 2009