Integration of Object-oriented Behaviour-modelling Techniques

Holger Lewin

4th April 2007

Abstract: In this diploma thesis I will define the complex modelling technique CUML that contains the behaviour-modelling techniques CActivities and CNSDs. Integration of these techniques is achieved through model transformation to a common semantical domain. Both model transformation and the semantical domain language L3 are defined in this thesis. Although the definition of the Unified Modeling Language (UML) lacks formal semantics, there are many approaches to formalize semantics for the single sublanguages of the UML by mapping to a semantical domain. Nevertheless there is no approach defining formal semantics to a set of UML sublanguages that would allow complete modelling of object-oriented software systems. Such an approach would require the semantical integration of different behaviour modelling techniques.

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den / Unterschrift

Contents

1 Introduction

With the Unified Modeling Language (UML) as an accumulation of the most popular semi-formal visual languages for specifying object-oriented software systems, the present day software engineer has a wide variety of behaviour-modelling techniques to choose from. In the context of Model Driven Architecture (MDA), however, it is no longer sufficient to model a system's behaviour from different views i.e. with different types of diagrams. The more complex a system is and the larger the number of persons or different enterprises sharing the modelling process, the more important becomes the integration of the different views of the model, with the objectives of model checking, making statements about model properties and last but not least automated code generation. This requires a common formal semantics for all modelling techniques used in the model.

The UML Definition [6] lacks a complete formal semantics. The following two approaches could fill this gap: First, the informal and intuitive semantics defined in [6] could be enhanced to be formal. Sadly, the current condition the UML is in seems to make this impossible: The UML Superstructure is full of the so called "semantic variation points" that leave the precise semantics of many concepts to the implementation platform. Further, it is impossible to formalize the intuitive semantics of UML Activities inherited from Petri Nets: In [18] it is shown, that there is no form of Petri Nets that is able to cover all the features of UML Activities yet. A second approach is to define formal semantics for essential views of the UML, as described in [11]: Formal semantics are defined for a bunch of UML sublanguages that together allow a complete specification of a system (e.g. Class Diagrams and Statecharts) and form the 'essential' UML. Diagrams of other kinds can be seen as projections out of the essential part of a model. Nevertheless it would be necessary to define a common formal semantics for the UML sublanguages contained in the essential part.

This need to integrate the essential views in order to specifiy common formal semantics gives rise to the concept of a complex modelling technique introduced in [8]. This complex modelling technique consists of techniques for structure modelling, and constructive and descriptive behaviour modelling.

My diploma thesis deals with the integration of the constructive behaviour modelling techniques of a complex modelling technique on the syntax level. First of all, a complex modelling technique has to be defined. [8] proposes CUML as such a technique, that contains UML Activities and Nassi-Shneiderman Diagrams as constructive behaviour modelling techniques. Since CUML is only introduced informally in [8], a formal syntactical definition has to be given first. To take up the idea of an essential UML (see above), I will define CUML as a UML subset by a UML Profile in Chapter 2. This subset contains UML Class Diagrams, UML Actions and UML Activities. Since Nassi-

Shneiderman Diagrams are not part of the UML, I will define them as a concrete syntax for a subset of UML Activities.

The integration of the constructive CUML techniques CActivities and CNSDs is achieved by transformation of CUML models to a semantical domain language. In Chapter 3, I will motivate and define the Low-Level-Language L3, that has been introduced informally as a semantical domain for CUML in [8].

My definition of the model transformation from CUML to L3 is the issue of Chapter 4.

1.1 MOF and Graph Transformation

The Meta-Object Facility (MOF) is a standard of the Object Management Group (OMG) for Model Driven Architecture (MDA). Since the languages CUML and L3 are defined using MOF, I will give a short introduction here.

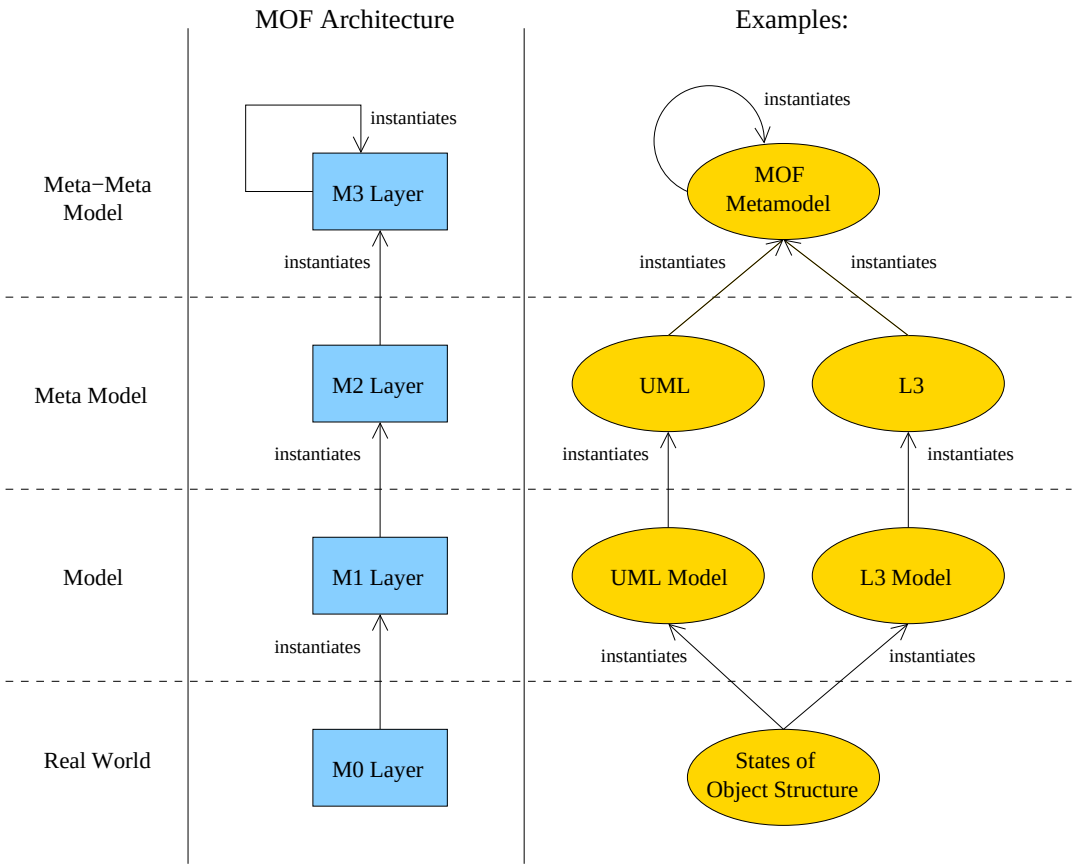

Figure 1.1: MOF: 4-layer Architecture

MOF provides a metamodelling architecture consisting of the four layers of models

depicted in Figure 1.1.

The MOF Metamodel at the M3 layer is actually a meta-meta model describing itself, which is indicated by the "instantiates"-arrow from and to the M3 icon. This M3 model is used as a language to describe meta models at the M2 layer as its instances. Examples of M2 models are the UML and the language L3 introduced in this thesis. The models that instantiate the M2 models are e.g. a UML model or an L3 model at the M1 layer that describe e.g. an object-oriented software system. An execution of this system at runtime is a model placed at the M0 layer.

The intended purpose of MDA is to simplify software development processes and improve the quality of software. This is achieved by formally defined models and automated model transformation. The MOF-defined modelling languages - the meta models at the M2 layer - allow the development of such formally defined models[1]. Transformations of these models are defined at the M2 layer. The models at the M3 layer can be understood as the input for such a model transformation.

The most popular approaches for MOF model-to-model transformation are the graph transformation-approach and relational approaches based on the OMG's relational Query/View/Transformation (QVT) standard. A comparative study of a graph transformation-based and a relational approach is shown in [13].

For the definition of the model transformation from CUML to L3, I will use typed attributed graph transformation. In this approach, the transformation rules refer directly to instance specifications of the meta models of CUML and L3, and define the transformation in an operational way, which is more convenient for the documentary application in Section 4.2 than the relational approach. typed attributed graph gransformation of MOF-based models is formalized in [7].

I will give an intuitive introduction to typed attributed graph transformation by the little example below. For a formal introduction and deeper insight in the theoretical foundation of graph transformation consult [10].

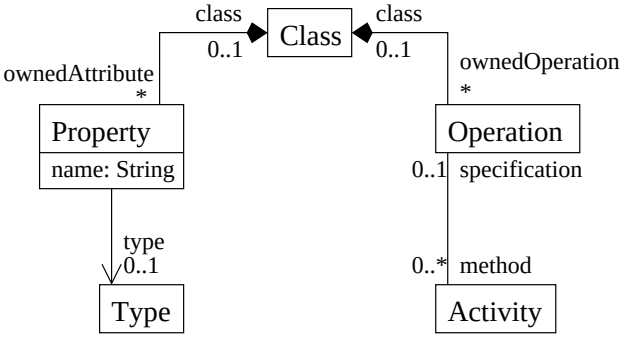

Figure 1.2: Abstract Syntax: CUML Classes

The rule example in Figure 1.3 is defined on the abstract syntax of Class Diagrams

[1]Whereby only the syntax of these models is formally defined in the case of UML.

(see Figure 1.2). Graph transformation rules are visualized as consisting of three parts: A left-hand-side (LHS), a right-hand-side (RHS) and an arbitrary number of negative application conditions (NACs). The object structure of the RHS is the qualification of rule application. The rule is executed on every part of the model that matches with this pattern. So in the context of the example rule, every instance of *Operation* named **init** that is owned by an instance of *Class* is replaced by an instance of *Operation* named similar to the class instance.

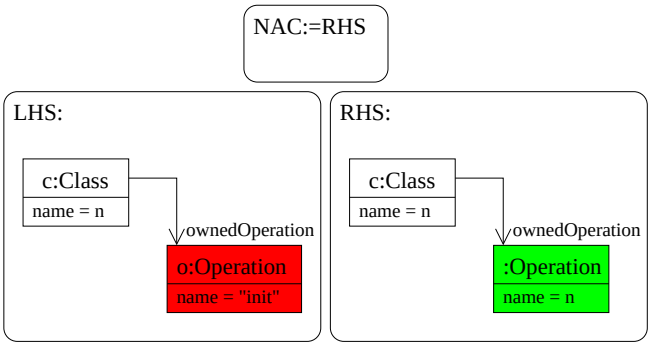

Figure 1.3: Rule Example

The object structure of the RHS indicates that *o:Operation* is deleted from the model, because there is no object with the identifier *o*. Deleted objects are depicted red, which has no formal meaning. The object *c:Class* appears in both the LHS and the RHS and is therefore preserved by the rule. Objects appearing only in the RHS are created, like the operation named **init** in the example. Objects created by the rule are depicted green, which is no formalism either. NACs are object structures that can prevent the rule from being applied on a certain match found in the model. The NAC of the example rule - which consists of the same object structure as the RHS - indicates, that the rule will not be applied for a certain class, if there is already an operation with the same name as the class.

1.2 UML Profiles and Stereotypes

The UML Superstructure [6] contains the profiling mechanism to adapt UML metaclasses to different purposes, which includes the ability to tailor the UML meta model to different platforms. A popular example is the adaption of UML components to Enterprise Java Beans. Such adaptions are grouped in a so-called UML profile.

CUML is a restricted form of UML and can therefore be defined as a UML profile. This profile consists of constraints that are expressed on the UML metaclasses, and of stereotypes that extend the UML metaclasses. Therefore I will give a short introduction to the UML profiling mechanism, concentrating on stereotypes.

In a UML profile, the metaclasses of the UML Superstructure can be extended by use of the UML metaclass *Stereotype*, which is a specialization of *Class* itself[2]. Therefore a stereotype can have properties, that are referred to as tagged values. I will describe the profiling mechanism by use of the CUML stereotype *Argument*.

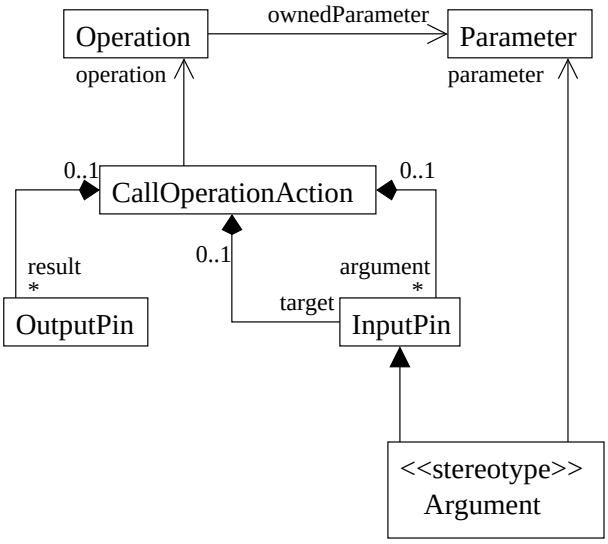

Figure 1.4: Abstract Syntax: CallOperationAction

The abstract syntax in Figure 1.4 shows that the stereotype *Argument* extends the UML metaclass *InputPin*. The extension relation is depicted as an arrow pointing from the stereotype to the metaclass with a filled arrowhead. If the extension would be annotated with the keyword "required", every instance of *InputPin* in a model that uses the profile had to be extended by an instance of the stereotype *Argument*.

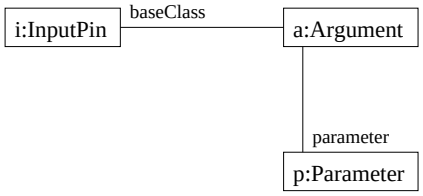

Figure 1.5: Instance Specification: Extended InputPin

The instance specification in Figure 1.5 shows an instance of *InputPin* extended by an instance of *Argument*: The extension relation of the profile results in a link be-

[2]Consult [6] for the abstract syntax of the UML metaclasses contained in the package *Profiles*.

tween the stereotype instance and the metaclass instance, whereas the metaclass instance is available to the stereotype instance at the *baseClass* link end. The property *Argument::parameter* results in the tagged value identifying an instance of *Parameter*.

The stereotypes and constraints of the UML profile, that are introduced during the discussion of CUML in Chapter 2, are summarized in the appendix in Section 6.2.

1.3 Running Example

I will use a running example for the introduction of the concrete syntax of CUML techniques. A data model for finite automata will serve as example for a structure model (see Figure 2.4). Furthermore I will model simple operations on this structure via the CUML behaviour modelling techniques introduced in Chapter 2 to form an example CUML model including CClassDiagrams, CActivities and CNSDs.

The results of the transformation of the example class diagram and some of the example operation specifications will document the semantical domain language L3 and the model transformation from CUML to L3.

1.4 Referred Language Specifications

The description of CUML in Chapter 2 frequently refers to the UML 2.1 Superstructure [6].

1.5 Legend

Text written in Typewriter typeface refers to an OCL expression:

```
context c inv:
  [OCL]
```

Text written in italics refers to a UML namespace: *Kernel::Classifier*

Text written in boldface refers to a value: **true**

1.6 Tools

This document was created using LaTeX. For editing CUML Class Diagrams, CUML Activity Diagrams, the concrete syntax diagrams of CUML and L3 and the rules of the model transformation, I have used Xfig, which is a drawing application for the X Windows system. For the CNSD diagrams I have used the latex package StrukTeX.

2 CUML - Complex Modelling Technique

The Comprehensive/Compact Unified Modelling Language (CUML) as sketched in [8] is a subset of UML2.0 - if you just consider the abstract syntax of the visual languages contained. CUML consists of a technique for modelling the structure of a system - CUML Class Diagrams -, and techniques to model the behaviour constructively - CUML Activities and CUML Nassi Shneiderman Diagrams (CNSDs), based on the classical Nassi Shneiderman Diagrams, a well-known informal modelling technique.

The need to restrict the UML for the concrete modelling of software systems arises from the fact that the UML has to serve multiple purposes. During the development process of a software system, the behaviour model can be placed on many different levels of abstraction. With the UML it is possible to express the state of the behaviour model on each of that levels. Sometimes some aspects of the model are more important than others: with different kinds of diagrams the UML is able to give different views on the model. CUML however is designed to specify an object oriented software system on the lowest level of abstraction with the objective to be able to generate executables for certain platforms. Therefore comprehensive information about the model on the most concrete level is required in a CUML model, what leads to the restriction to Class Diagrams and Activities in CUML.

CNSDs are formally introduced to CUML as concrete syntax for a restriction of UML Activities. Considering CNSDs one might ask why not use the UML directly instead of introducing new visual languages for behaviour modelling by tracing them back on UML concepts? Of course the CNSDs introduced in this chapter are no enrichment of the UML, but a way to encapsulate constructs of UML Activities that are used frequently. UML Activities use a token concept for modelling data flow, similar to the token flow of Petri Nets. Especially this informal token semantics is a disadvantage of the UML Activities. Besides the semantical problems already stated in the introduction in Chapter 1, exact modelling of data flow leads to the following problems: The activity diagrams contain a lot of elements that manage token flow (object nodes, object flow edges) and obstruct the view on the main purpose, i.e. the action organization. Further most of the modelers are used to think data flow in terms of program code, i.e. access to variables or attributes, which is very different from the token concept in UML Activities. CNSDs are designed to solve these two problems: By encapsulating the data flow in the abstract syntax, data flow can be modeled by variable access, so the modeler has not to deal with token flow in the concrete syntax, which concentrates on the action organization. This makes it a lot easier to edit the CActivities, as will be seen from the examples.

In the first section of this chapter I will give a short description of CUML Class Diagrams. Although the structural part of CUML is out of the scope of my thesis, I will need the structure to rely on when introducing CUML Activities in Section 2.2. Section

2.3 deals with the CUML Nassi-Shneiderman diagrams as an alternative concrete syntax for CUML Activities. CUML is described in this chapter by abstract syntax diagrams of the UML containing the CUML stereotypes. The structure of the abstract syntax of UML is flattened, i.e. features of abstract superclasses are pulled down to their concrete subclasses. This might increase the readability for the reader who is not familiar with the UML metaclass hierarchy. The restrictions of UML to CUML are described by OCL Constraints. The UML stereotypes and OCL constraints in this chapter form the UML Profile for CUML that can be found in the appendix in Chapter 6.

2.1 CUML Class Diagrams

In order to model the dynamic part of an object-oriented software system, a model of the static structure is required. Hence I will give a short introduction to CUML class diagrams first. Since my thesis deals with behaviour modelling in the first place, the version of CUML Class Diagrams introduced here only serve the purpose of assigning properties and operations to a certain class. A more sophisticated handling of the issue considering such features as inheritance and polymorphy is out of scope and left for future work.

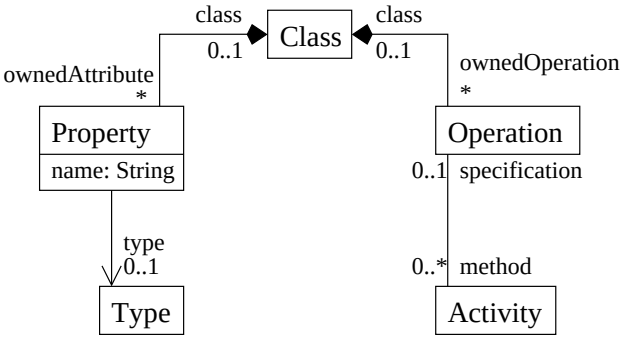

Figure 2.1: Abstract Syntax: CUML Classes

Figure 2.1 shows the abstract syntax of CUML Class Diagrams. Since there are no Interfaces in a CUML model, every *Operation* or *Property* has to be owned by a certain *Class*. The behaviour of an *Operation* has to be specified by a CUML Activity. This leads to the following constraints in the UML Profile:

- Every instance of *Classifier* is instance of *Class*.

```
context Classifier inv:
 self.allInstances()->forAll(oclIsTypeOf(Class))
```

- Every *Operation* must be owned by a class.

```
context Operation inv:
```

```
self.class->size() = 1
```

- Every *Property* must be owned by a class.

```
context Property inv:
 self.class->size() = 1
```

- There is exactly one *Activity* for every particular pairing of an implementing *Class* and an *Operation*.

```
context Communications::Operation:
inv:  self.class->size() = 1 implies
self.method->select(m | m.oclIsTypeOf(Activity))->size() = 1
```

- There are no autonomous activities, i.e. every activity is associated with a classifier as its context.

```
context StructuredActivities::Activity inv:
  self.context->size() = 1
```

According to the idea of CUML as a comprehensive subset of the UML, we require maximum significance of the modeled structure: as a matter of course, every *Class* referred to in the model has to be given, as well as every *Property* or *Operation* of these.

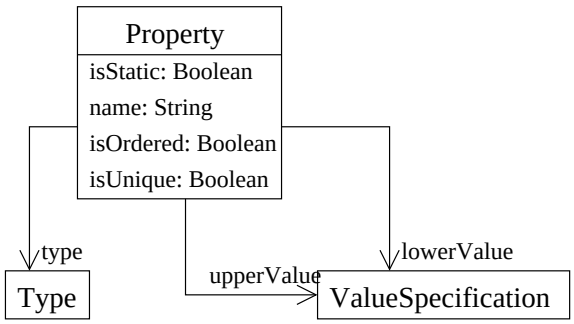

Figure 2.2: Abstract Syntax: Property

The lower and upper bounds of a *MultiplicityElement* (such as a *Property* or *Parameter* of an *Operation*, see Figure 2.2) have to be expressed as an integer and unlimited natural respectively, so specification of these bounds through expressions that could only be evaluated to runtime can be avoided.

- Lower and upper bound of a *MultiplicityElement* have to be expressed as integer and unlimited natural respectively.

```
context Kernel::MultiplicityElement inv:
 self.lowerValue.oclIsTypeOf(LiteralInteger) and
 self.upperValue.oclIsTypeOf(LiteralUnlimitedNatural)
```

The UML contains the primitive data type *UnlimitedNatural* to provide a way to express an unlimited number designated by the character '*' in the concrete syntax. To avoid the appearance of the type *UnlimitedNatural* where it is not necessary i.e. where the information can be expressed using the type *Integer*, I restrict the use of *UnlimitedNatural* to the following points:

- Instances of *LiteralUnlimitedNatural* are only allowed as specification of a *MultiplicityElement*s upper bound, the capacity of an *ObjectNode* or the weight of an *ActivityEdge*.

```
context LiteralUnlimitedNatural inv:

 self.allInstances()->forAll(n |
 MultiplicityElement.allInstances()->exists(upperValue = n) or
 ActivityEdge.allInstances()->exists(weight = n) or
 ObjectNode.allInstances()->exists(upperBound = n))
```

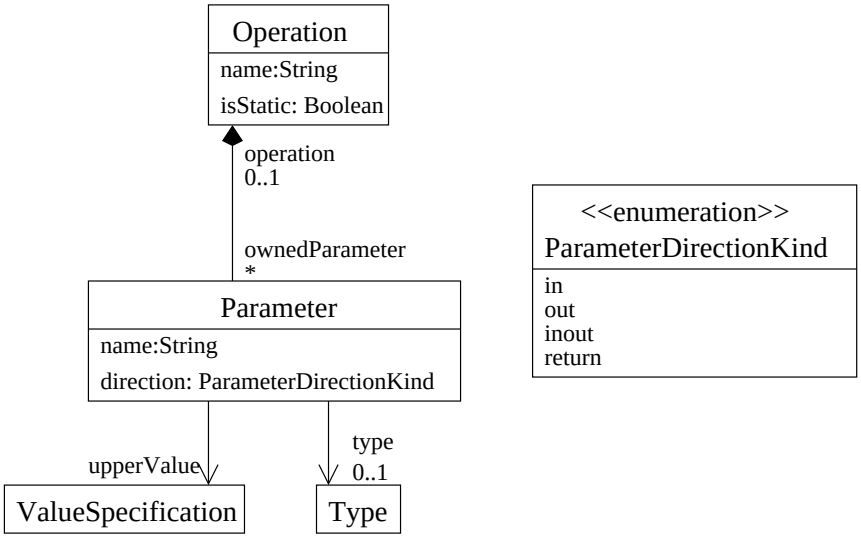

Figure 2.3: Abstract Syntax: Operation

As can be seen from the abstract Syntax of *Operation* in Figure 2.3, an operation owns a number of Parameters. In CUML, it is required that a parameter is owned by an operation, since there are no other behavioural features in CUML than operations.

Another interesting point is the enumeration type *ParameterDirectionKind*: it provides the enumeration literals **in**, **out**, **inout** and **return**, and [6] tells us about the semantics of *ParameterDirectionKind*, that a parameter can be passed in, out or in and out of a behavioural element. If the behavioural element is an operation, it can have one return parameter. This is a rather poor semantical statement in my opinion. I would expect

that different kinds of parameters are used to indicate how the values, that are passed to or from the behaviour by use of a parameter, are handled within the behaviour. Further the semantics of *ParameterDirectionKind* in the context of an operation being the parameterized behaviour is even more unclear.

For CUML, I propose the following semantics of *ParameterDirectionKind*: **out** parameters do not occur in CUML, since the only parameterized behaviours are operations and therefore **return** parameters the only way to pass an object or value out of a behavioural element. An **in** parameter indicates that the object passed to the operation must not be changed during operation execution, i.e. the operation has no side-effects on this parameter. Accordingly, an **inout** parameter indicates side-effects on a parameter.

- Every *Parameter* is owned by an *Operation*.

```
context Parameter inv:
  self.allInstances().operation->size = 1
```

- *ParameterDirectionKind* is restricted to: **in**, **inout** and **return**.

```
context Parameter inv:
  self.direction = in or self.direction = inout or
  self.direction = return
```

As the abstract syntax diagrams show, *TypedElement*s like *Property* or *Parameter* do not have to be typed. In CUML, however, typing of *TypedElement*s is required, what leads to another constraint:

- Every *TypedElement* has a type.

```
context TypedElement inv:
  self.allInstances().type->size = 1 or
  self.oclIsTypeOf(InputPin)
```

Some features of the UML require typelessness of typed elements (e.g. *TestIdentityAction* in 2.2.2). To work around this, CUML assumes a common super class for all classes that are not given any explicit super class in the model. This is common to modern object-oriented programming languages like Java, and according to those the CUML root class is called *Object*. All the Constraints above are part of the UML Profile for CUML: see 6.2.1.

Figure 2.4 shows the class diagram of the running example. There is one package, named *automaton*, that contains all classes - *Automaton, State, Transition, IO*. All classes except *IO* have constructor operations that are, needless to say, static operations, which is indicated by the underlining of the operations. All operations are modelled completely, i.e. with all their parameters and return type. An automaton consists of states and transitions, and therefore the class *Automaton* is associated to the classes *State* and *Transition* via composition associations. This special kind of association indicates, that,

to use the running example, instances of *Automaton* are responsible for the lifecycle of *State*- and *Transition*-objects, i.e. these are created and deleted only by *Automaton*-objects. The example diagram contains a «use»-dependency to indicate that within the class *Automaton* the class *IO* is used (e.g. calls of *IO*-operations, access to *IO*-attributes).

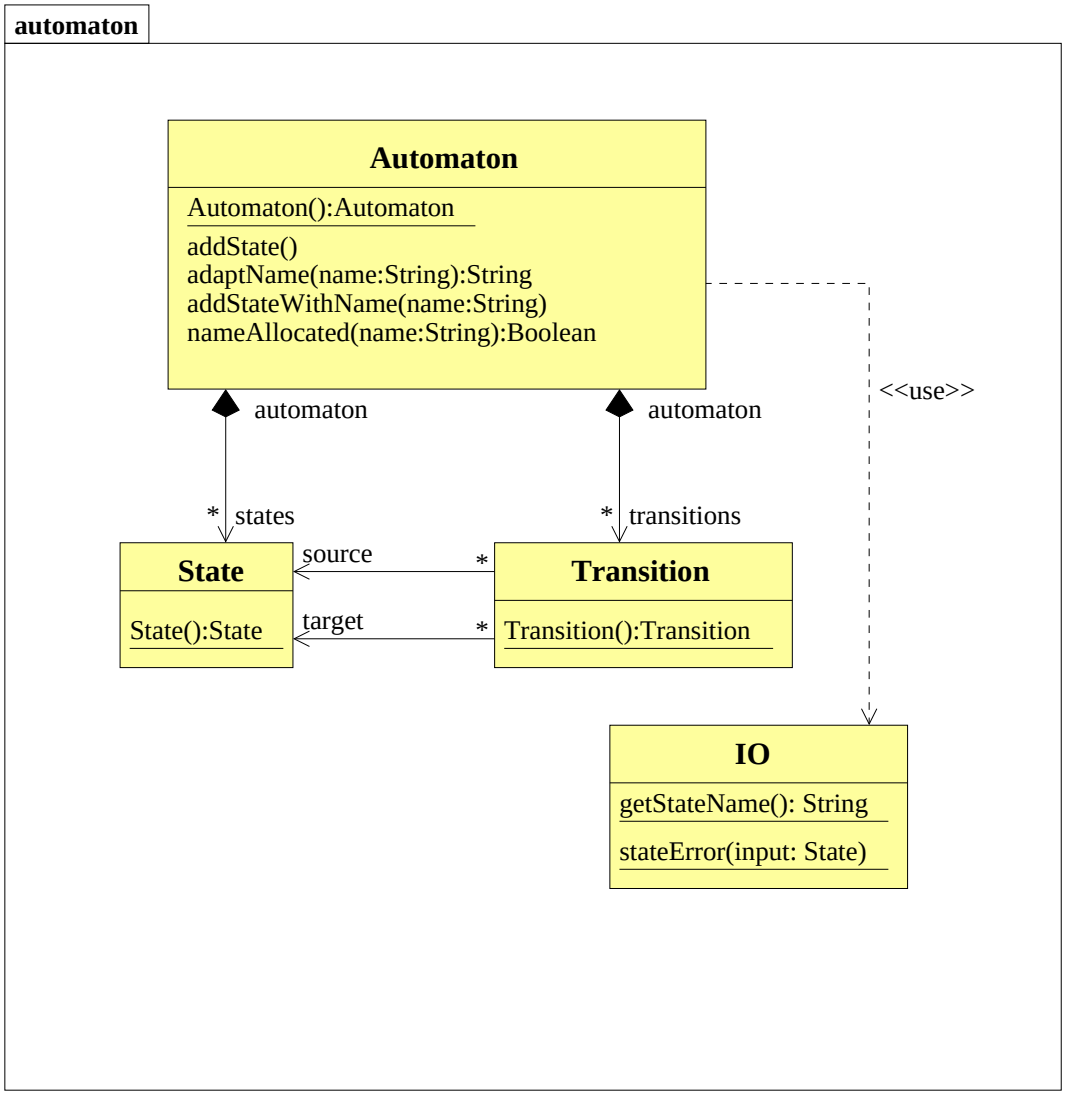

Figure 2.4: CClassDiagram: automaton.

Since the structural part of CUML is out of scope, CUML Class Diagrams are introduced in a basic version here. A more sophisticated version should feature interfaces, abstract classes and features, visibility etc.

2.2 CUML Activities

In CUML, Activities are the core technique for modelling behaviour. In a CUML model the responsibility of a classifier for its owned behaviour - i.e. what methods have to be implemented by a class - is defined in the structural part by CUML ClassDiagrams. The behaviour of every concrete method declared in the structural part of the modelling technique via CClass diagrams has to be associated with exactly one CActivity in the constructive part [1].So the technique for modeling the methods of a certain class should focus on the flow of data/objects and control. Therefore Activities are the technique of choice.

CUML Activities (CActivities) are a true subset of the UML Activities. This is due to a number of requirements that have to be fullfilled in the special application area of CUML. As stated before, CActivities will be used to model the behaviour of an object-oriented software system on the most concrete level of abstraction, so only some of the concepts of UML Activities are useful.

In this section I will give a short introduction to the abstract and concrete syntax and the semantics of the features of UML Activities used in CUML Activities.

The semantics of UML Activities, like of every other kind of UML diagram, contain the so-called 'Semantic Variation Points': The precise semantic is often left to the implementation platform, since the UML is placed on a more general stage of abstraction. Nevertheless would it be necessary to show the conformity of the semantics of UML and the formal semantics of CUML once it is constituted.

The behavioural semantics of UML Activities that is part of the UML Superstructure [6] is partly inherited from Petri Nets. The most simple Petri Nets consist of a directed graph of places and transitions as nodes. A behavioural semantics is achieved by the flow of so-called tokens along the edges. The token can stand for the focus of control, or, in more sophisticated versions of Petri Nets, for data. So it is possible to model the flow of control and data by use of token flow. Accordingly, there are two kinds of token in UML: control token and data token. The rules by which the token flow in UML Activities is defined are introduced below, but I will not give a detailed description, the more so as the formal semantics of Petri Nets has been adopted in an informal and intuitive way, so that the semantics of UML Activities is far from being well-defined at all: While a mapping from UML Activities to Colored Petri Nets can be given for a basic version of UML Activities (see [17]), there is no version of Petri Nets that could serve as a semantical domain for the complete UML Activities yet (see [18]).

Anyway, the semantics of UML is out of the scope of my thesis and so I will give only a slight overview of the semantics of UML Activities. For a deeper insight consult [6].

Possibly because some of the features of UML Activities were introduced to UML for reengineering purposes, i.e. to allow creating an UML model from existing source code, there is no notation specification for these features in the UML Standard [6]. For these features including *VariableAction*, *StructuralFeatureAction*, *TestIdentityAction* et al. I

[1] Note that an *Operation*, if owned by an *Interface*, possibly will be implemented by more than one method.

will give my own notation. Furthermore the colouring of the CActivities are neither part of the UML standard nor of CUML.

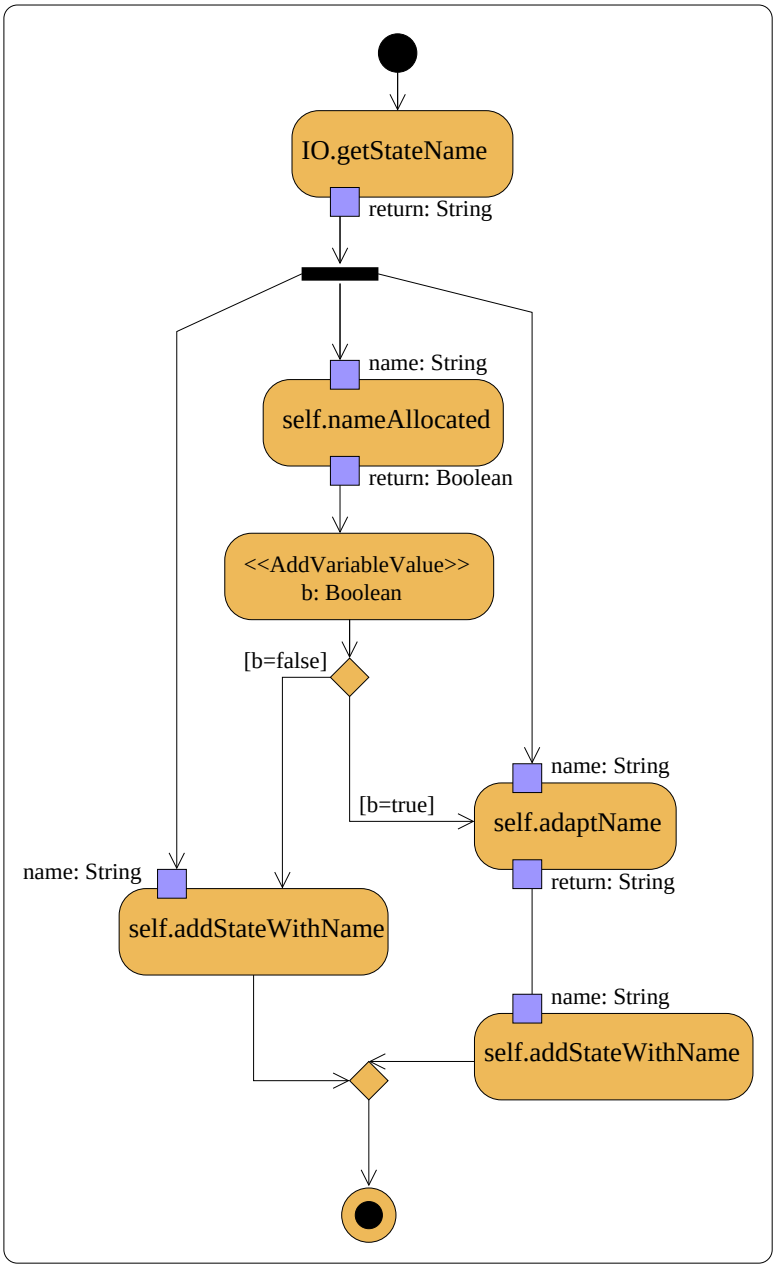

Figure 2.5: CActivity Diagram: Automaton.addState.

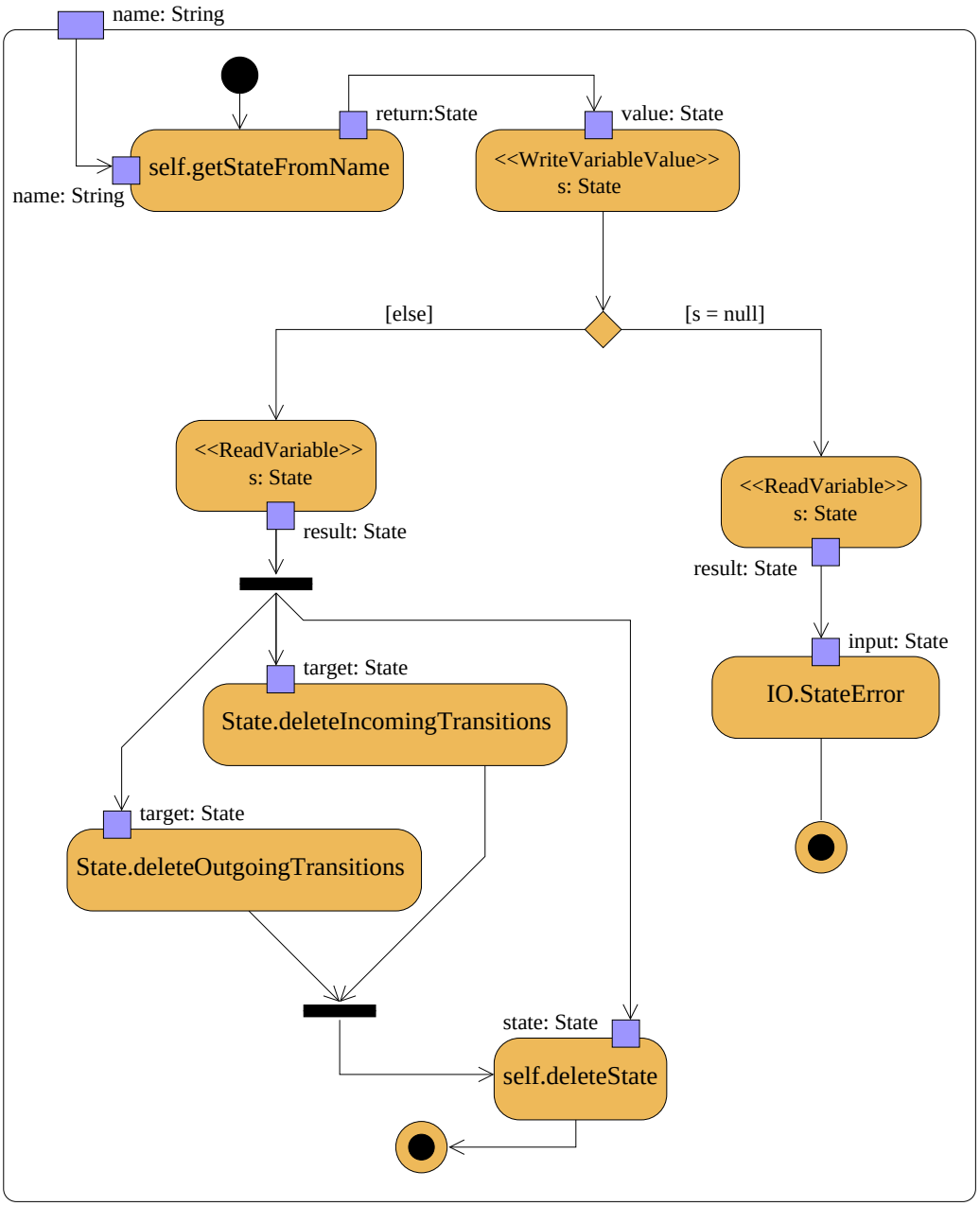

Figure 2.6: CActivity Diagram: Automaton.deleteState.

The CActivity Diagrams in Figures 2.5 and 2.6 show the behaviour modelling of two operations from the CClass Diagram (2.4): *Automaton.addState* und *Automaton.deleteState*. In the following short introduction to CActivities I will explain the elements contained in the diagrams.

2.2.1 Activity

An *Activity* mainly consists of *Action*s, *ControlFlow* to organize the execution of action, and *ObjectFlow*s to model the flow of objects and data from and to actions and in and out of the Activities.

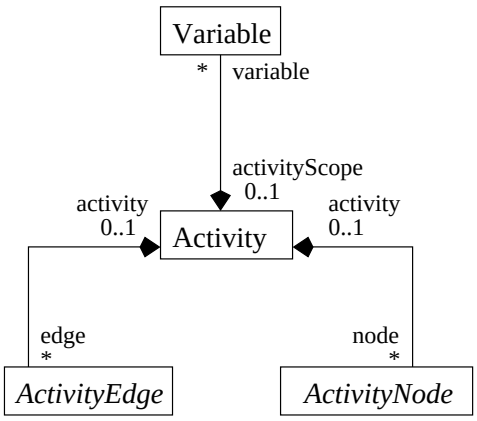

Figure 2.7: Abstract Syntax: Activity

The abstract syntax in Figure 2.7 shows that an *Activity* has *Variables*, *ActivityEdges* and *ActivityNodes*. *ActivityEdges* and *ActivityNodes* can also be owned by a *SequenceNode*, which I will introduce later.

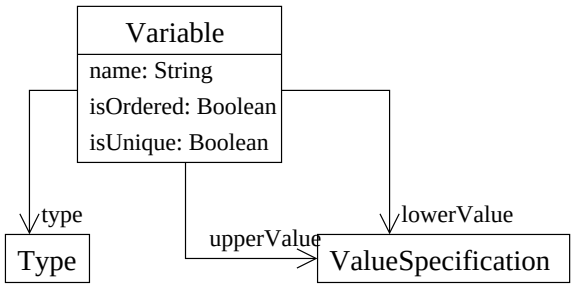

Figure 2.8: Abstract Syntax: Activity Variable

The associated *Activity* of a *Variable* is its scope, so for CUML I require every *Variable* to be owned by an *Activity*.

The abstract syntax in Figure 2.9 shows the graph-like structure formed by activity nodes and edges. An edge is a connection between two activity nodes along which tokens flow. An edge is weighted to specify the maximum number of tokens that can flow along the edge on each traversal (default weight: **1**). Activity edges are associated with

an instance of *ValueSpecification* as guard (default: **true**). The guard expression must evaluate to **true** for every token to pass along the edge.

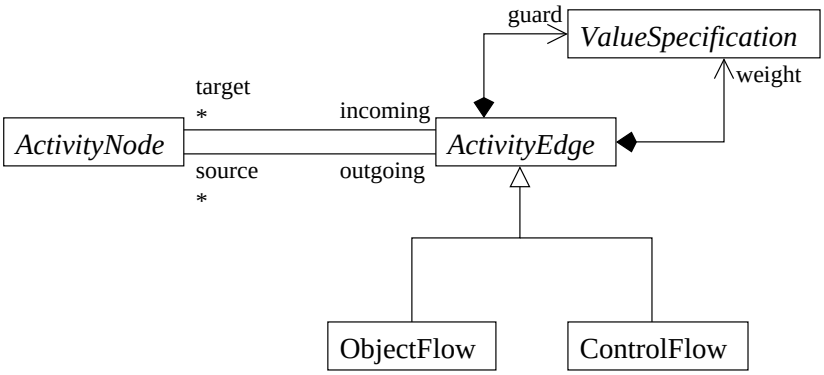

Figure 2.9: Abstract Syntax: Flows

A *ControlFlow* is an activity edge along which only control tokens flow. A *ControlFlow* may not have object nodes at either end, except for object nodes with the predefined control type.

An *ObjectFlow* can have data or object tokens passing along it. Source and target have to be object nodes.

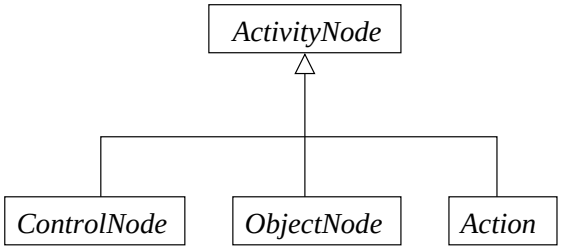

Figure 2.10: Abstract Syntax: Nodes

As the abstract syntax diagram in Figure 2.10 shows, the nodes are specialized as *ControlNodes*, *ObjectNodes* and *Actions*.

CActivities contain control structures from UML Activities on the level of *Intermediate-Activities*. This level includes modeling concurrent data and control flow and branching of data and control flow. Figure 2.11 shows the hierarchy of control nodes in the abstract syntax.

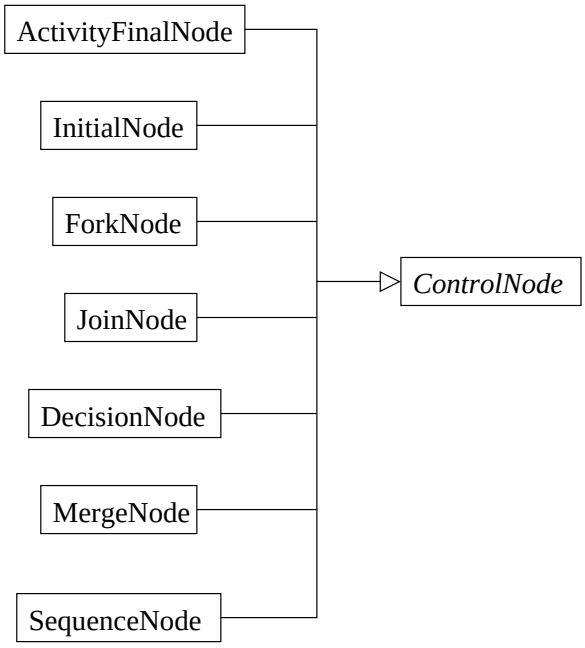

Figure 2.11: Abstract Syntax: ControlNodes

Execution of a CActivity starts with a control token placed on every *InitialNode* contained by the CActivity directly [2].

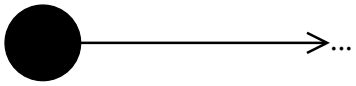

Figure 2.12: Concrete Syntax: InitialNode

If a token reaches an *ActivityFinalNode*, the CActivity is terminated. All executing actions are terminated and the contents of output parameters are passed to the caller. All other tokens in the CActivity are destroyed. Every *ActivityParameterNode* that corresponds to a return parameter is provided a so called null token, which is an object token that contains no object.

[2] A *StructuredActivityNode* like *SequenceNode* can also contain *InitialNode*s

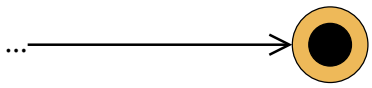

Figure 2.13: Concrete Syntax: ActivityFinalNode

DecisionNode allows branching of flows via case distinction. The outgoing edges are associated with instances of *ValueSpecification* as guards. If the guard of an edge evaluates to **true**, the token is offered to the edge. The guard expressions do not have to be disjoint or complete.

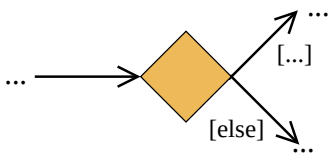

Figure 2.14: Concrete Syntax: DecisionNode

MergeNode brings together alternate flows. Tokens from any of the incoming edges are offered to the outgoing edge.

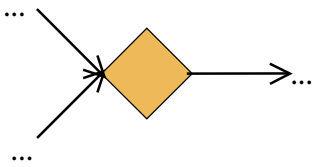

Figure 2.15: Concrete Syntax: MergeNode

A *ForkNode* is used for concurrent flows. The token arriving at a *ForkNode* is duplicated for every outgoing edge.

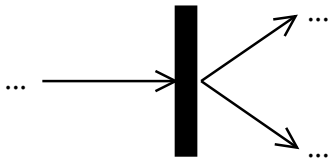

Figure 2.16: Concrete Syntax: ForkNode

JoinNode is used to synchronize concurrent flows. If a token is offered to all incoming edges, a token is offered to the outgoing edge.

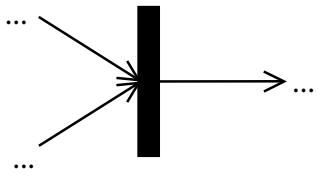

Figure 2.17: Concrete Syntax: JoinNode

A sequence node is a container for activity nodes. The contained nodes are executed according to the flow relation. When a sequence node starts executing, a control token is offered to all nodes without predecessor.

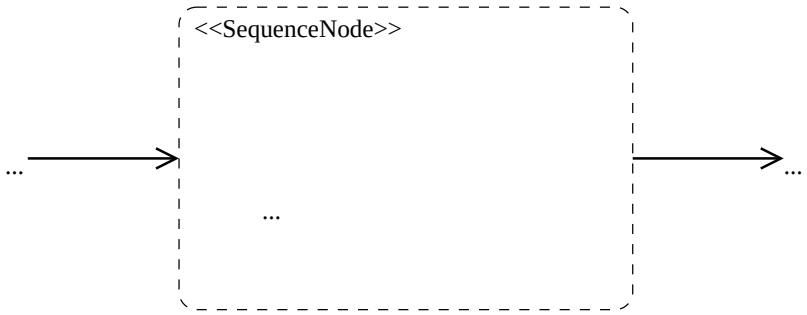

Figure 2.18: Concrete Syntax: SequenceNode

Instances of *SequenceNode* are used to encapsulate parts of an CUML Activity if CUML Nassi-Shneiderman Diagrams are used as the concrete syntax.

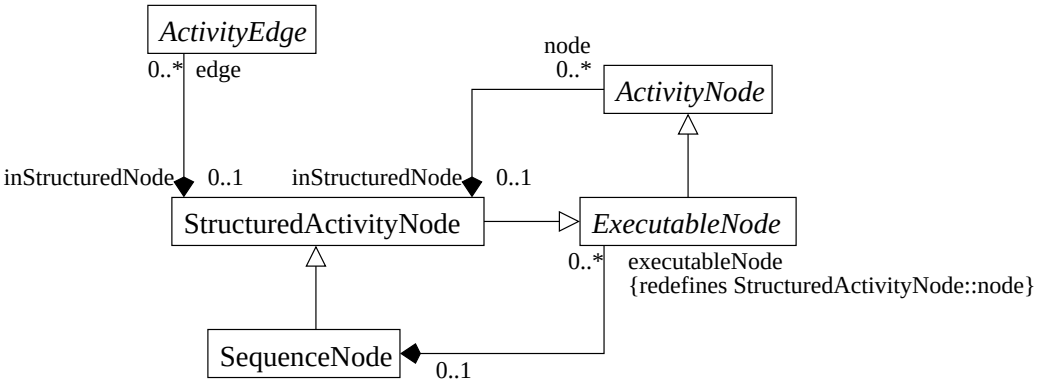

Figure 2.19: Abstract Syntax: Sequence Node

As the abstract syntax in Figure 2.19 shows, a sequence node can contain activity nodes and activity edges. In the concrete syntax these are the nodes and edges that are drawn inside the rectangle notation of the sequence node. Further, the nodes linked to the sequence node via *executableNode* link ends are executed, when the sequence node itself starts execution. Since in CUML I use instances of *SequenceNode* only to encapsulate abstract activity fragments that correspond to CUML Nassi-Shneiderman statements in the concrete syntax, the multiplicity of *executableNode* link ends is restricted to '1' in the UML Profile [3]:

- Every *SequenceNode* has one *ExecutableNode* of type *InitialNode*.

```
context SequenceNode inv:

 self.executableNode->size() = 1 and
 self.executableNode.oclIstTypeOf(InitialNode)
```

Figure 2.20 shows the concrete subclasses of the abstract class *ObjectNode*. An object node can contain token conforming to the type of the node. An upper bound specifies the maximum number of tokens allowed in an object node (default: unlimited). An object node can have more than one outgoing edge, nevertheless a token can only traverse one of the outgoing edges.

[3]CUML Nassi-Shneiderman diagrams are introduced as concrete syntax for CUML Activities in 2.3

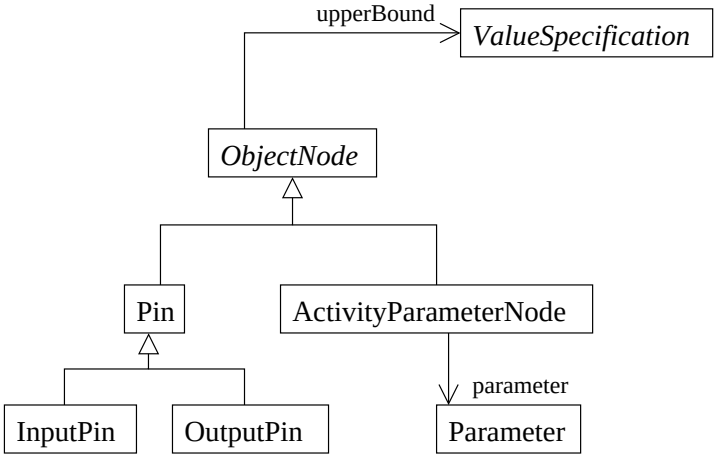

Figure 2.20: Abstract Syntax: ObjectNodes

A pin is an object node that provides input or output to actions. An action that represents the execution of parameterized behaviour has one pin for every parameter. Action execution can only start if all input pins contain the required number of object or data tokens. The required number of tokens for actions executing operations is indicated by the multiplicity specified for the parameter of that operation. For actions that read or write attributes or variables, the number of required tokens depends on the multiplicity of the attribute or variable respectively.

An instance of *ActivityParameterNode* provides input and output to activities. If the activity specifies the implementation of an operation, number and type of the parameter nodes have to correspond to the parameters of the associated operation.

2.2.2 Actions

An action represents the execution of behaviour. An action is atomic from the point of view of the activity containing it, although the executed behaviour may be complex. Pins provide input and output to actions that represent the execution of parameterized behaviour. There are actions that represent predefined behaviour (e.g. *TestIdentity-Action*) and actions that can execute behaviour that is part of the model itself (e.g. *CallOperationAction*).

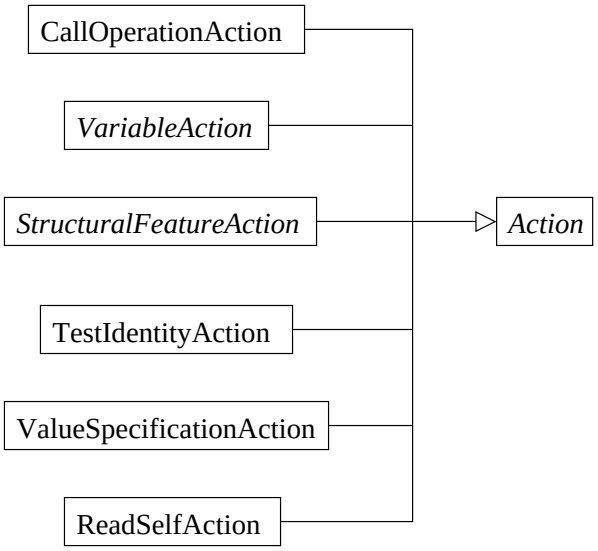

Figure 2.21: Abstract Syntax: Actions

An action execution starts, when all input data is provided via input pins or the action is activated via control flow. After the execution all output data is placed on the output pins of the action. For a detailed description of the behavioural semantics of UML Activities inherited from Petri Nets consult [6].

An instance of *CallOperationAction* invokes the execution of a method. Its input pins correspond to the operation's parameters. A special target input pin takes in the object the method is called on. After the execution, a token containing the result of the method is placed on a return pin.

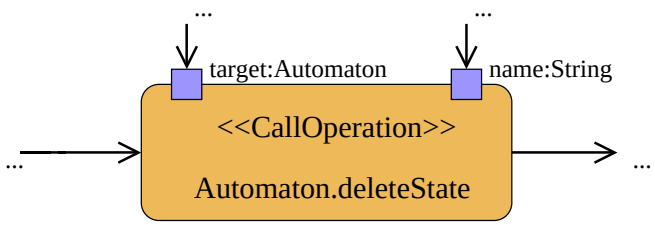

Figure 2.22: Concrete Syntax: CallOperationAction

The abstract syntax in Figure 2.23 shows that instances of *CallOperationAction* are linked to an operation. The pins of the action are not explicitly linked to the parameter for that they are expected to provide the values. Instead the UML Superstructure requires the pins of the action to conform to the parameters in type, multiplicity and ordering by natural language constraints. To make the connection explicit in the UML Profile for

CUML, I introduce a stereotype *Argument* that extends the UML metaclass *InputPin* and is associated with the metaclass *Parameter*. The parameter connected to the extended pin has the same type and multiplicity as the pin itself, what is ensured by OCL constraints:

«stereotype» Argument

- **Metaclass** *Pin*
- **Description** If the pin is owned by a *CallOperationAction*, it has to correspond to a parameter of the operation.
- **Tagged Values**

  parameter: Parameter The parameter the pin provides the values for.

- **Constraints**

  [1] The associated parameter has the same type as the pin.

  ```
  context Argument inv:
    self.baseClass.type = self.parameter.type
  ```

  [2] The associated parameter has the same multiplicity as the pin.

  ```
  context Argument inv:
    self.baseClass.multiplicity =
    self.parameter.multiplicity
  ```

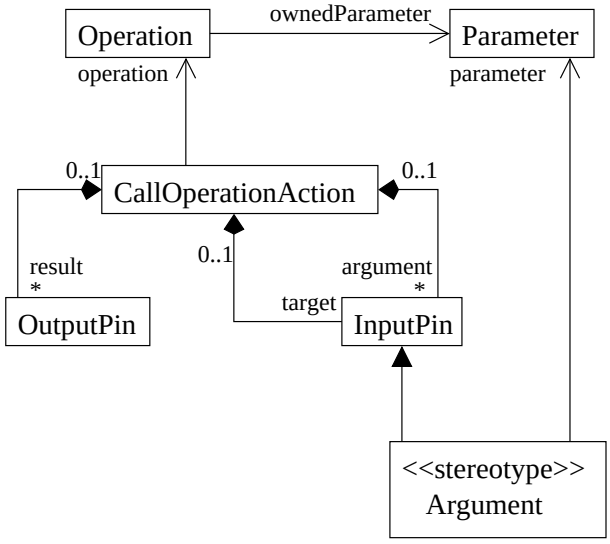

Figure 2.23: Abstract Syntax: CallOperationAction

StructuralFeatureAction is an abstract class for access to structural features of a classifier such as an attribute or association end. Execution requires an object token as input identifying the object which feature will be accessed. Concrete subclasses are: *AddStructuralFeatureValueAction* for write access, *ReadStructuralFeatureAction* for read access and *RemoveStructuralFeatureValueAction* for removing values from the structural feature.

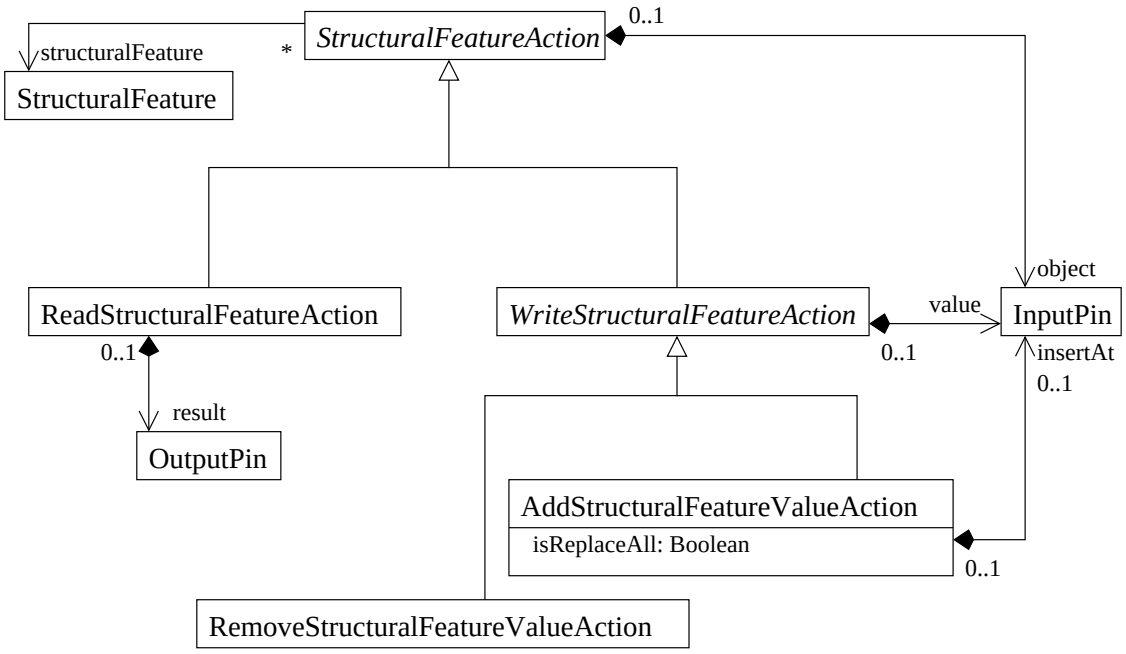

Figure 2.24: Abstract Syntax: StructuralFeatureAction

Figure 2.24 shows how the action is linked to a structural feature. The strucutural feature is required to be static in [6]. For an instance of *WriteStructuralFeatureAction* the new value to assign is provided by an input pin. The UML Superstructure seems to be inconsistent in the point of the type of the pin that provides the new value. In the description it says: "It has an input pin on which the value that will be added is put." But the correponding constraint requires the same type for the pin and the classifier owning the structural feature. I assume this a bug and that the type of the pin at the *value* link end has to have the same type as the structural feature. Multiplicity of the pin has to be **1**. If the structural feature is ordered, the insertion point has to be provided by the input pin at *AddStructuralFeatureValueAction::insertAt*. Surprisingly, [6] defines no way to access a value of an ordered structural feature at a certain index. If *AddStructuralFeatureValueAction::isReplaceAll* is true, all values of the structural feature are deleted before the new value is assigned. For further semantical details consult [6].

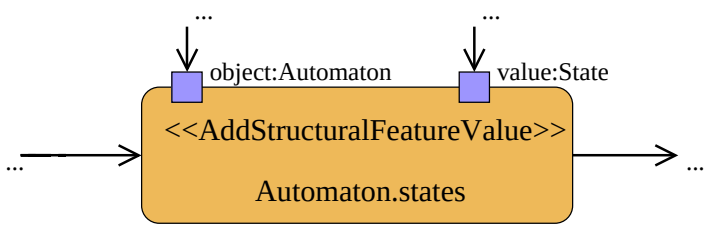

Figure 2.25: Concrete Syntax: AddStructuralFeatureValueAction

Figure 2.25: Since [6] does not contain any specific notation, the kind of the action is just indicated by the name of the UML metaclass in brackets.

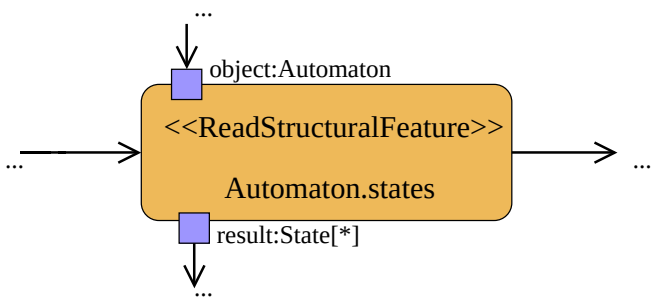

Figure 2.26: Concrete Syntax: ReadStructuralFeatureAction

Figure 2.26: The result pin is of the same type of the structural feature. Its multiplicity has to be the same as the feature's, or, if the feature is single-valued, can be multi-valued. [6] claims this to be helpful in the case the multiplicity of the structural feature changes, because the action model will not be affected. However, the token semantics of an activity highly depends on the upperBound specifications of object nodes, so such decisions have to be made carefully.

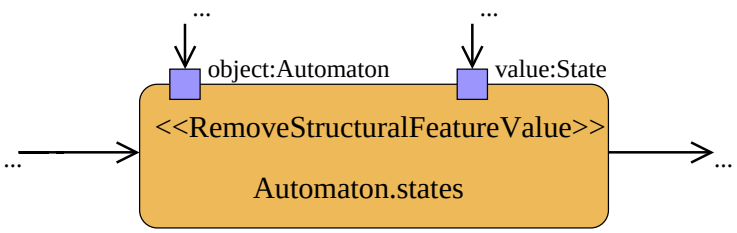

Figure 2.27: Concrete Syntax: RemoveStructuralFeatureValueAction

VariableAction is an abstract class for access to variables of an Activity. Concrete

subclasses are: *AddVariableValueAction* for write access, *ReadVariableAction* for read access and *RemoveVariableValueAction* for removing values from the variable.

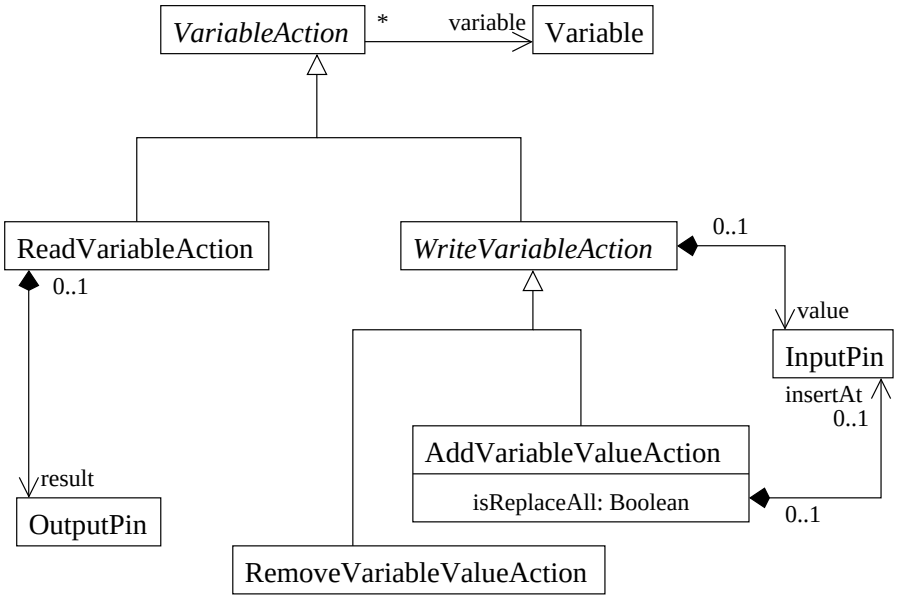

Figure 2.28: Abstract Syntax: VariableAction

The abstract syntax of variable actions in Figure 2.28 is very similar to the abstract syntax of structural feature actions, except from the fact that variable actions are linked to a variable instead of a structural feature. Further a variable action does not need a target object.

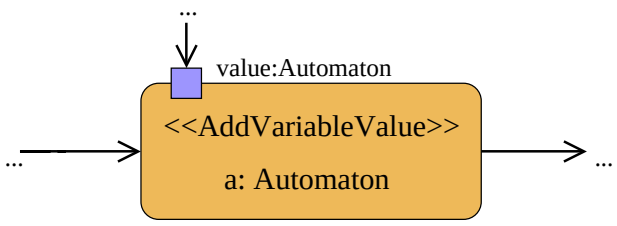

Figure 2.29: Concrete Syntax: AddVariableValueAction

According to [6] a variable has to be owned by either an activity or a structured node (e.g. sequence node). Since in the Low-Level-Language L3 there is no such concept as a structured node, the UML profile for CUML requires a variable to be owned by an activity, to avoid name conflicts:

- A *Variable* must be owned by an *Activity*.

```
context Variable inv:

  self.activityScope->oclIsTypeOf(Activity)
```

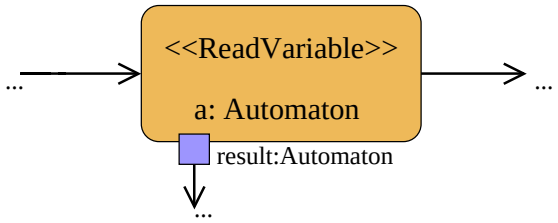

Figure 2.30: Concrete Syntax: ReadVariableAction

A variable is only accessible within the activity that owns it. The concept of variables within activities was introduced to UML for reengineering purposes, i.e. creating an activity model from existing program source code. While source code variables that are only written once can be translated to object flow easily, [6] proposes a complex mechanism for the general case, including wrapper objects for variables und use of the metaclass *DataStoreNode*, which was originally designed for data base storage. This shows the weakness of the token concept, because it would cause an activity to contain a lot of functional behavior concerning the data flow, whereas the UML views should abstract from that as far as possible and concentrate on the business behaviour of an application.

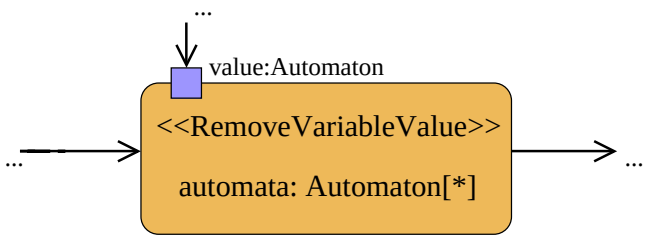

Figure 2.31: Concrete Syntax: RemoveVariableValueAction

An instance of *ValueSpecificationAction* can be used to evaluate a given value specification. After execution, a token containing the result of the evaluation is placed on the result pin.

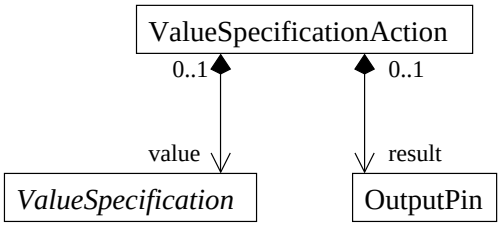

Figure 2.32: Abstract Syntax: ValueSpecificationAction

In the abstract syntax in Figure 2.32, *ValueSpecificationAction* is linked to *ValueSpecification*. The value specification is either a literal of one of the primitive data types of the UML (Boolean, UnlimitedNatural, Integer, String), or it is an expression that can be evaluated at runtime to a certain value. A description of CUML expressions follows in Section 2.2.3.

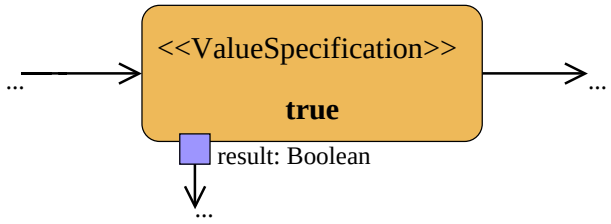

Figure 2.33: Concrete Syntax: ValueSpecificationAction

In Activities two objects can be compared by a *TestIdentityAction*. If the two object tokens consumed contain the same object, a token containing the **true** value is placed on the result pin. Otherwise the result is **false**.

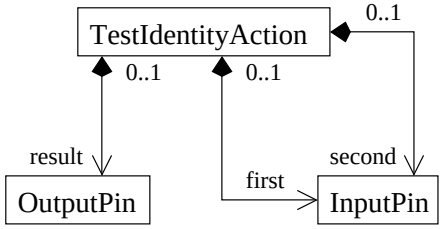

Figure 2.34: Abstract Syntax: TestIdentityAction

The abstract syntax in Figure 2.34 shows how the input objects for the comparison are provided by input pins. Multiplicity of these pins is '[1..1]' and [6] requires them to

have no type. Since for type safety CUML requires every typed element to have a type, in CUML the input pins have the CUML root type *Object*. This way the case can be avoided that a token containing a value of primitve type is placed on the input pins: [6] defines no semantics for this case.

- The input pins of a *TestIdentityAction* have the type *Object*.

```
context TestIdentityAction inv:

  self.first.type = Object and
  self.second.type = Object
```

The output pin has the type 'Boolean' and multiplicity '[1..1]'. The semantics of *TestIdentityAction* is undefined for primitive values in [6]. In CUML, every primitive value is treated as an object and therefore the comparison will return **true** for the same values and **false** otherwise.

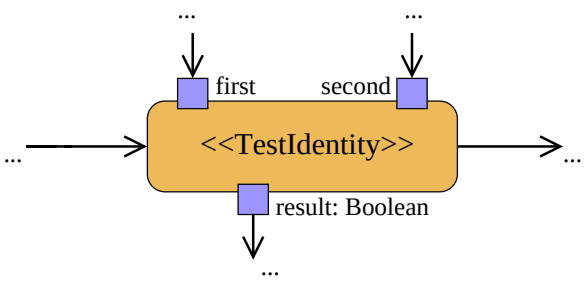

Figure 2.35: Concrete Syntax: TestIdentityAction

An instance of *ReadSelfAction* simply returns the object on which the operation implemented by the activity containing the action is called on (host object). In the case that within an activity an operation is called on the host object, I recommend to indicate that in the concrete syntax by adding the prefix 'self.' to the name of the call action for simplicity. Same for structural feature actions. For an example, see the activity diagram in Figure 2.6.

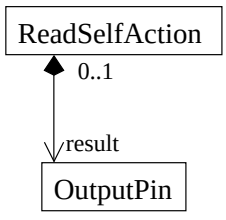

Figure 2.36: Abstract Syntax: ReadSelfAction

The result pin of an instance of *ReadSelfAction* (see abstract syntax in Figure 2.36) has the type of the classifier that is the context of the activity. In CUML, this is the class that owns the activity.

2.2.3 Value Specifications

A value specification can identifiy a value or instance, or can yield a value or instance when evaluated. Instances of the subclasses of *LiteralSpecification* identify a value of one of the primitive types provided by the UML: Boolean, UnlimitedNatural, Integer and String. The abstract syntax of literal specifications in Figure 2.38 shows that there is further the metaclass *LiteralNull* to allow the specification of the absence of a value.

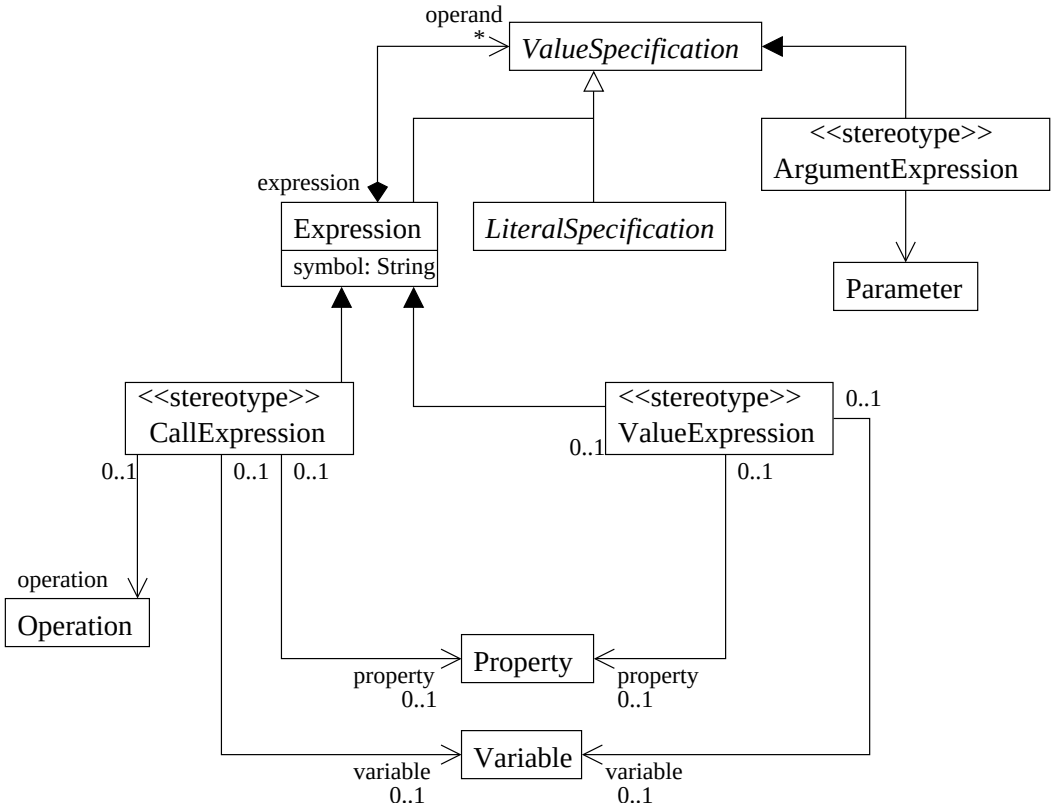

Figure 2.37: Abstract Syntax: ValueSpecification

An instance of *Expression* must be evaluated at runtime to a certain value. [6] decscribes an expression as a "node in an expression tree". In fact the tree structure consists of instances of *ValueSpecification*, as the abstract syntax diagram of value specifications shows (Figure 2.37). Of course the instances of *Expression* are the only ones to have children in this tree structure, identified by the link ends that are instances of the *Expression::operand* association end.

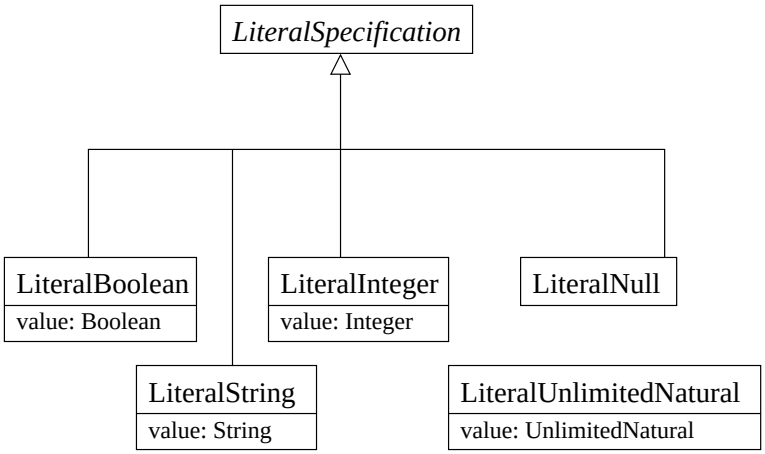

Figure 2.38: Abstract Syntax: LiteralSpecification

The semantical description in [6] seems a bit fuzzy: the only information about an expression is expressed by its string attribute 'symbol'. On how to evaluate or even check consistency of an expression tree, it says: "The interpretation of this symbol depends on the context of this expression.". To achieve a little of syntactical standard for CUML, I propose three kinds of entities in a UML model an expression should be able to refer to within an activity: The structural features of the host classifier, the operations executable on the host classifier and the variables of the activity. To make these relations between the model elements explicit and to avoid string parsing during the model transformation, the UML Profile for CUML contains the following stereotypes:

«stereotype» ArgumentExpression

- **Metaclass** *Expression*
- **Description** If an *Expression* represents an argument of an operation call, it is extended by this stereotype.
- **Tagged Values**

  parameter: Parameter[0..1] — Parameter of the *Operation* of which the expression is an argument.

- **Constraints**

  [1] The associated parameter is parameter of the operation of which this expression is an argument.

  ```
  context ArgumentExpression inv:
    self.operand.operation.parameter->includes(self.parameter)
  ```

An expression, that is the operand of an expression referring to a parameterized element, must be extended by an instance of *ArgumentExpression*. This stereotype is introduced to the UML Profile for CUML to make the relation between the expression operand and the parameter explicit.

«stereotype» CallExpression

- **Metaclass** *Expression*
- **Description** If an *Expression* represents an operation call, it is extended by this stereotype.
- **Tagged Values**

operation: Operation[1]	The Operation which is called.
property: Property[0..1]	Attribute of the enclosing *Operation*s *Class* designating the object the operation is called upon.
variable: Variable[0..1]	*Variable* of the enclosing *CActivity* designating the object the operation is called upon.

- **Constraints**

[1] If the corresponding *Operation* is not static, one of the following Tagged Values is set: parameter, property, variable.

```
context CallExpression inv:
  not(self.operation.isStatic) implies
  (parameter->size() + property->size() + variable->size() = 1)
```

[2] If the corresponding *Operation* is static, none of the following Tagged Values is set: parameter, property, variable.

```
context CallExpression inv:
  self.operation.isStatic implies
  (parameter->size() + property->size() + variable->size() = 0)
```

Any expression that refers to an operation has to be extended by the stereotype *CallExpression*. The stereotype instance identifies the operation instance via the *operation* end and the target object via the *property* end, if the operation is called on a property, or the *variable* end, if the opertion is called on a variable of the enclosing activity.

«stereotype» ValueExpression

- **Metaclass** *Expression*
- **Description** If an *Expression* represents an attribute, parameter or variable, it is extended by this stereotype.

- **Tagged Values**

property: Property[0..1]	Attribute of the enclosing *Operations Class* containing the value.
variable: Variable[0..1]	*Variable* of the enclosing *CActivity* containing the value.

- **Constraints**

[1] One of the following Tagged Values is set: parameter, property, variable.

```
context ValueExpression inv:
 self.operation.isStatic implies
 (parameter->size() + property->size() + variable->size() = 1)
```

An expression referring to a property or a variable has to be extended by the stereotype *ValueExpression*. The stereotype instance identifies the property or variable via the *property* end or the *variable* end respectively.

2.3 CUML Nassi-Shneiderman Diagrams

CActivityDiagrams offer different ways to model behaviour by using control structures common to modern programming languages. These control structures include different forms of case distinction, loops etc. But these features defined for UML have the following disadvantages: Using the structured way to model e.g. iteration over a collection by use of the UML metaclass *StructuredActivityNode*, the modeler has to deal with token flow. As stated in Section 2.2, modeling token flow is much different from modeling control or data flow in program code and leads to complex low level structures that obstruct the view on the important parts of an Activity Diagram. If the modeler uses the basic concepts like *DecisionNode*/*MergeNode* for such an iteration, editing the Activity Diagram is such a pedestrian business, that it can be easily understand why UML modelling is used by most software enginneers in an intuitive or informal way only, if it is used at all. Another problem concerning collection iterations is, that the UML does not distinguish between single-valued and multi-valued types. If an attribute specifies a collection, it has a multiplicity greater than 1 and the collection type is given via the meta-attributes *isOrdered* and *is Unique* (see [6], *Kernel::Property*). So the UML abstracts from generic collection types offered by e.g. the Java Collection framework. The CActivityDiagram in Figure (2.66) shows, that modeling an iteration over the elements of such an implicit collection in ActivityDiagrams is much more complex than e.g. the use of the Java Iterator for the Java Collection Framework.

Therefore it would be useful to have a behaviour specification technique that offers the typical structures of higher programming languages. This will make it easier to create and edit UML models and increase the use of the UML for specifying behaviour. To achieve this goal, CUML introduces CUML Nassi-Shneiderman Diagrams (CNSDs).

Although [8] gives examples for Nassi-Shneiderman Diagrams, there is no definition of the concrete syntax of CUML Nassi-Shneiderman Diagrams yet. If you consider, that

this technique was introduced in [15] that was published more than thirty years ago, it is clear that its scope has to be extended to fit the needs of today's state of the art of modelling object-oriented software systems. This will not lead to any problems with the language definition, because [15] introduces Nassi-Shneiderman Diagrams informally as a form of visual pseudo-code for arbitrary platforms, and this is exactly the way it is used here.

2.3.1 CNSDs as Concrete Syntax for CActivities

In the following I will point out how the CNSDs are used as a concrete Syntax for CActivities. To show how the elements of CNSDs are traced back on the abstract syntax of CActivities, I will use CActivity diagrams instead of the abstract syntax of the examples directly.

The activities that can be presented in the concrete syntax of Nassi-Shneiderman Diagrams are a true subset of the CUML activities, because every CNSD statement encapsulates a certain structure of abstract syntax. Activities that contain abstract syntax fragments that do not conform to the required structure of any CNSD statement cannot be presented as CNSDs. Therefore it will be necessary to identify the activites that are contained in the CNSD subset of CUML activities. Figure 2.39 shows that activities can be extended with an instance of the stereotype *CNSDActivity* if they are in the CNSD subset.

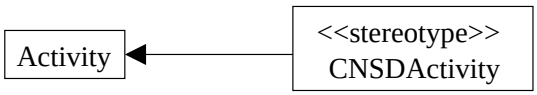

Figure 2.39: Abstract Syntax: CNSDActivity

In this section I will describe how every CNSD statement conforms to an activity structure that is encapsulated by an instance of *SequenceNode*. From this follows that the nodes contained directly by an CNSD activity must be instances of *SequenceNode* or *InitialNode* or *ActivityFinalNode*, what leads to the CNSDActivity constraint 1. Of course this constraint does not guarantee a correct CNSD activity structure by itself. In the following I will introduce further constraints that shall assure a correct activity structure in the abstract syntax for every CNSD statement. Nevertheless I recommend the definition of the language of CNSD Activities in a constructive way, e.g. with a graph grammar, which would be helpful especially for tool support in the future. Unfortunately this is out of the scope of my thesis.

«stereotype» CNSDActivity

- **Metaclass** *Activity*

- **Description** An Activity extended by this stereotype is graphically represented with the conrete syntax of CNSDs.

- **Constraints**

  [1] The directly contained activity nodes have one of the following types: *InitialNode*, *ActivityFinalNode*, *SequenceNode*.

```
context CNSDActivity inv:

  self.baseClass.node->forAll(oclIsTypeOf(InitialNode) or
  oclIsTypeOf(ActivityFinalNode) or
  oclIsTypeOf(SequenceNode))
```

In the following I will introduce the different types of statements.

Assignment

In CNSDs, a value can be assigned to a variable or a *Property*. In the little example in Figure 2.40, the value of an activity variable named 'stateCollection' is assigned to an attribute 'states' of the class owning the operation specified by the CNSD (indicated by 'self').

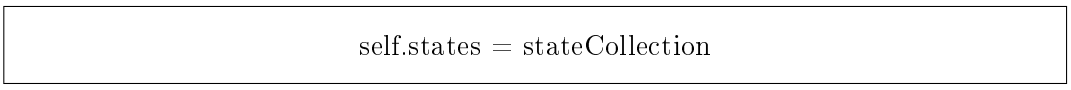

Figure 2.40: CNSD-Assignment.

Figure 2.41 shows the same CActivity as a CActivityDiagram. The value token is created by a *ValueSpecification*. The value is assigned to the *Property* by a *AddStructuralFeatureAction*. The execution semantics of sequence nodes defined in [6] says, that execution of a sequence node leads to a control token placed on every initial node. Further all actions that have no predecessor concerning control flow are executed. Since I use sequence nodes just for encapsulation of CNSD statements, I require sequence nodes to have an initial node that is predecessor to the actions of the statement, to avoid parallel execution (see 2.2 for the description of sequence nodes). Termination of the execution of a sequence node requires an instance of *ActivityFinalNode*.

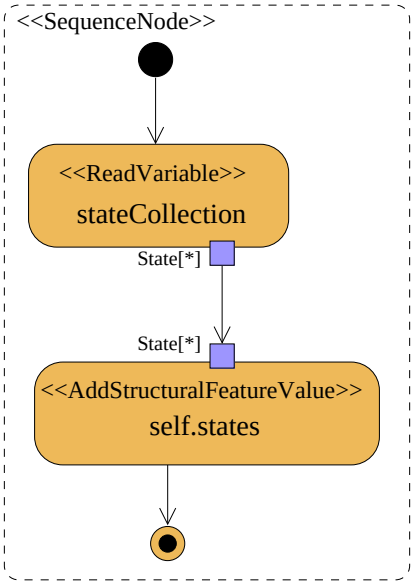

Figure 2.41: CActivityDiagram-Assignment.

In general, objects or values to be assigned can be retrieved from a *ReadStructuralFeatureAction*, *ReadVariableAction* or *ValueSpecificationAction*. The value can be asssigned to a variable with an *AddVariableValueAction* or to an attribute with an *AddStructuralFeatureValueAction*. The action that creates the token passes it to the action that assigns the value of the token. These two actions are encapsulated by an instance of *SequenceNode*. Figure 2.42 shows the metaclass *SequenceNode* extended by the stereotype *CNSDAssignment* to identify the action that provides the value by the end *supplier* and the action that writes the value to a variable or structural feature by the end *designator*.

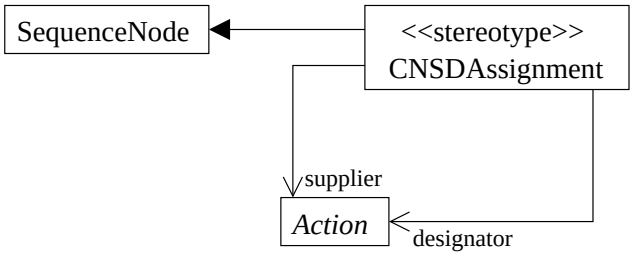

Figure 2.42: Abstract Syntax: CNSDAssignment

The appropriate type of the supplier action, the object flow from the supplier to the designator action and the containment of both actions in the extended sequence node are ensured by the constraints of the stereotype *CNSDAssignment* below:

«stereotype» CNSDAssignment

- **Metaclass** *SequenceNode*
- **Description** A SequenceNode extended by this stereotype is the abstract syntax of a CNSD assignment statement.
- **Tagged Values**

designator: Action[1]	The action that writes the value to an attribute or variable.
supplier: Action[1]	The action that provides the input value for the designator action.

- **Constraints**

[1] The designator action has one of the following types: *WriteVariableAction*, *WriteStructuralFeatureAction*.

```
context CNSDAssignment inv:
  self.designator.oclIsTypeOf(WriteVariableAction) or
  self.designator.oclIsTypeOf(WriteStructuralFeatureAction)
```

[2] The supplier action provides input for the designator action.

```
context CNSDAssignment inv:
  self.supplier.result.outgoing.target =
  self.designator.value
```

[3] Supplier and designator are directly contained by the base class.

```
context CNSDAssignment inv:
  self.baseClass.node.include(self.supplier) and
  self.baseClass.node.include(self.designator)
```

[4] designator is directly contained by the extended sequence node.

```
context CNSDAssignment inv:
  self.baseClass.node->includes(designator)
```

[5] supplier is directly contained by the extended sequence node.

```
context CNSDAssignment inv:
  self.baseClass.node->includes(supplier)
```

Operation call

A CNSD OperationCall statement consists of an instance of *CallOperationAction* which corresponding operation has no result type. Further it consists of the actions necessary to deliver the tokens containing the target object and parameters needed.

self.deleteState(s)

Figure 2.43: CNSD-OperationCall.

The example in Figure 2.43 calls an operation on the context object of the CNSD (indicated by the UML keyword *self*). Figure 2.44 shows the same statement in the concrete syntax of CActivities. Actually the target object must be delivered to the action by an object flow from an instance of *ReadSelfAction*. For simplification, the action icon just shows the name of the operation with suffix 'self'.

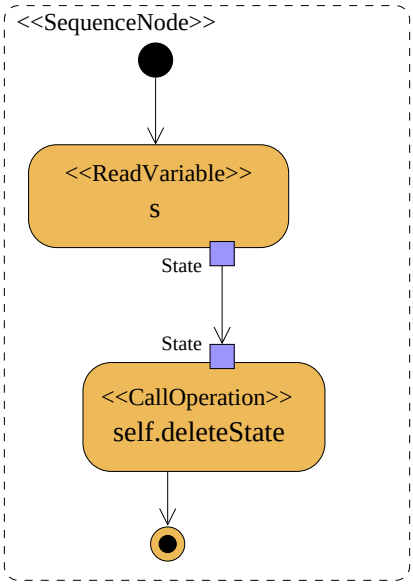

Figure 2.44: CActivityDiagram-OperationCall.

Figure 2.45 shows how the stereotype *CNSDOperationCall* is used to identify an instance of *CallOperationAction*. This instance must be directly contained in the sequence node, see constraint [1] of *CNSDOperationCall*.

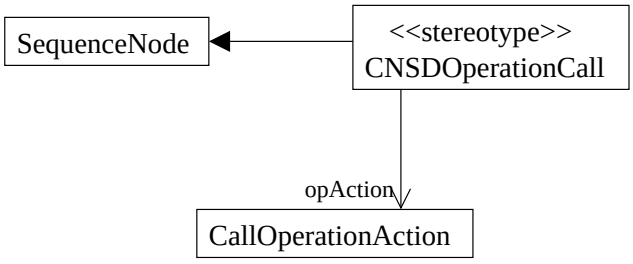

Figure 2.45: Abstract Syntax: CNSDOperationCall.

«stereotype» CNSDOperationCall

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD operation call statement.
- **Tagged Values**

  opAction: CallOperationAction[1] The action that executes the operation.
- **Constraints**

  [1] opAction is directly contained by the extended sequence node.

```
context CNSDOperationCall inv:
 self.baseClass.node->includes(opAction)
```

The assignment statement and the operation call statement are the only kind of atomic CNSD statements. If CUML will be extended to e.g. exception handling, more atomic statements could be necessary. Note that [15] introduces just structural statements like the ones that follow below.

Statement blocks

Figure 2.46 shows a CNSD that consists of two sequentialized statement blocks. Figure 2.47 shows the same CActivity in the concrete Syntax of CActivity Diagrams. Every statement block in the CNSD represents a sequence node in the abstract syntax which encapsulates the behaviour of that statement block. The sequence node of a statement block is connected to the sequence node of its predecessor statement by control flow. If there is no predecessor in the CNSD, it is connected to the initial node. If a statement block has no successor in the CNSD, the corresponding sequence node is connected to the final node by control flow.

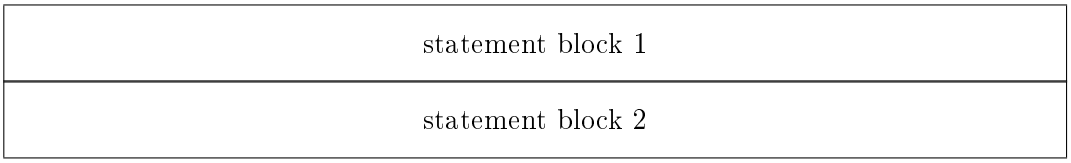

Figure 2.46: CNSD-Sequence.

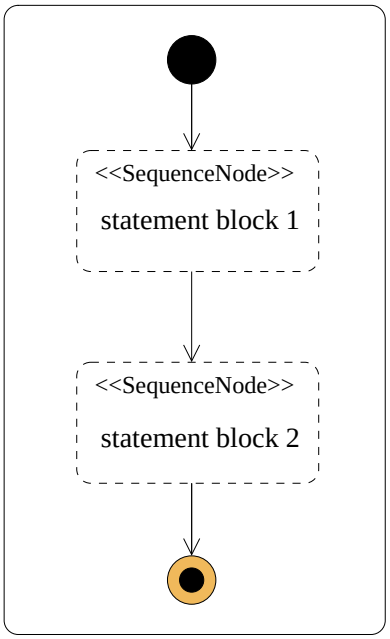

Figure 2.47: CActivityDiagram-Sequence.

IfElse

Figure 2.48 shows an if-else statement block of a CNSD. It contains the specification of a boolean value in the guard expression and two traces for the possible values of the guard expression (**true** and **false**). The traces are executed dependent on the evaluation of the guard expression.

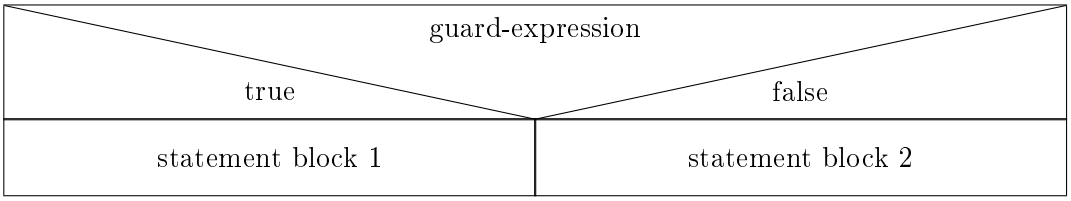

Figure 2.48: CNSD-IfElse

Figure 2.49 shows how the if-else structure is encapsulated by a sequence node. Case distinction is modeled with a decision node. The guard expression is an instancce of *ValueSpecification* associated with the decision node's outgoing egde that has the sequence node of the **true** trace as target. The decision node's other outgoing edge is annotated with the predefined guard 'else' (see [6], *IntermediateActivities::DecisionNode*).

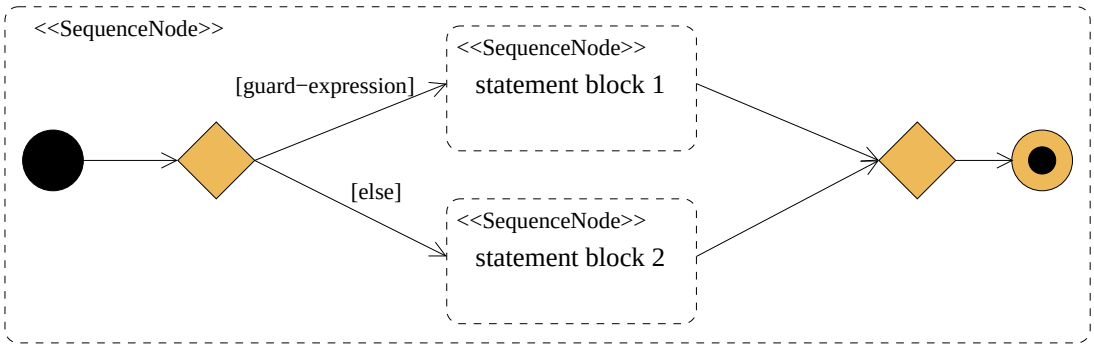

Figure 2.49: CActivityDiagram-IfElse

The example 'Automaton.nameAllocated()' in Figure (2.65) contains an if-else statement block. Figure 2.50 shows the abstract syntax of the stereotype *CNSDIfElse*: The sequence nodes containing the if-body and the else-body are identified by the *ifNode* end and the *elseNode* end respectively. The guard expression is reachable via the *ifCondition* end and the decision node via the *decisionNode* end. The structure of the control flow, i.e. that the sequence nodes containing the if- and else-body are successors of the decision node, is assured by the constraints of the stereotype *CNSDIfElse*.

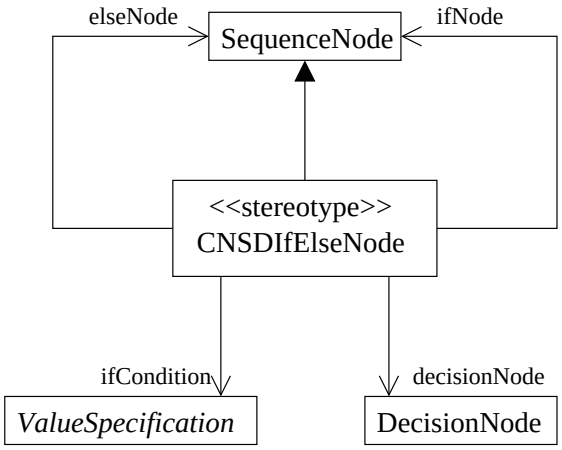

Figure 2.50: Abstract Syntax: CNSDIfElse.

«stereotype» CNSDIfElse

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD IfElse statement.
- **Tagged Values**

decisionNode: DecisionNode[1]	The DecisionNode evaluating the if condition.
elseNode: SequenceNode[1]	Contains the abstract syntax of the CNSD else-trace.
ifCondition: ValueSpecification[1]	The guard expression of decisionNode.
ifNode: SequenceNode[1]	TContains the abstract syntax of the CNSD if-trace.

- **Constraints**

[1] The decisionNode has 2 outgoing edges.

```
context CNSDIfElse inv:
  self.decisionNode.outgoing->size() = 2
```

[2] elseNode and ifNode are successors of decisionNode.

```
context CNSDIfElse inv:
  self.decisionNode.outgoing->collect(source)->
  includes(ifNode->union(elseNode))
```

[3] The ifCondition is guard of the edge from decisionNode to ifNode.

```
context CNSDIfElse inv:
```

```
self.decisionNode.outgoing->exists(guard = ifCondition and
source = ifNode)
```

[4] decisionNode is directly contained by the extended sequence node.

```
context CNSDIfElse inv:

self.baseClass.node->includes(decisionNode)
```

[5] elseNode is directly contained by the extended sequence node.

```
context CNSDIfElse inv:

self.baseClass.node->includes(elseNode)
```

[6] ifNode is directly contained by the extended sequence node.

```
context CNSDIfElse inv:

self.baseClass.node->includes(ifNode)
```

While

The while statement consists of a guard expression in a corner-shaped figure to identify the embedded statements (see Figure 2.51). If the guard expression evaluates to **true**, the embedded statements are executed. This is iterated in a loop until the guard expression evaluates to **false**.

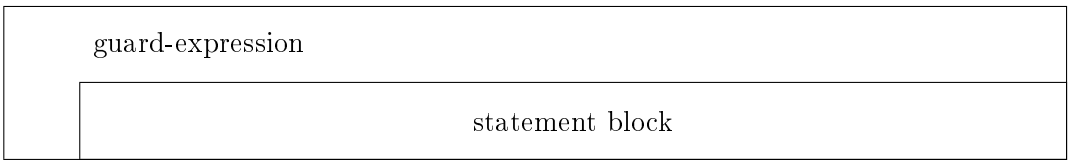

Figure 2.51: CNSD-While

Figure 2.52 shows the same CActivity in a CActivity diagram: the guard expression is an instance of *ValueSpecification* on the outgoing edge of a decision node. The embedded part is encapsulated by a sequence node.

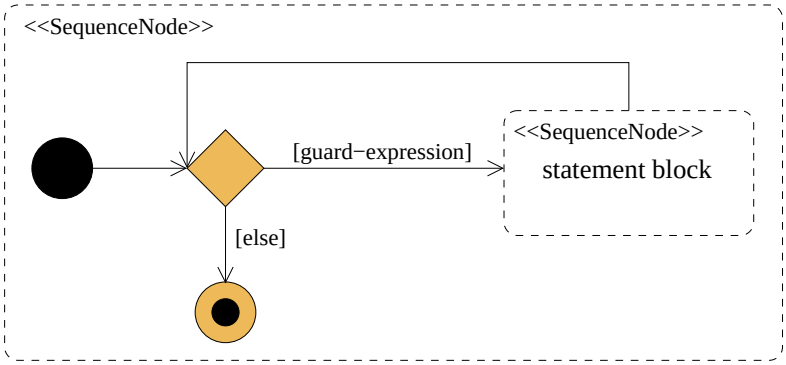

Figure 2.52: CActivityDiagram-While

The abstract syntax in Figure 2.53 shows how the stereotype *CNSDWhile* is used to identify the guard expression (*loopCondition* end), the decision node (*decisionNode* end) and the sequence node containing the activity modelling of the loop body (*body* end):

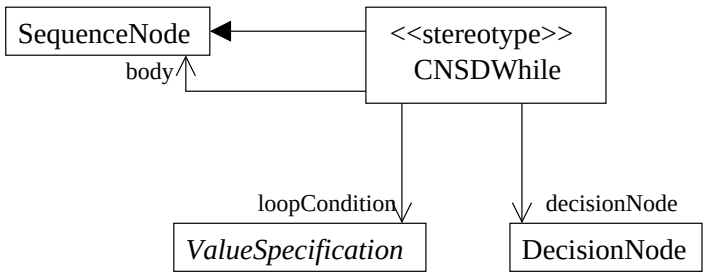

Figure 2.53: Abstract Syntax: CNSDWhile.

The constraints for the stereotype CNSDWhile assure the correct structure of control flow and that the guard annotates the control edge from the decision node to the body.

«stereotype» CNSDWhile

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD While statement.
- **Tagged Values**

decisionNode: DecisionNode[1]	The DecisionNode evaluating the loop condition.
body: SequenceNode[1]	Contains the abstract syntax of the CNSD loop body.
loopCondition: ValueSpecification[1]	The guard expression of decisionNode.

- **Constraints**

  [1] The body node is predecessor of the decisionNode.

  ```
  context CNSDDoWhile inv:
    self.body.outgoing.target = decisionNode
  ```

  [2] The body node is successor of the decisionNode.

  ```
  context CNSDDoWhile inv:
    self.body.incoming.source = decisionNode
  ```

  [3] The loopCondition is guard of the edge from decisionNode to body.

  ```
  context CNSDDoWhile inv:
    self.body.incoming.guard = loopCondition
  ```

  [4] The decision node is successor of the *InitialNode*.

  ```
  context CNSDDoWhile inv:
    self.decisionNode.incoming.source.oclIsTypeOf(InitialNode)
  ```

  [5] decisionNode is directly contained by the extended sequence node.

  ```
  context CNSDDoWhile inv:
    self.baseClass.node->includes(decisionNode)
  ```

  [6] body is directly contained by the extended sequence node.

  ```
  context CNSDDoWhile inv:
    self.baseClass.node->includes(body)
  ```

DoWhile

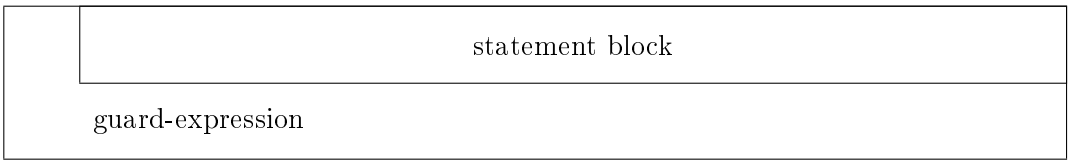

Figure 2.54: CNSD-DoWhile

The do-while statement (see Figure 2.54) is very similar to the while statement. The CActivity diagram can be seen in Figure 2.55.

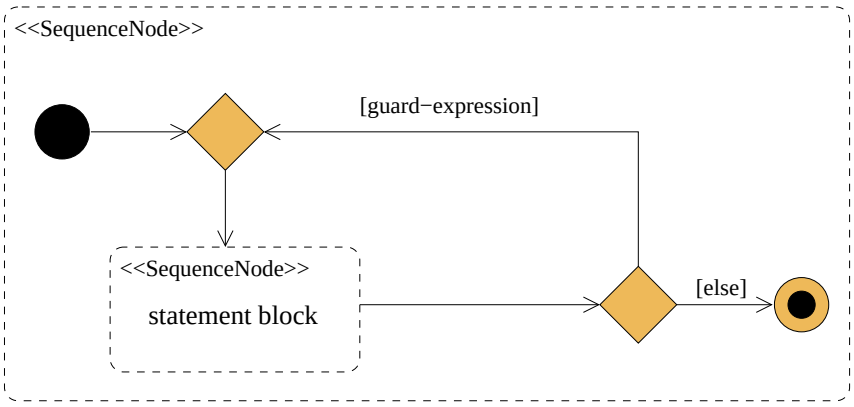

Figure 2.55: CActivityDiagram-DoWhile

As the Figure 2.56 shows, the stereotype *CNSDDoWhile* identifies the same elements as the stereotype *CNSDWhile*. But the constraints of *CNSDDoWhile* ensure the different structure of control flow: the sequence node containing the body is successor of a merge node which is successor of the initial node.

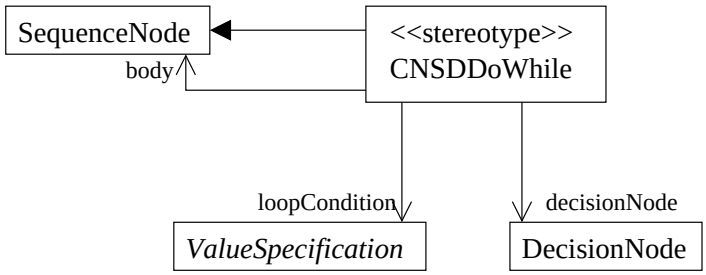

Figure 2.56: Abstract Syntax: CNSDDoWhile.

«stereotype» CNSDDoWhile

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD DoWhile statement.
- **Tagged Values**

decisionNode: DecisionNode[1]	The DecisionNode evaluating the loop condition.
body: SequenceNode[1]	Contains the abstract syntax of the CNSD loop body.
loopCondition: ValueSpecification[1]	The guard expression of decisionNode.

- **Constraints**

  [1] The body node is predecessor of the decisionNode.

  ```
  context CNSDDoWhile inv:
    self.body.outgoing.target = decisionNode
  ```

  [2] The body node is successor of the decisionNode.

  ```
  context CNSDDoWhile inv:
    self.body.incoming.source.oclIsTypeOf(MergeNode)
  ```

  [3] The loopCondition is guard of the edge from decisionNode to body.

  ```
  context CNSDDoWhile inv:
    self.decisionNode.outgoing.guard = loopCondition and
    self.body.incoming.source.incoming->
    exists(e | e.guard = loopCondition)
  ```

  [4] The body node is successor of the *MergeNode*.

  ```
  context CNSDDoWhile inv:
    self.body.incoming.source.oclIsTypeOf(MergeNode)
  ```

  [5] The body predecessor node is successor of the *InitialNode*.

  ```
  context CNSDDoWhile inv:
    self.body.incoming.source.incoming.source.
    oclIsTypeOf(InitialNode)
  ```

  [6] decisionNode is directly contained by the extended sequence node.

  ```
  context CNSDDoWhile inv:
    self.baseClass.node->includes(decisionNode)
  ```

  [7] body is directly contained by the extended sequence node.

  ```
  context CNSDDoWhile inv:
    self.baseClass.node->includes(body)
  ```

Foreach

The foreach statement is introduced to CNSDs in [8]. It allows iteration over a multivalued property of a classifier, i.e. an attribute or owned association end, or over a multivalued variable. Every element of the multivalued data is accessed once. There is no specific ordering unless there is an ordering specified for the property or variable.

Notationally, the foreach statement specializes the while statement. As the example in figure (2.65) shows, the foreach statement is visualized as a while statement with the

keyword 'foreach' at the top followed by the initialization of the iteration variable: in the example, an element of 'self.states' is assigned to a variable 's' during each single iteration.

Figure 2.57: CNSD-ForEach

Figure 2.58 shows the same CActivity as Figure 2.57 in the concrete syntax of CActivity diagrams. The values of the collection (an attribute or variable) are stored in an instance of *CentralBufferNode*, from where token containing the elements are provided to the sequence node encapsulating the body of the foreach statement severally. In the body the element value is written to a variable first, to make it accessible within the body recursively. If the execution of the body stops, the next element is retrieved from the buffer node and execution of the body starts for the new element value. If execution of the body stops for the last element of the collection, there is no token left in the sequence node encapsulating the foreach statement recursively, so execution of this *SequenceNode* stops.

If the foreach loop is left via break statement, it is necessary to stop execution and destroy all tokens contained by offering a token to the final node (see example in Figure 2.66). Otherwise it would be possible that token remain in the buffer node. That could cause a deadlock situation if the foreach statement is contained in the body of another foreach statement.

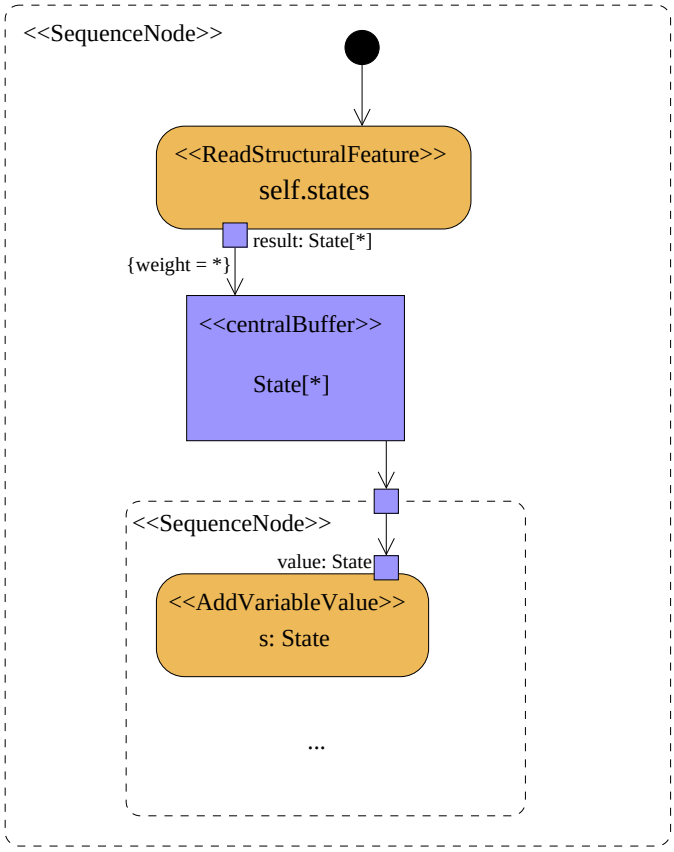

Figure 2.58: CActivityDiagram-Foreach

Figure 2.59 shows the abstract syntax of the stereotype *CNSDForEach*: the action that provides the values of the collection, the buffer for these values, the action that writes a single value of the collection to the iteration variable and the sequence node containing the iteration body are identified.

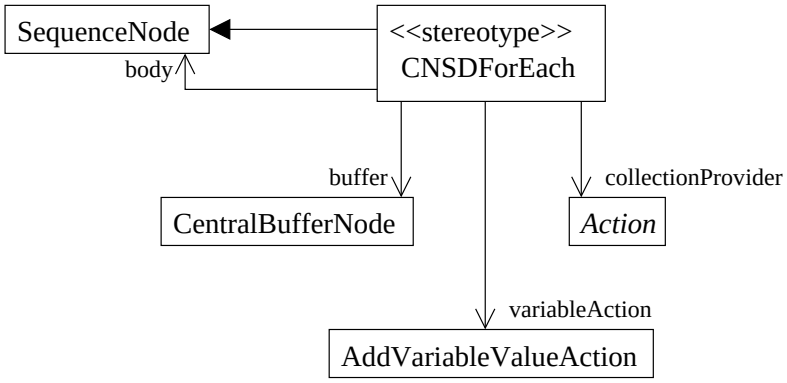

Figure 2.59: Abstract Syntax: CNSDForEach.

«stereotype» CNSDForEach

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD ForEach statement.
- **Tagged Values**

collectionProvider: Action[1]	Action that provides the elements of the collection.
buffer: CentralBufferNode[1]	Buffer that stores the elements of the collection.
variableAction: AddVariableValueAction[1]	Action that writes the value of the variable before each iteration.
body: SequenceNode[1]	SequenceNode encapsulating the body of the foreach statement.

- **Constraints**

[1] CollectionProvider is of one of the following types: ReadVariableAction, ReadStructuralFeatureAction.

```
context CNSDForEach inv:
  self.collectionProvider.oclIsTypeOf(ReadVariableAction)
  or self.collectionProvider.
  oclIsTypeOf(ReadStructuralFeatureAction)
```

[2] Edge weight from collectionProvider's result pin to buffer is '*'.

```
context CNSDForEach inv:
  self.collectionProvider.result.outgoing.weight = *
```

Case

The example in Figure 2.60 contains a case statement[4]. The case statement consists of a given value of any type and traces with guard expressions. A trace will be executed, if its guard expression can be evaluated to the given value. If the given value could not be evaluated to any of the guard expressions, the default trace is executed.

If more than one guard expression can be evaluated to the given value, the choice of the trace to be executed is non-deterministic. Such non-determinisms will be dealt with on the semantical domain side.

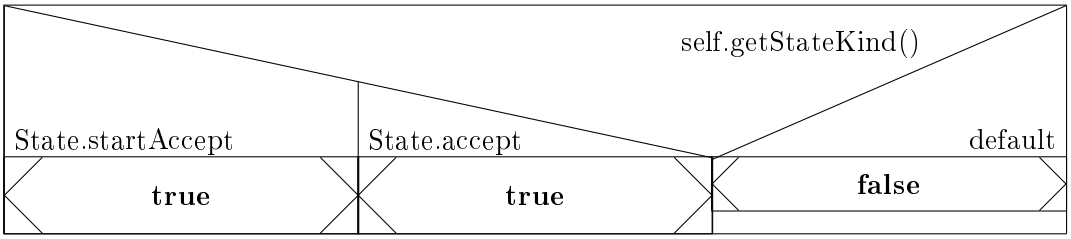

Figure 2.60: CNSD: State.isAccepting

Figure 2.61 shows the example in Figure 2.60 as CActivityDiagram. The case distinction is realized with a decision node again. The traces for the different cases are encapsulated by a sequence node. For every trace, an outgoing edge of the decision node leads to the corresponding sequence node. The edges are associated with an expression that compares the guard expression of the trace to the given value of the case statement. The sequence node of the default trace is connected to the decision node via a control flow associated with the predefined 'else'-guard. If there is no body for the default trace, the activity edge with the else guard has the final node as target to avoid deadlock situations.

[4]The example contains break statements that will be introduced below.

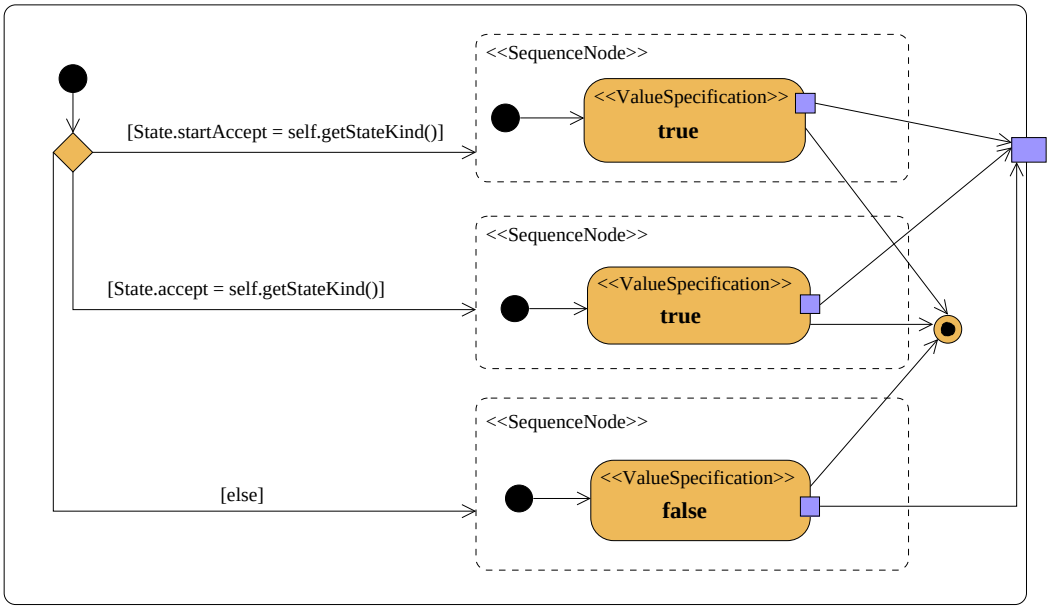

Figure 2.61: CActivity: State.isAccepting

The abstract syntax of *CNSDCase* in Figure 2.62 shows, that the stereotype is used to identify the decision node that is responsible for case distinction and the sequence nodes containing the activity modelling of the different traces.

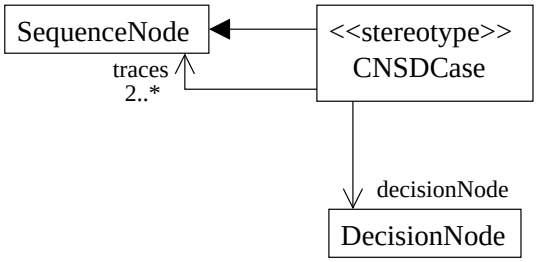

Figure 2.62: Abstract Syntax: CNSDCase.

«stereotype» CNSDCase

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD case statement.
- **Tagged Values**

decisionNode: DecisionNode[1]	The decision node implementing the case distinction.
traces: SequenceNode[*]	The sequence nodes that encapsulate the abstract syntaxes of the different traces of the case statement.

- **Constraints**

  [1] The decision node is predecessor to all traces.

  ```
  context CNSDCase inv:
    self.traces->collect(incoming)->forAll(source = decisionNode)
  ```

  [2] decisionNode is directly contained by the extended sequence node.

  ```
  context CNSDCase inv:
    self.baseClass.node->includes(decisionNode)
  ```

  [3] All traces are directly contained by the extended sequence node.

  ```
  context CNSDCase inv:
    self.baseClass.node->includes(traces)
  ```

Break

With the break statement the control flow is passed to the successor of the encapsulating loop (see Figure 2.63). Within a loop, i.e. a while, dowhile or foreach statement, the encapsulating loop is exited and the successor statement of the loop is executed.

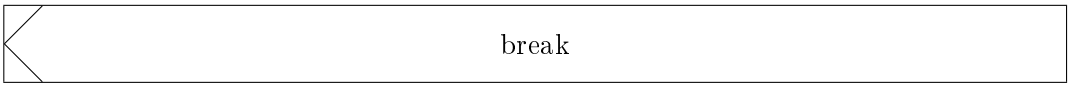

Figure 2.63: CNSD-break

The example in Figure 2.66 shows the realization of the break statement in CActivities: the break statement occurs within a foreach statement in the CNSD (Figure 2.65). The predecessor statement of the break statement is an assignment ('allocated = **true**'). The corresponding sequence node of that assignment in Figure 2.66 is connected to the final node of the foreach statement's sequence node. This causes the execution of the foreach statement to stop and the successor of the foreach statement is executed.

Return

The return statement (notation s. Figure 2.64) ends the control flow and provides the return value of the method. The return value can be an attribute, variable or given value.

The example 'Automaton.nameAllocated' (figures 2.65, 2.66) contains a return statement with variable. The CActivityDiagram in Figure 2.66 shows, that the return value is retrieved from an instance of *ReadVariableAction* and passed to the parameter node

that corresponds to the return parameter of the activity's operation. The control flow is passed to the final node of the enclosing CActivity.

The example in Figure 2.60 contains a return statement with *ValueSpecification*.

Figure 2.64: CNSD-return

2.3.2 Example Diagrams

The example diagrams show the CActivity of the operation *Automaton.nameAllocated*. Figure 2.65 is the CNSD representation of this CActivity.

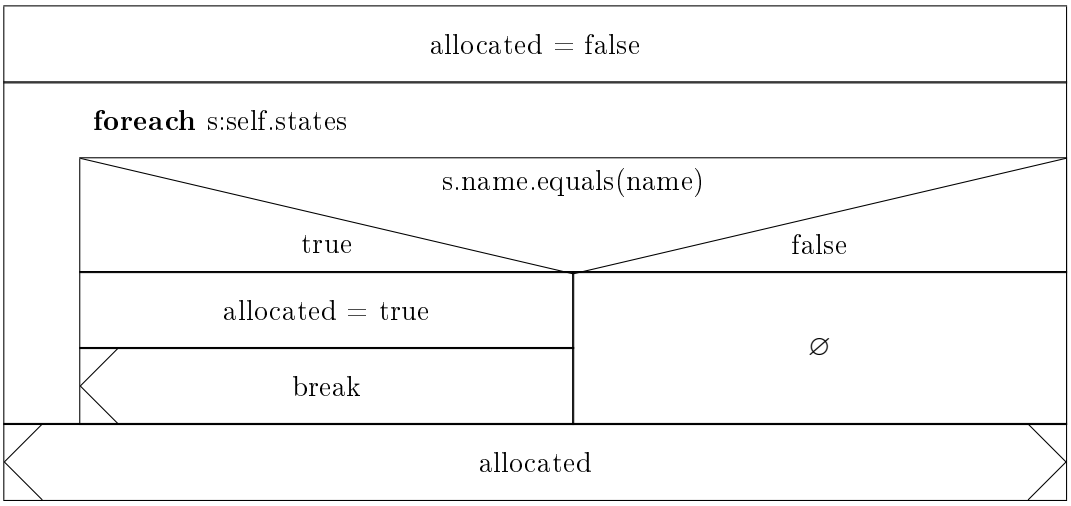

Figure 2.65: CNSD: Automaton.nameAllocated.

Figure 2.66 is the CActivity Diagram representation of the same CActivity. The structure of the CActivity Diagram contains the sequence nodes that correspond to the statements of the CNSD in 2.65. But even if this encapsulating sequence nodes would be eliminated [5], it is obvious that the CNSD representation is much more structured than the CActivityDiagram. Especially the modelling of the iterator loop is easier to edit and understand thanks to the compact and intuitive CNSD foreach statement.

[5] e.g. by the first level of the model transformation in 4, leading to the CActivityDiagram in Figure 4.7.

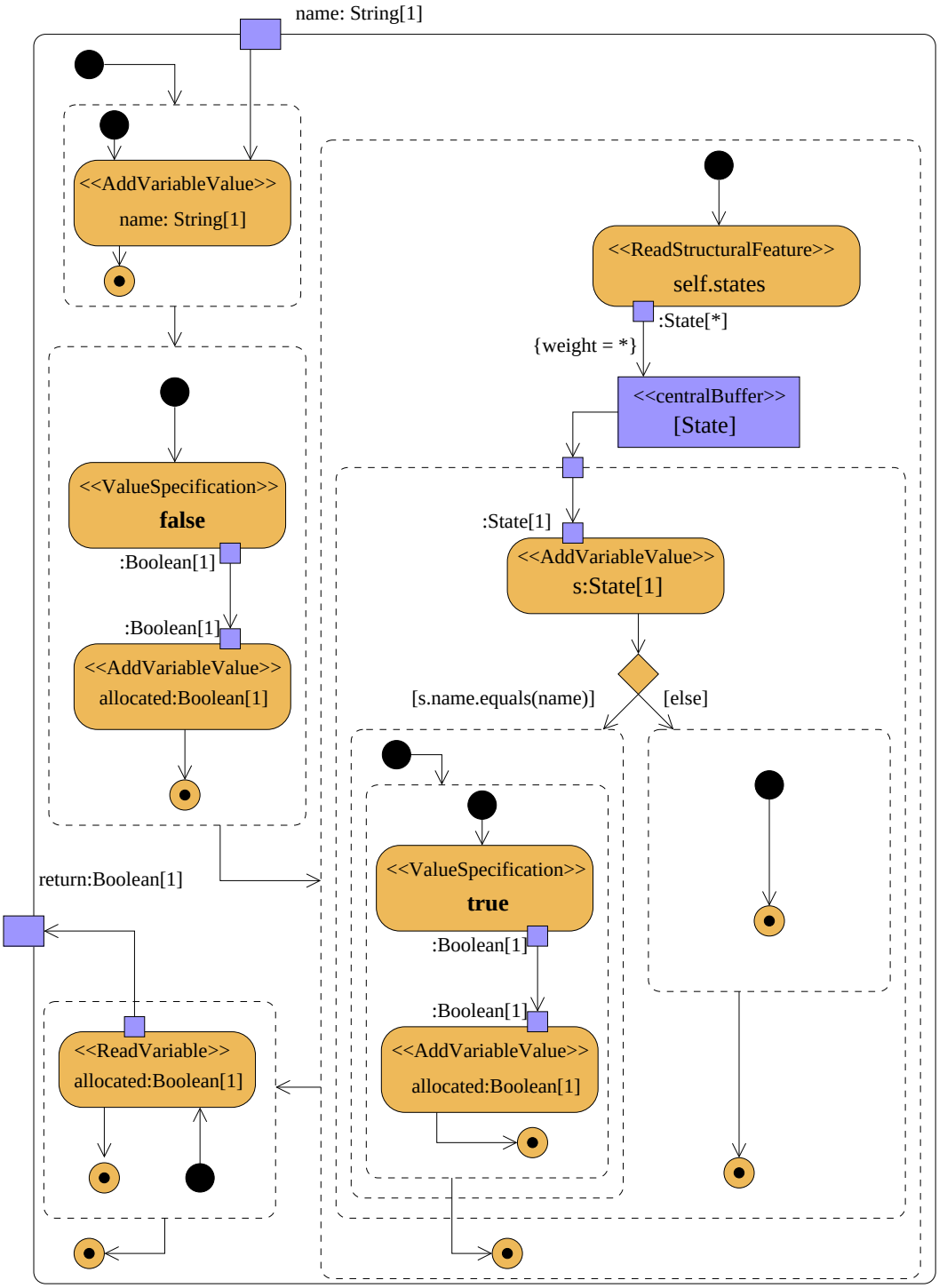

Figure 2.66: CActivityDiagram: Automaton.nameAllocated.

3 L3 - Semantical Domain Language

In view of the many approaches that define formal semantics for UML Activities by mapping to a semantical domain, the introduction of another semantical domain like L3 has to be motivated. There are three main arguments for the introduction of a new semantical domain for complex modelling techniques:

[1] [16] contains an examination of the semantical domains for UML Activities. Looking at the expressiveness of this semantical domains, it becomes obvious that only Petri Nets cover data flow, which is an essential feature of CUML. As stated before, [18] shows that there is no known version of Petri Nets that covers all features of UML Activities. Although CUML contains just a restricted version of UML Activities itself, L3 should not be restricted, since it is supposed to serve as semantical domain for arbitrary modelling techniques for object-oriented software systems.

[2] L3 is designed as a semantical domain for a complex modelling technique. While there are a lot of approaches to define semantics for sublanguages of the UML - e.g. [9]: ASMs for Activities, [12]: Object-Z for Class Diagrams, [14]: Graph Transformation for State Machines -, there is none that gives a semantical domain for a set of modelling techniques that contains languages for the specification of the structure, the behaviour and the reqirements of an object-oriented software system. The use of existing approaches would lead to the semantical domain consisting of multiple formalisms, whereas it is in question if they could be combined at all: especially the way how behaviour references the structure of an object-oriented system is important at this point. L3 however consists of a structural part, a constructive part and a descriptive part and the semantics will be defined for these parts together.

[3] The main purposes of this approach are verification and code generation. For verification, different semantical domains for the different views of a complex modelling technique have to be analyzed. This requires interoperability of the semantical domains, which is often achieved informally: e.g., the connection of behaviour and structure is often achieved via name mapping. Further the structure of semantical domains of existing approaches is not very similar to the structure of source code of object-oriented programming languages, so code generation must be based on the complex model again, which requires another integration of the views of the complex model. L3 contains concepts for all important features of object-oriented programming languages, so that program code can be generated from an L3 model directly.

In this chapter I will introduce the Low-Level-Language (L3) by a MOF M2 layer meta model. Although [8] describes many properties of L3 and gives a lot of examples, there is still no formal definition of this language.

As stated before, L3 is the common semantical domain for arbitrary object-oriented modelling techniques in the integration concept for complex modelling techniques. Therefore it has to feature the following properties:

- **explicit contents:** As we have seen in Chapter 2 looking at the UML, there can be a lot of information about a model that is contained in this model implicitly. One disadvantage of this is, that there is information contained in the model that is not easily accessible, what is bad for the people who have to document, maintain it etc. I have tried to avoid this in CUML by introducing some stereotypes to make implicit relations between model elements explicit. Further implicit model contents can make the work on the semantical domain side - as model checking, code generation - more difficult.
- **separation of aspects:** As stated in Chapter 1, the information in a model can be divided in structural, constructive and descriptive aspects of the model. Mixing these up in the modelling techniques can help to make the work of the system modeller more easy. But in the semantical domain the structure and the behaviour should be strictly separated from the properties both have to fulfill in the implementation. This makes constraints and requirements explicit for model checking.
- **low level concepts:** Since L3 should be designed to serve as semantical domain for arbitrary object-oriented modelling techniques, it has to be a target domain for model transformation from every possible technique as the source domain. Therefore the concepts for structure and behaviour modelling in L3 have to be on a low level of abstraction.
- **model tracing:** Since the model checking operates on the L3 model, it is necessary for each L3 model element to be linked to its source element in the CUML model for modeler feedback. Due to this, the structure of the L3 meta model is very similar to the structure of the CUML meta model. For the explicit tracing of model elements an intermediate structure meta model is introduced in 4.1.

In Section 3.1 I will introduce the structural part of L3. Although, as stated above, the structural issue is out of the scope of my thesis, I will need the structure meta model to introduce the constructive behaviour modelling part of L3 in Section 3.2. Section 3.3 describes the L3 expression meta model.

3.1 L3 Structure

As stated above, dealing with the structural part of L3 is out of the scope of my thesis and I will therefore introduce a basic version of the L3 Class Structure that features the basic concepts I will need for the behaviour part of the meta model below.

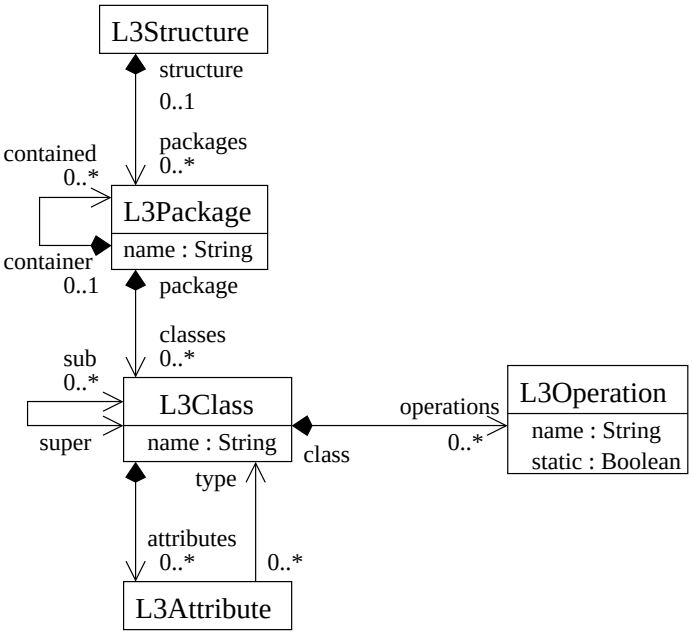

Figure 3.1: L3 Structure

Figure 3.1 shows the abstract syntax of the L3 Class Structure: An L3 model is a tree with the root being an instance of the metaclass *L3Structure*. Every L3 model contains exactly one instance of *L3Structure*:

- *L3Structure* is a singleton class.

```
context L3Structure inv:
 self->allInstances()->size() = 1
```

The class structure consists of a number of packages. Root packages are linked to the class structure directly via the *L3Package::structure* link end, while nested packages are linked to the package being the container. There must be no cycle in the package hierarchy, what is ensured by the following OCL constraint:

- No cycles in the package hierarchy.

```
context L3Package inv:
 self->getContainer(Bag)->excludes(self)

L3Package::getContainer(acc:Bag(L3Package)):  Bag(L3Package);
getContainer=
 if self.container->size() = 0
 then acc
  else self.container->getContainer(acc->including(self.container))
 endif
```

An instance of *L3Package* must either be directly linked to the class structure or be contained in another package:

- Every *L3Package* must have an owner.

```
context L3Package inv:

 self.structure + self.container = 1
```

As can be seen from the abstract syntax, the inheritance structure in L3 is restricted to single-inheritance. Every class must be contained in a package. To ensure the hierarchy of *L3Class*es to be cycle-free, the following constraint is used:

- No cycles in the class hierarchy.

```
context L3Class inv:

 self->getSuper(Bag)->excludes(self)

L3Class::getSuper(acc:Bag(L3Class)):  Bag(L3Class);
getSuper=
 if self.super.name = Object
 then acc
 else self.super->getSuper(acc->including(self.super))
 endif
```

The helper operation in the constraint above shows that there is a default superclass in L3 named 'Object' that is the root element of any class hierarchy tree.

In Figure 3.2 the abstract syntax of operations in L3 is shown. In a valid L3 model, every operation must be linked to an instance of *L3Method* via the *L3Operation::method* link end. This method is the specification of the operation's behaviour. The operation is linked to its parameters via the *L3Operation::parameter* link end. If the operation can have side-effects on a parameter, this is indicated by the parameter's attribute *isOut*. The method's return paramter is identified by the *L3Operation::return* link end.

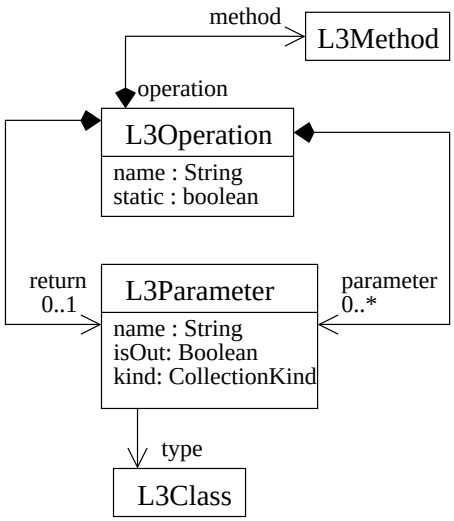

Figure 3.2: L3 Operation

Because the language L3 is designed as a backend to be used for model checking and code generation, there is no concrete syntax. Nevertheless I like to show the results of the transformations of the examples given in Chapter 2. For the visualization of the class structure resulting from the class diagram in Figure 2.4 I use a modified form of class diagrams in Figure 3.3.

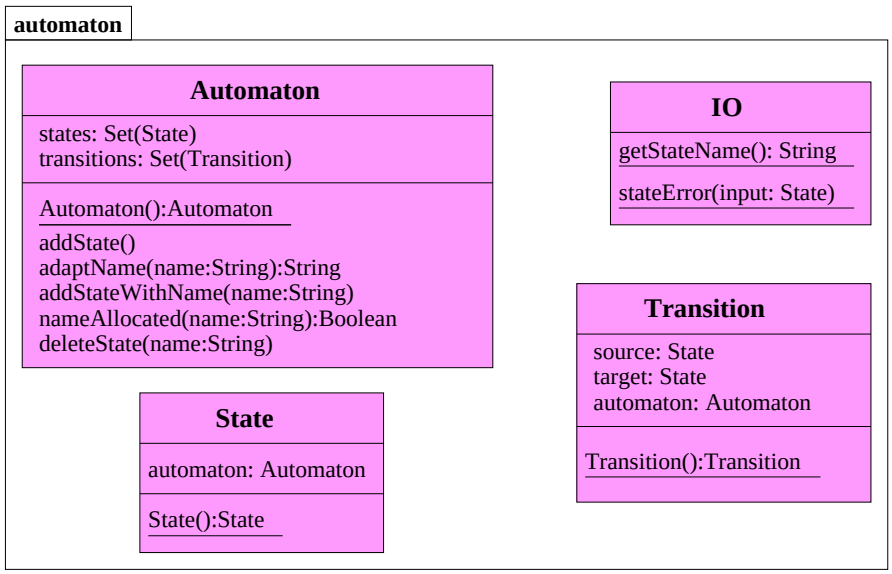

Figure 3.3: L3 Package: automaton

The class structure diagram shows, that the UML association ends result in attributes, so there are no associations in L3. Due to the principle of separation of aspects the information about the correspondence of the ends of an association is not contained in the structural part of L3, but leads to the following constraints in the descriptive part:

- Correspondence of states and automaton.

```
context Automaton inv:

  self.states->forAll(automaton = self) and
  State.allInstances(s | automaton.states.includes(s))
```

- Correpondence of transitions and automaton.

```
context Automaton inv:

  self.transitions->forAll(automaton = self) and
  Transition.allInstances(t | automaton.transitions.includes(t))
```

The specification of multiplicities in CUML class diagrams can lead to further constraints in the descriptive part of L3. But the desriptive part is out of the scope of this thesis. The constraints above just illustrate the separation of aspects and shall not propose a certain modelling technique for the descriptive part.

Another principle mentioned before was the explicit character of L3. The attributes of the class structure, that were specified as collections by their multiplicity in the CUML class diagram implicitly are now specified with a collection type explicitly. Table 3.1 shows, how the L3 collection types correspond to the combinations of the attributes *isOrdered* and *isUnique* of the UML metaclass *MultiplicityElement*. The abstract syntax of L3 collections is shown in Figure 3.8.

	unique	not unique
ordered	OrderedSet	Sequence
not ordered	Set	Bag

Table 3.1: L3 Collection Types

Another element of the class diagram in Figure 2.4 is the «uses» dependency from the class *Automaton* to the class *IO*. Dependencies can be used in UML models to make existing relations between model elements more obvious within certain UML views. They are of no use for the purposes of L3 - i.e. model checking and code generation -, and therefore will be not translated during model transformation.

3.2 L3 Method

The *L3Method* metaclass is the L3 concept for behaviour specification. Concerning the model transformation from CUML to L3, every CUML Activity will be translated into

an L3 Method. On the first glance an L3 Method may look very similar to CUML Activities, but although they feature the same basic concepts of control flow there are two main differences: where CUML Activities model the flow of data via token flow, the access to data within L3 Methods is managed via local variables. Further there are no actions in L3 that are provided with values at their pins, but expressions that feature operation calls and data access. This is due to the fact, that the abstract syntax of L3 is designed to be close to the domains of code generation, e.g. Java Bytecode or the Common Intermediate Language (CIL) of the .NET Framework.

Figure 3.4 shows the abstract syntax of *L3Method*. Every L3 Method consists of instances of *L3Node* and *L3Edge* and has a graph-like structure. An L3 Method has local variables and arguments.

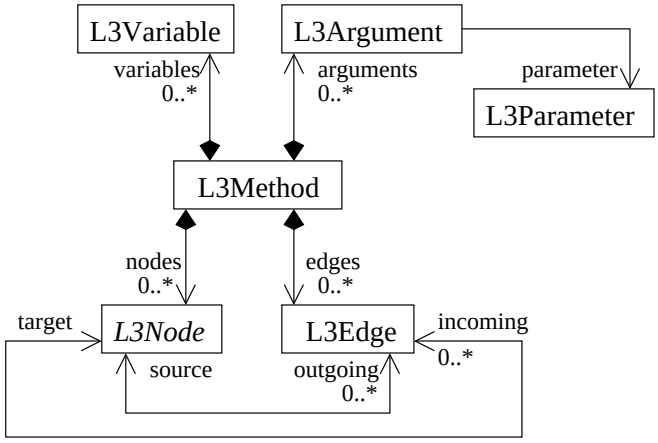

Figure 3.4: L3 Method

Every argument of an L3 Method corresponds to a parameter of the operation of which the method is the specification.

- The corresponding parameter of an *L3Argument* has to be one of the parameters of the argument's *L3Method* and vice versa.

```
context L3Method inv:

 self.argument->collect(parameter) = self.operation.parameter
```

The abstract syntax diagram in Figure 3.5 shows the concrete subclasses of *L3Node*.

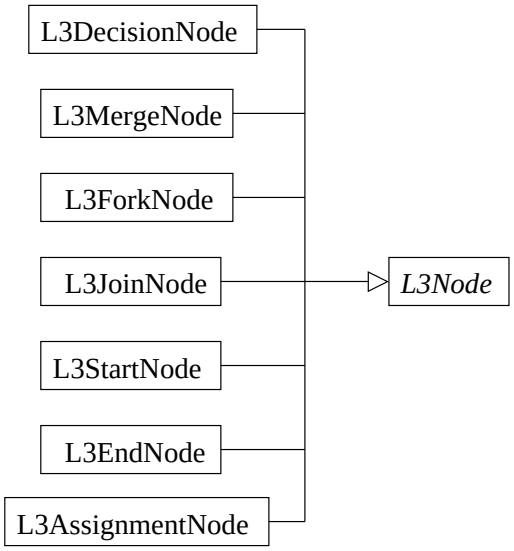

Figure 3.5: L3 Nodes

Instances of *L3StartNode* are the nodes where control flow starts in an L3 method during execution. A method can have any number of start nodes. Start nodes can have only one outgoing edge and no incoming edges:

- A start node has no incoming edge.

```
context L3StartNode inv:
  self.incoming->size() = 0
```

- A start node has one outgoing edge.

```
context L3StartNode inv:
  self.outgoing->size() = 1
```

If control flow reaches an *L3EndNode* during method execution, the execution terminates. An end node can have one incoming edge and no outgoing edge:

- An end node has no incoming edge.

```
context L3EndNode inv:
  self.incoming->size() = 1
```

- An end node has one outgoing edge.

```
context L3EndNode inv:
  self.outgoing->size() = 0
```

L3ForkNode allows parallel control flow: an instance of this class can have any number of outgoing edges and one incoming edge. During method execution, all outgoing edges are traversed simultaneously. Counterpart of *L3ForkNode* is the metaclass *L3JoinNode*. An instance of this can have any number of incoming edges and one outgoing edge. During execution, the outgoing edge is not traversed until control flow has activated all incoming edges. This leads to the following constraints:

- A fork node has one incoming edge.

```
context L3ForkNode inv:
 self.incoming->size() = 1
```

- A join node has one outgoing edge.

```
context L3JoinNode inv:
 self.outgoing->size() = 1
```

DecisionNode allows branching of control flow. The outgoing edges of a decision node are annotated with an expression [1]. During execution, all outgoing edges which guards evaluate to **true** are traversed. Only the outgoing edges of decision nodes are allowed to have guards.

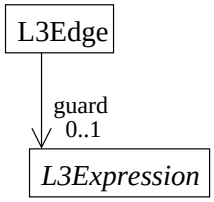

Figure 3.6: L3 Edge Guard

- An outgoing edge of a decision node has a guard expression.

```
context L3Edge inv:
 self.source->oclIsTypeOf(L3DecisionNode) implies
 self.guard->size() = 1
```

- Other edges have no guard expressions.

```
context L3Edge inv:
 not(self.source->oclIsTypeOf(L3DecisionNode)) implies
 self.guard->size() = 0
```

[1] Expressions are introduced in 3.3 below.

Figure 3.7 shows the abstract syntax of *L3AssignmentNode*. The value to be assigned is provided by evaluation of the expression at the *supplier* link end. The value is assigned to the value container at the *designator* link end. The L3 concept of value containers is introduced in Figure 3.8 below.

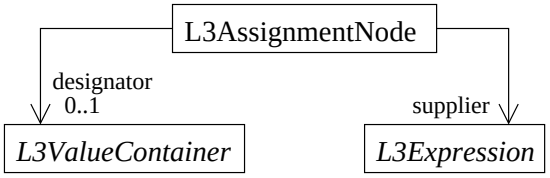

Figure 3.7: L3 AssignmentNode

The type of the designator of an assignment node has to conform to the type of the supplier. The types of value containers and expressions are indicated by associated instances of *L3Class*, since L3 does not deal with primitive data types yet. This issue is out of the scope of my thesis, so I propose wrapper classes in the L3 model for the primitive data types. If the supplier expression refers to a call of a static operation, there must not be a designator:

- Type conformance of designator and supplier [2].

```
context L3AssignmentNode inv:

  self.supplier.type->getSuper()
  ->including(self.supplier.type)->includes(self.designator.type)
```

- If the called operation is static, there is no designator.

```
context L3AssignmentNode inv:

  self.supplier.operation.static implies
  self.designator->size() = 0
```

The value containers attributes, variables and arguments, that appear as an assignment designator within an L3 method, have to be accessible within the method. This is ensured by the following constraints:

- If a *L3Attribute* is designator of a *L3AssignmentNode*, the *L3Attribute* is accessible within the *L3Method* containing the *L3AssignmentNode*.

```
context L3Method inv:

  self.nodes->select(oclIsTypeOf(L3AssignmentNode))
  ->select(designator.oclIsTypeOf(L3Attribute))
  ->forAll(a | self.operation.class.accessible(a))
```

[2]The operation L3Class::getSuper() is defined in Section 3.1.

```
L3Class::accessible(a:  L3Attribute):  boolean;
accessible=
 if self.attributes->exists(attr | (attr = a) or (attr::accessible(a))
 then true
 else false
 endif
```

- If an *L3Variable* is designator of a *L3AssignmentNode*, the *L3Variable* must be contained by the enclosing *L3Method*.

```
context L3Method inv:

 self.nodes->select(oclIsTypeOf(L3AssignmentNode))
 ->select(designator.oclIsTypeOf(L3Variable))
 ->forAll(v | self.variables->includes(v))
```

- If an *L3Argument* is designator of a *L3AssignmentNode*, it hast to be an argument of the enclosing *L3Method*.

```
context L3Method inv:

 self.nodes->select(oclIsTypeOf(L3AssignmentNode))
 ->select(designator.oclIsTypeOf(L3Argument)
 ->forAll(a | self.arguments->includes(a))
```

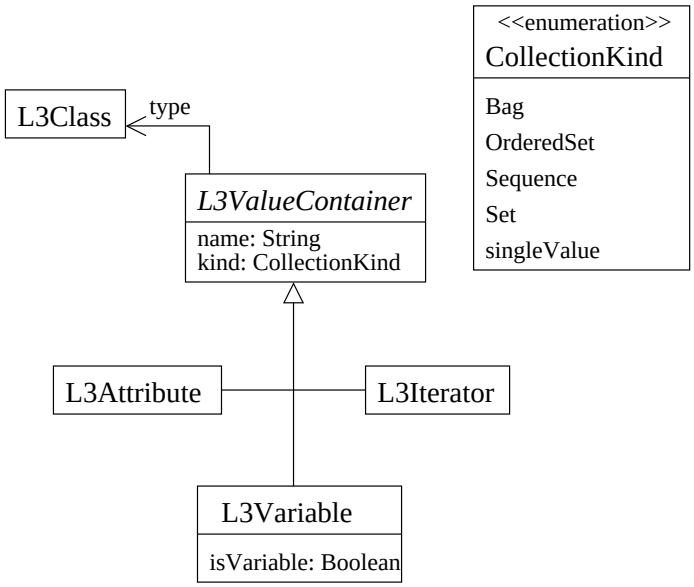

Figure 3.8: L3 ValueContainer

The abstract syntax of L3 value containers is shown in Figure 3.8. Value containers give access to values as attributes of classes, iterators for collections, arguments of methods

and local variables of methods. Every value container can be a collection of any type (see 3.1 for a characterization of collection types) or a single value. If a variable is an iterator, it can be operated on with iterator expressions (see 3.3).

Figure 3.9 shows the L3 Method of the L3 Operation *Automaton.addState*. This Method is the result of the model transformation I will introduce in 4. Its source CUML Activity is the example in Figure 2.5.

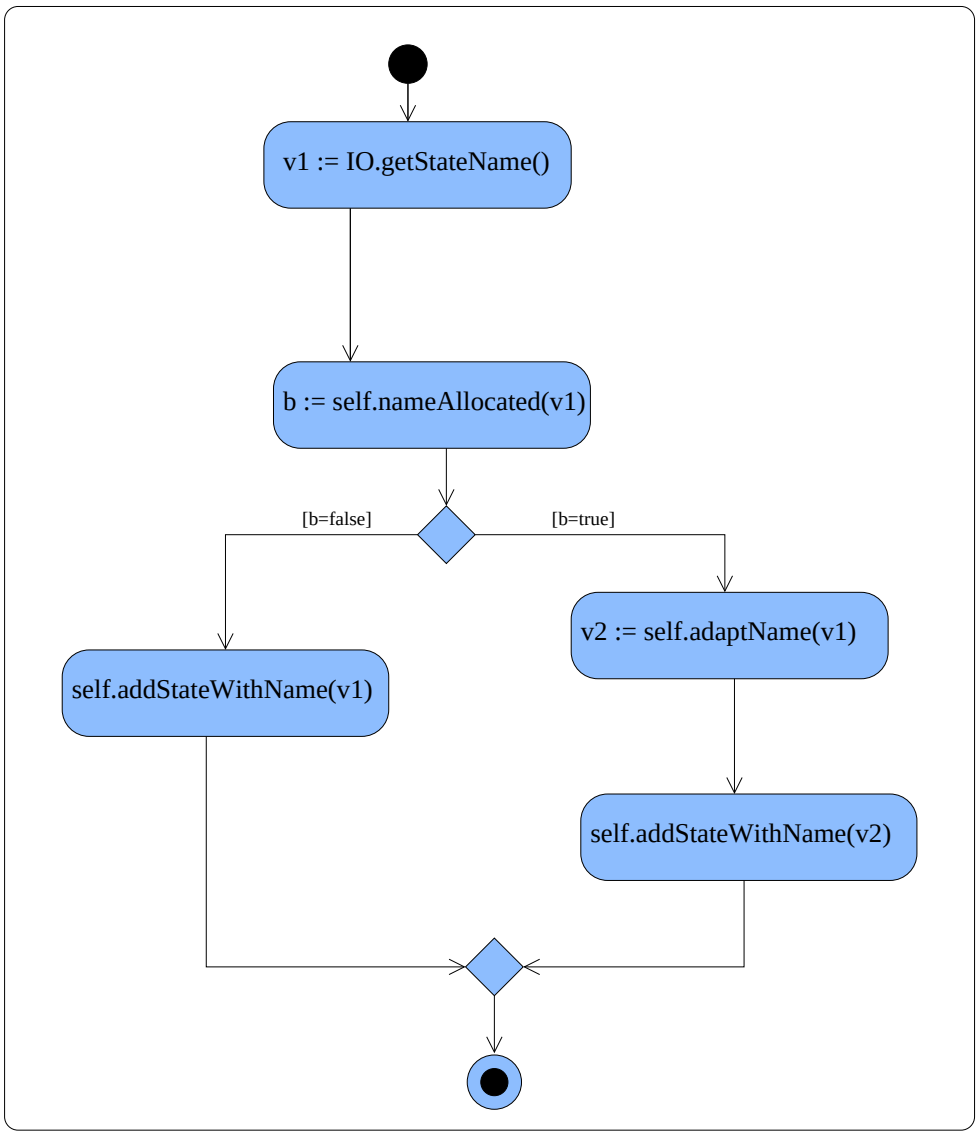

Figure 3.9: L3 Method: Automaton.addState

3.3 L3 Expression

The abstract syntax of L3 Expressions slightly differs from that of value specifications in CUML. As can be seen from figure 3.10, the abstract superclass of expressions is *L3Expression*. Any expression is a tree structure, wherein the children of an element are identified via the *operands* link end, and the parent of an element is identified via the *parent* link end.

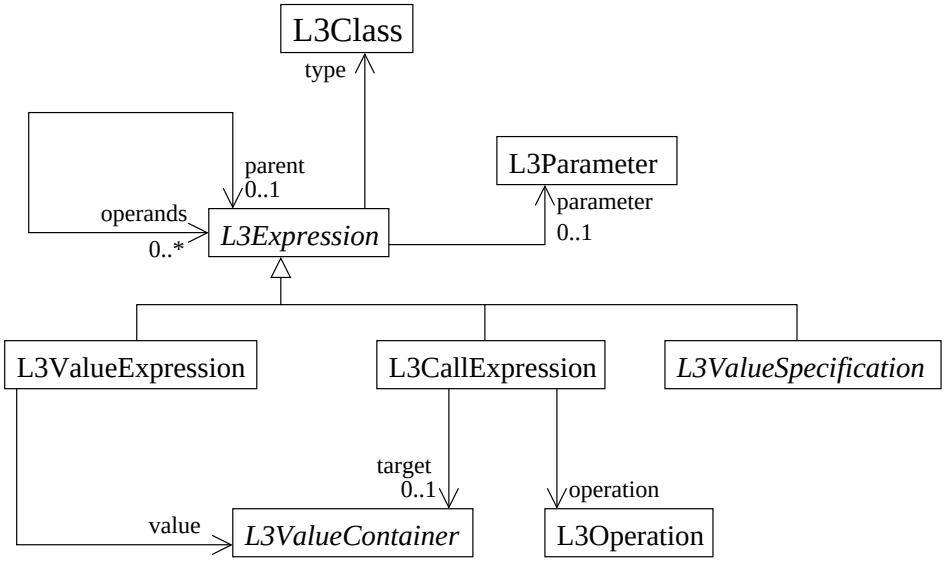

Figure 3.10: L3 Expression

An instance of *L3ValueExpression* refers to a L3 value container (see 3.2). Operations can be called within expressions via *L3CallExpression*. A call expression identifies the value container containing the target object, if it refers to a non-static operation. If the parent of an expression is a call expression, the former provides values for the arguments needed for method execution, and therefore must be linked to the corresponding parameter of the operation. This leads to the constraints below:

- If linked to a static operation, a call expression identifies no target object.

```
context L3CallOperation inv:

 self.operation.static implies
 self.target->size() = 0
```

- If linked to a non-static operation, a call expression identifies the target object.

```
context L3CallOperation inv:

 not(self.operation.static) implies
 self.target->size() = 1
```

- If the parent of an expression is a call expression, a corresponding parameter of the called operation must be specified.

```
context L3Expression inv:

 self.parent->oclIsTypeOf(L3CallExpression) implies
 self.parent.operation.parameter->includes(self.parameter)
```

- Values for all parameters must be provided for call expressions.

```
context L3CallExpression inv:

 self.operands->collect(parameter) = self.operation.parameter
```

- The operation of a *L3CallExpression* has to be executable on the target of the *L3CallExpression*.

```
context L3CallExpression inv:

 self.target.type.canExecute(self.operation)

L3Class::canExecute(o: L3Operation): boolean;
canExecute=
 if self.operations.includes(o)
 then true
 else if self.super->oclIsTypeOf(Object)
  then if self.super->collect(canExecute(o)).includes(true)
   then true
   else false
   endif
  else false
  endif
 endif
```

Any value container that is referred to in an expression has to be accessible within the method. This leads to the following constraints:

- If an *L3Attribute* is value of a *ValueExpression*, it has to be accessible within the enclosing *L3Method*.

```
context L3Method inv:

 self.nodes->select(oclIsTypeOf(L3AssignmentNode))
 ->collect(supplier->getAttributes())
 ->union(self.edges->collect(guard->getAttributes()))
 ->forAll(a | self.operation.class.accessible(a))

L3Expression::getAttributes(): Bag(L3Attribute);
getAttribute=
 if self.value.oclIsTypeOf(L3Attribute)
 then self.value
 else self.operands->collect(getAttributes())
 endif
```

- If a *L3Variable* is value of a *ValueExpression*, it has to be a variable of the enclosing *L3Method*.

```
context L3Method inv:

 self.nodes->select(oclIsTypeOf(L3AssignmentNode))
 ->collect(supplier->getVariables())
 ->union(self.edges->collect(guard->getVariables()))
 ->forAll(v | self.variables.includes(v))

L3Expression::getVariables():  Bag(L3Variable);
getVariables=
 if self.value.oclIsTypeOf(L3Variable)
 then self.value
 else self.operands->collect(getVariables())
 endif
```

- If a *L3Argument* is value of a *ValueExpression*, it has to be an argument of the enclosing *L3Method*.

```
context L3Method inv:

 self.nodes->select(oclIsTypeOf(L3AssignmentNode))
 ->collect(supplier->getArguments())
 ->union(self.edges->collect(guard->getArguments()))
 ->forAll(a | self.arguments.includes(a))

L3Expression::getArguments():  Bag(L3Argument);
getAttribute=
 if self.value.oclIsTypeOf(L3Argument)
 then self.value
 else self.operands->collect(getArguments())
 endif
```

Figure 3.11 shows the abstract syntax of L3 value specifications. Value specifications allow the specification of values of the primitive data types allowed in a CUML model. According to [6], the primitive data types in CUML are instances of the UML metaclass *PrimitiveType*. For L3 I assume wrapper classes for the primitive types, as mentioned before, so in an L3 model the concrete subclasses of *L3ValueSpecification* have an instance of *L3Class* as their type. This is just a temporary workaround, because the issue of primitive data types is out of scope and left for future work.

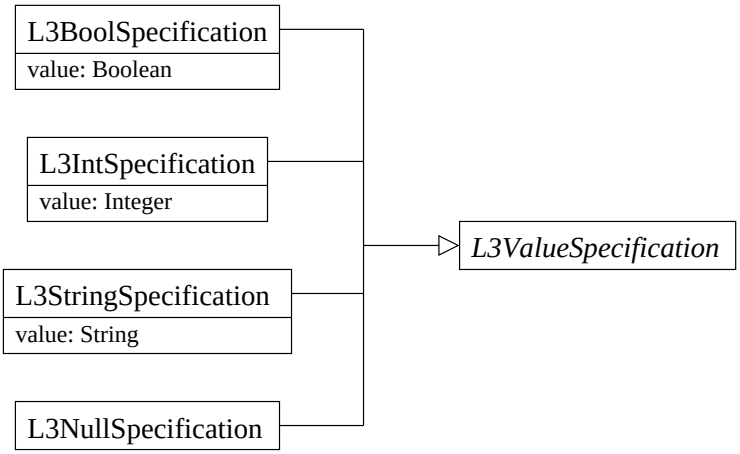

Figure 3.11: L3 ValueSpecification

Looking back at the pedestrian modelling of collection iterators in CUML Activities in Section 2.3, L3 benefits from the explicit collection types at this point. Figure 3.12 shows the abstract syntax of *L3IterateExpression*. An iterator expression is linked to the L3 Variable that is the iterator (i.e. the attribute *isIterator* has the value **true**). An iterator variable can be created using the *L3IterateCreate* expression, that identifies the collection the iterator refers to. During iteration the next value is provided by an instance of *L3IterateNext*. An *L3IterateHasNext* expression can be used to query if there are elements of the collection left for iteration. The example in Figure 3.17 contains iterator expressions.

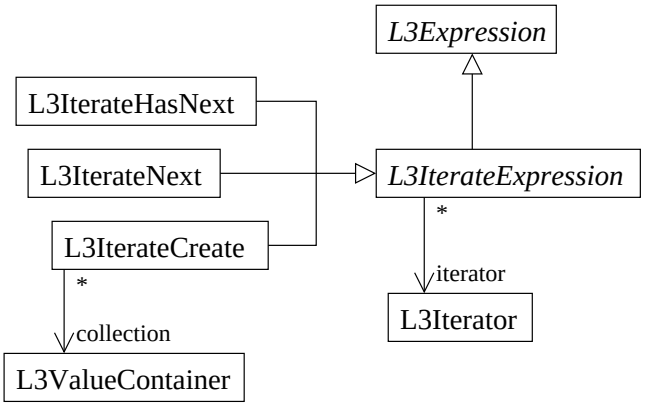

Figure 3.12: L3 Iteration Expressions

Since L3 features collection types, it has to provide access to collections. L3 collec-

tions can be manipulated via instances of *L3CollectionExpression*. Collection expression identify the collection that is accessed via the *collection* link end. The *value* link end indentifies the value that is added to the collection or removed from the collection respectively. If a value is added to an ordered collection type, i.e. *Sequence* or *OrderedSet*, the insertion point can be specified. It would be nice for L3 to feature insertion points for the removal of values from collections, too, but rather useless in this context, because CUML lacks this feature, as I already mentioned in Chapter 2. The L3 Method example in Figure 3.16 includes L3 Collection Expressions.

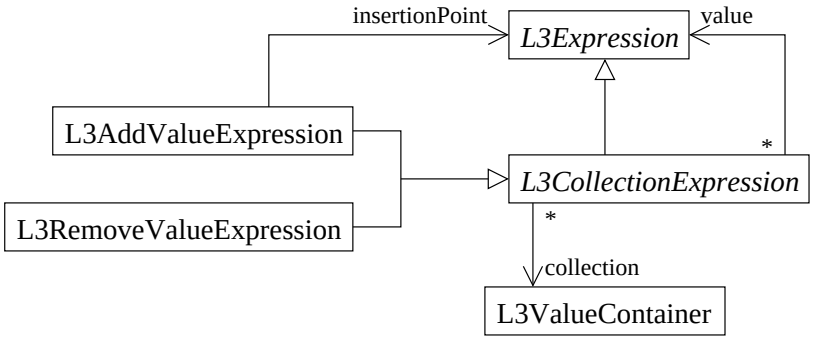

Figure 3.13: L3 Collection Expressions

The abstract syntax diagram in Figure 3.14 shows, that L3 expressions include an expression for the else trace of decision nodes.

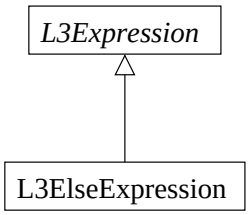

Figure 3.14: L3 Else Expressions

The abstract syntax in Figure 3.15 shows the metaclass *L3ComparisonExpression*. This expression compares the values that result from evaluation of its 2 operand expressions. Evaluation of a comparison expression is **true**, if the values are equal, and **false** otherwise.

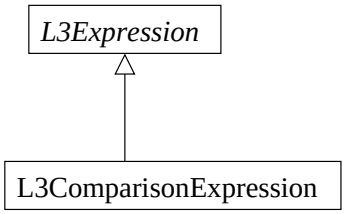

Figure 3.15: L3 Comparison Expressions

The following constraint ensures that a comparison expression has 2 operand expressions:

- An instance of *L3ComparisonExpression* has 2 operand expressions.

```
context L3ComparisonExpression inv:

 self.operands->size() = 2
```

Figure 3.17 shows the L3 Method yielded by model transformation from the CUML Activity example in figure 2.66. It features iterating over collections via L3 Expressions.

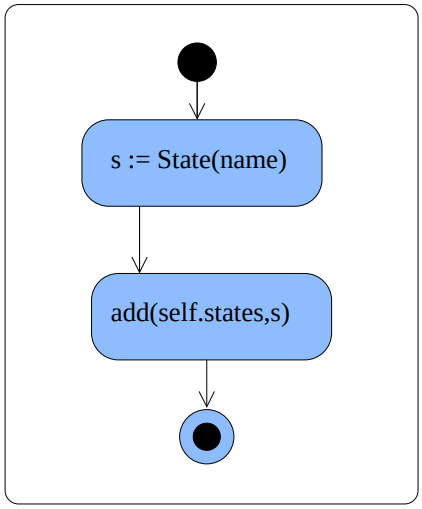

Figure 3.16: L3 Diagram: Automaton.addStateWithName

The L3 Method in Figure 3.16 features L3 Collection Expressions.

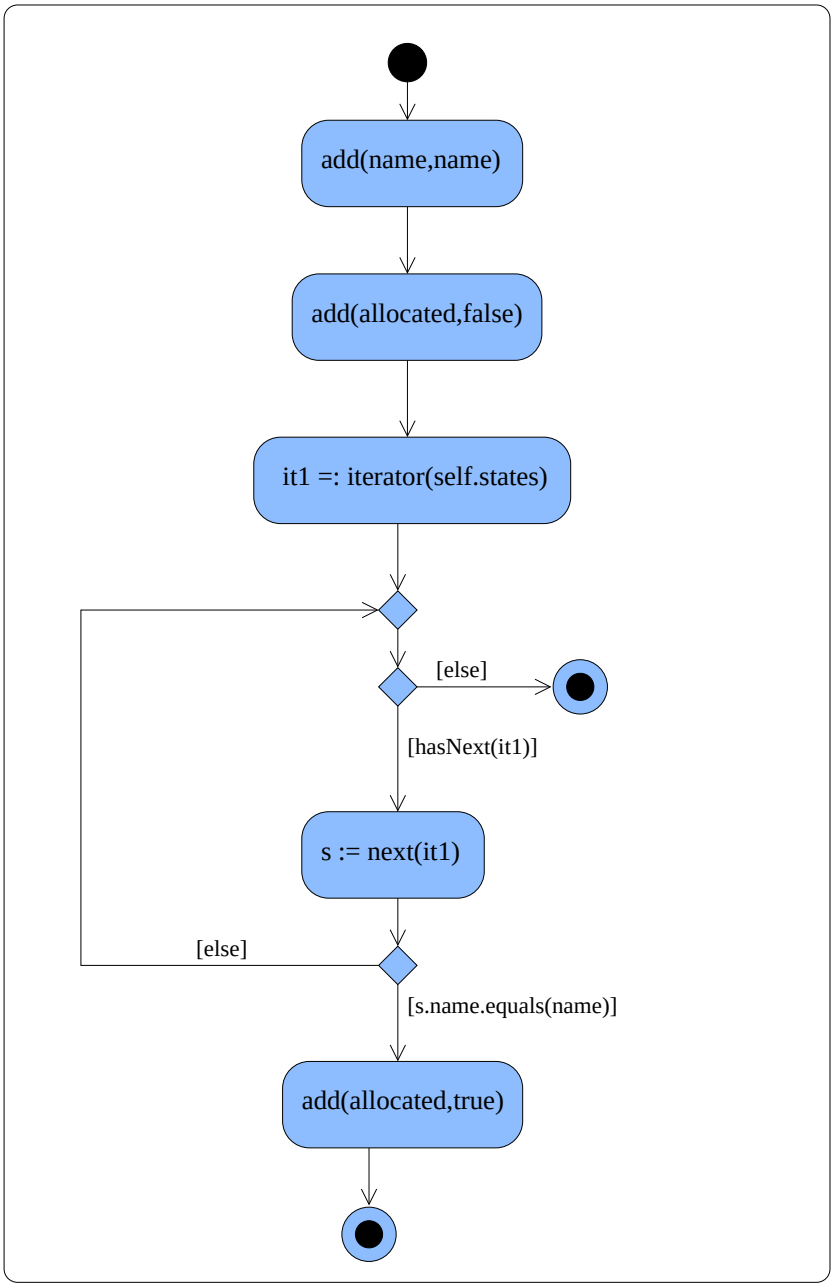

Figure 3.17: L3 Diagram: Automaton.nameAllocated

4 Model Transformation CUML → L3

In Chapter 2 I introduced the UML subset CUML that allows full specification of object-oriented software systems. Since the semantics of CUML inherited from UML is defined in an intuitive and informal way in [6], I introduced the Low Level Language (L3) in Chapter 3 that will serve as a semantical domain for CUML. Methodically, any CUML model will be translated into an L3 model, on which model checking and code generation works. To achieve this goal, I will introduce a model transformation from CUML to L3 by graph transformation in this chapter.

In the first Section 4.1 I will introduce another meta model: the Intermediate Structure. This meta model includes both the CUML and the L3 meta model and serves as type graph for the model transformation. The model transformation itself is given by the graph transformation system in Section 4.2. This chapter contains no analysis of the Graph Transformation system concerning termination and confluence, since this would be out of scope of this thesis. The main purpose of the model transformation here is to show how the features of CUML are mapped to L3. Such an analysis may become necessary for implementation of the model transformation.

A short introduction to transformation of MOF models by graph transformation is given in 1.1.

4.1 Intermediate Structure

As I pointed out before in the introduction of Chapter 3, the structure of the L3 meta model being very close to the structure of the CUML meta model allows tracing of L3 model elements back to CUML model elements. The Intermediate Structure consists of meta model elements that relate the CUML meta model elements and the corresponding L3 meta model elements.

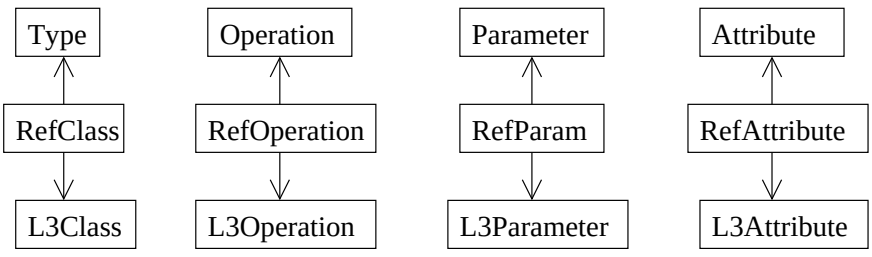

Figure 4.1: Intermediate Structure: Class Structure References

The abstract syntax in Figure 4.1 shows how CUML metaclasses are related to the metaclasses of the L3 Class Structure.

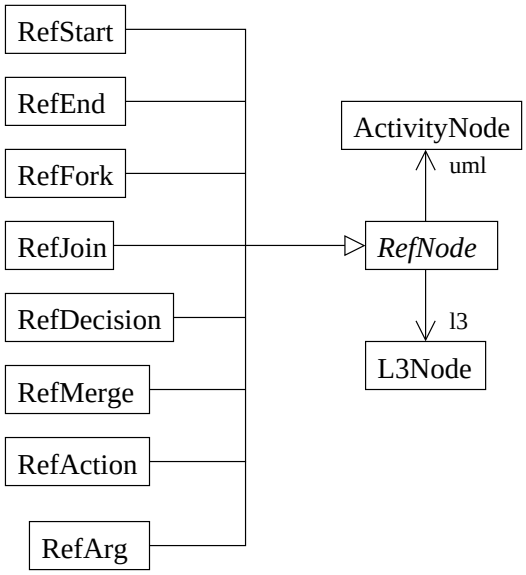

Figure 4.2: Intermediate Structure: Node References

The abstract syntax diagram in Figure 4.2 shows the abstract metaclass *RefNode* that relates CUML Activity nodes the correponding L3 Method nodes. Looking at the concrete subclasses of *RefNode* it becomes obvious that not for every CUML model element there is an corresponding L3 model element: While the metaclass *RefArg* is used to identifiy instances of *ActivityParameterNode* as source of L3 Arguments, there is no element in the Intermediate Structure for another kind of CUML object nodes: the pins of CUML Actions have no corresponding concept in L3, because the flow of data tokens in CUML is translated to variables.

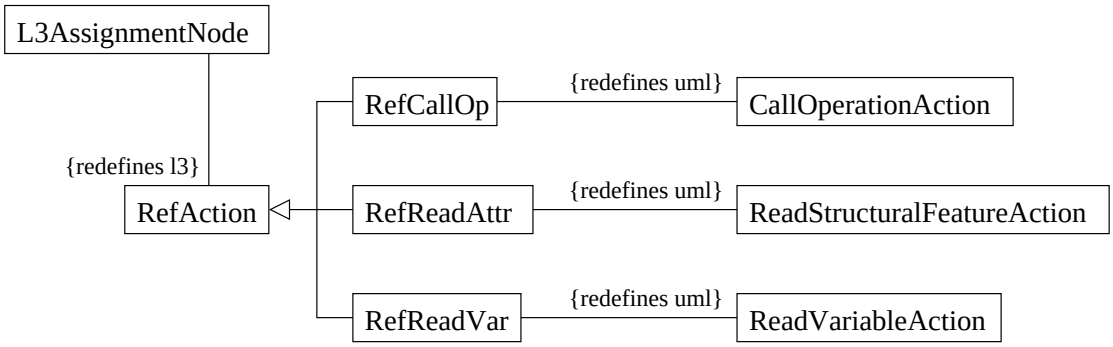

Figure 4.3: Intermediate Structure: Actions with Output

The abstract syntax in Figure 4.3 shows that the association ends *RefNode::uml* and *RefNode::l3* are redefined for the concrete subclasses of *RefNode*. Further this is an example for more than one CUML metaclass being mapped to the same L3 metaclass. Since all CUML metaclasses shown in the diagram can yield values that have to be stored in a variable or similar, they are mapped to the metaclass *L3AssignmentNode*.

The complete Intermediate Structure can be found in the appendix in Section 6.3.1.

4.2 Graph Transformation System

The Transformation System specifying the model transformation from CUML to L3 is organized in the following 3 levels[1]:

- **Level 0 - Preparation of CUML Models:** The structure of CUML Activities is manipulated, i.e. sequence nodes are eliminated, because in L3 there is no concept for encapsulated substructures of methods.
- **Level 1 - Generation of L3 Class Structure:** The transformation of CUML Class Diagrams into the L3 Structure.
- **Level 2 - Generation of L3 Methods:** The transformation of CUML Activities into L3 Methods.
- **Level 3 - Generation of L3 Expressions:** The transformation of CUML Value Specifications into L3 Expressions.

[1]The notion 'level' must not be mixed up with the notion 'layer', which has a certain meaning in the context of Graph Transformation. The levels I used just for grouping of the many rules.

Since the transformation system consists of many rules, not all of them will be discussed in this chapter. In the following I will give a slight overview for every level. During the discussion of a level I will pick out the rules that I consider the most interesting ones. The transformation system as a whole can be found in the appendix in Section 6.3.2.

4.2.1 Level 0 - Preparation of CUML Models

Since the low level concept of L3 Methods does not feature substructures similar to sequence nodes in CUML, activities are flattened by eliminating sequence nodes before the creation of the L3 model. Since the CUML model is changed during the model transformation, the transformation has to work on a copy of the CUML model.

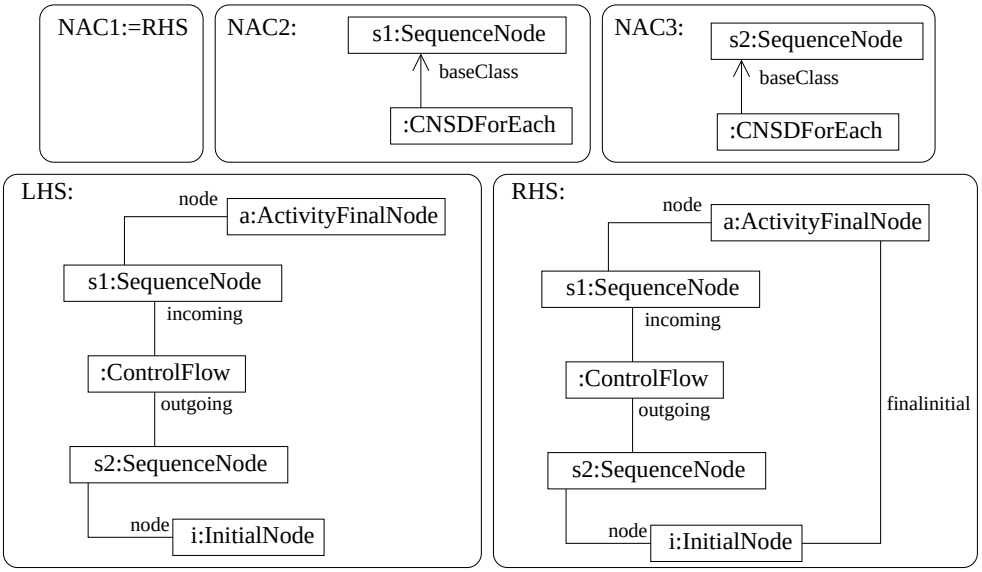

Figure 4.4: Rule: connectFinalInitial

CUML sequence nodes appear in a model as encapsulation of the abstract syntax of CUML Nassi-Shneiderman diagrams (CNSDs) only. As I mentioned before in Chapter 2, directly contained nodes of CNSDs are instances of *SequenceNode*, *InitialNode* or *ActivityFinalNode*. So the elimination of sequence nodes on this level is achieved by replacing control flow between sequence nodes by new control flow from the inner structure of the predecessor sequence node to the inner structure of its successor sequence node (see the rules in figure 4.4 and Figure 4.5).

The Negative Application condition of rule 'connectFinalInitial' (Figure 4.4) shows that a sequence node extended by the stereotype *CNSDForEach* is an exception [2]: If a sequence node is the abstract syntax of a CNSD foreach statement, it is not translated

[2]Namely an exception in the natural language meaning of the word.

into L3 by translation of its inner structure and the identifying encapsulating sequence node is needed in Level 2. Therefore rule application is forbidden by NAC2 and NAC3 in this case. Execution of the rules in the correct order is ensured by the link *finalInitial* between the inner structure of sequence nodes that are connected by control flow: the link is created by the rule 'connectFinalInitial' and is required in the LHS of rule 'createConnection'.

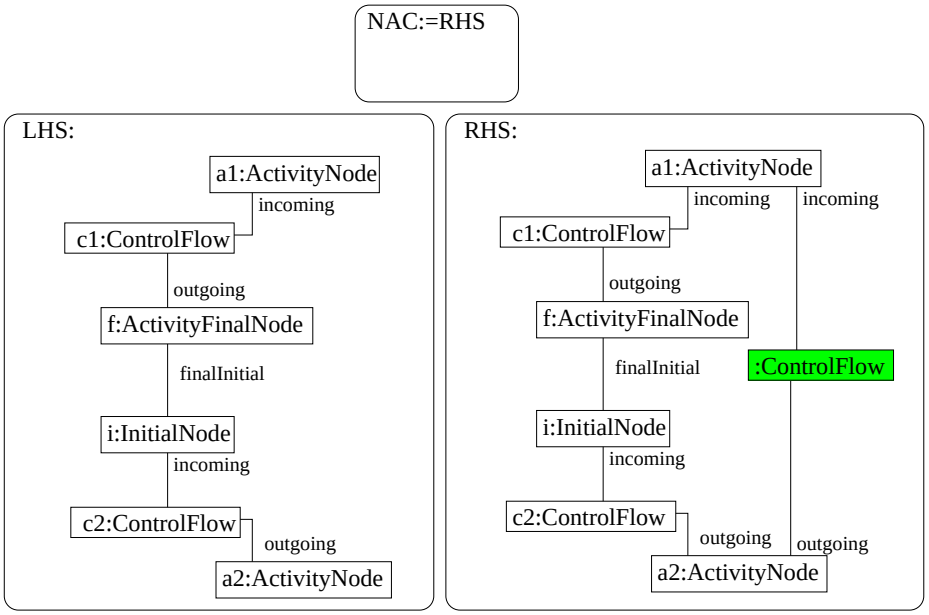

Figure 4.5: Rule: createConnection

Similar rules for sequence nodes that are connected to the initial node or the final node of a CUML Activity can be found in the appendix in Figure 6.7 and Figure 6.8 respectively.

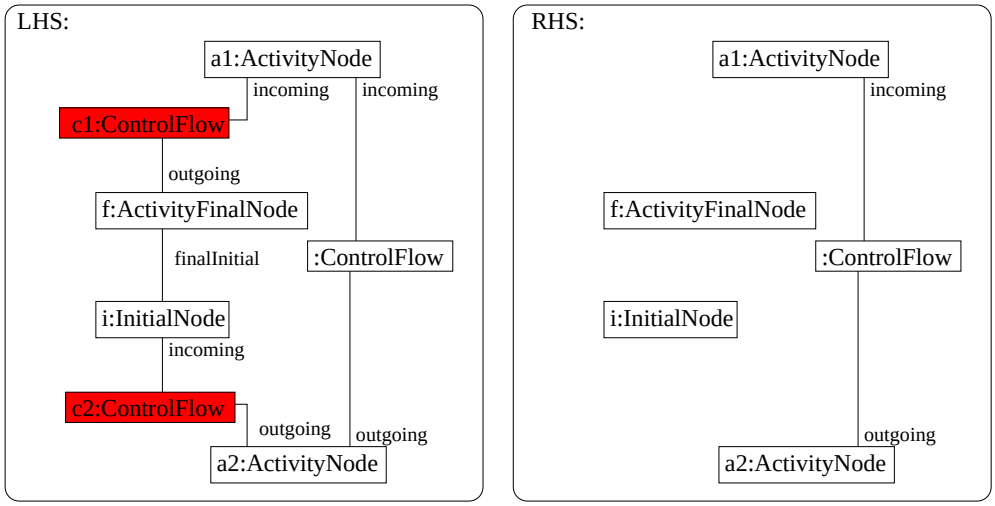

Figure 4.6: Rule: deleteSequenceFlow

If the new control flow between the inner structures is created, the former control flow and the intermediate link *finalInitial* are deleted by the rule 'deleteSequenceFlow' (see Figure 4.6 above).

The complete set of rules contained in this level of the model transformation can be found in the appendix in Chapter 6.3.2: It contains rules for the elimination of sequence nodes that are not directly contained in an activity but in another sequence node and therefore can have instances of *ControlNode* as successor or predecessor.

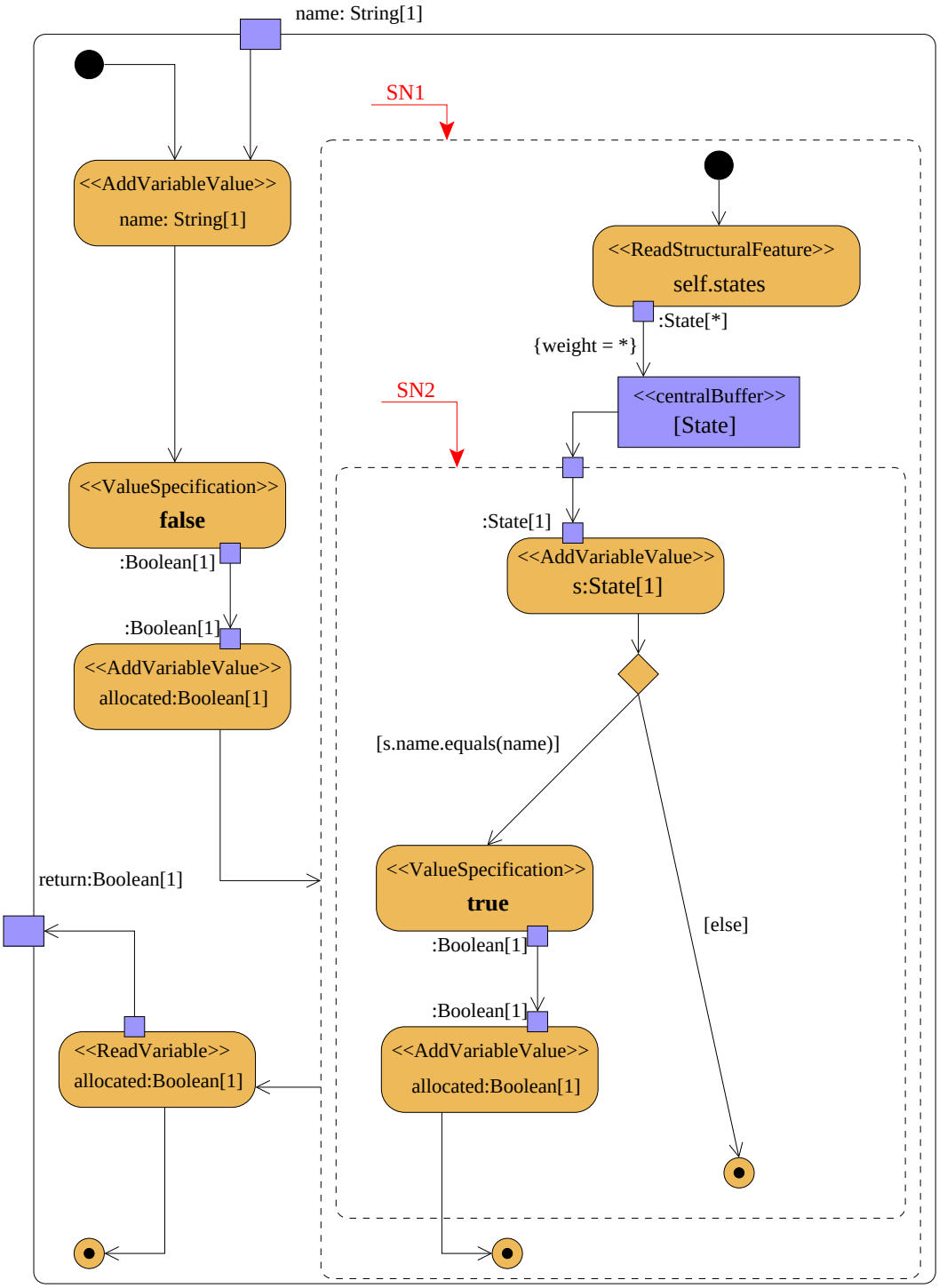

Figure 4.7: L3 Diagram after Level 0: Automaton.nameAllocated

Figure 4.7 shows the CUML Activity Automaton.nameAllocated after application of the Level 0 rules. There are 2 sequence nodes left in the activity: SN1 has not been eliminated because of NAC2/NAC3 of rule 'connectFinalInitial'. SN2 has not been eliminated, because there are no rules for eliminating sequence nodes without incoming and outgoing control flow.

As can be seen from the example, the rules from Level 0 are not only useful for transformation of CUML Activities into L3 Methods, but also for manipulating CNSDs for displaying them in the CUML Activity view.

4.2.2 Level 1 - Generation of L3 Class Structure

This level consists of the rules that create the L3 class structure from the abstract syntax of CUML Class Diagrams.

First of all, the class structure object has to be created by application of the rule 'createClassStructure' (see figure 6.15 in the appendix). When the class structure object exists, CUML packages that are not nested within other packages (i.e. root packages of CUML namespaces) can be created and linked to the class structure by the rule 'createPackage' (Figure 6.16). Nested packages are translated by the rule 'createNestedPackage' (Figure 6.17). When the corresponding L3 package exists for a CUML package, all its classes can be created by application of the rule 'createClass'.

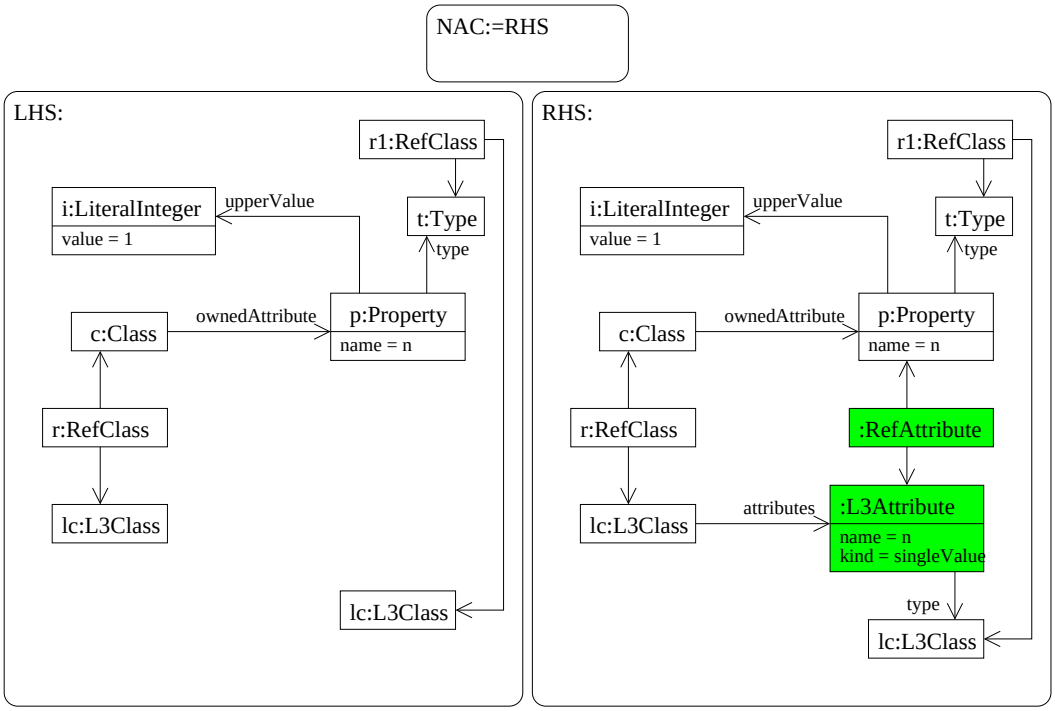

Figure 4.8: Rule: createAttribute

When the L3 classes exist, their features can be created. When attributes are created, the implicit collection types of CUML have to be translated into the explicit collection types of L3. The rule 'createAttribute' in Figure 4.8 creates an L3 attribute for a CUML attribute with upper multiplicity bound **1**, i.e. a non-collection attribute.

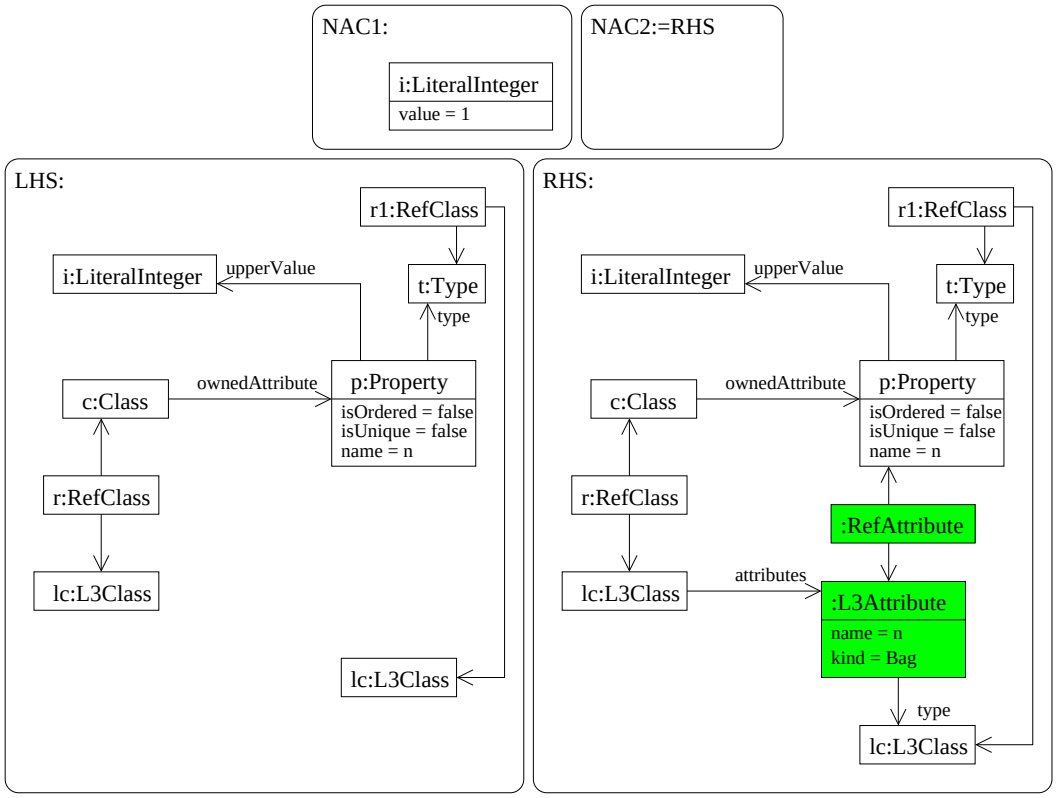

Figure 4.9: Rule: createAttributeBag

Further rules exist for the creation of attributes of a collection type. Figure 4.9 shows the rule 'createAttributeBag' for creating an L3 attribute of the collection type *Bag*. The rules for creating attributes of the collection types *Sequence*, *Set* and *OrderedSet* can be found in the appendix (see figures 6.19, 6.20, 6.21).

CUML operations are translated into L3 operations by application of the rule 'createOperation' (see figure 6.22). When an L3 operation exists, the parameters of the corresponding CUML operation can be translated. This requires the translation of CUML collection types into L3 collection types again. Therefore the transformation system contains 5 rules for the creation of L3 parameters: 'createParameter' for CUML parameters of non-collection type (see figure 6.23), and the rules 'createParameterBag' (Figure 6.24), 'createParameterSequence' (Figure 6.25), 'createParameterSet' (Figure 6.26) and 'createOrderedSet' (Figure 6.27) for CUML parameters of one of the collection types. If

the CUML parameter is of the *in* direction kind, the *isOut* attribute of the corresponding L3 parameter is set to **false** by application of the rule 'setParamIn' (see Figure 6.28).

Similar rules exist for creation of the return parameter of L3 methods (figures 6.29 - 6.33).

4.2.3 Level 2 - Generation of L3 Methods

After the translation of CUML class diagrams to the L3 class structure, the L3 methods can be created using the rules from Level 2.

L3 methods are created for existing L3 operations by aplication of the rule 'createMethod' (Figure 6.34). If a method exists, its arguments can be created from the parameters of the corresponding CUML activity. Luckily there is no need to create the type of an argument (remember creation of the collection types above), because any argument is linked to its corresponding operation parameter. Nevertheless translation of types is necessary when creating the variables of an L3 method: the rules can be found in Figure 6.36 - 6.40.

When an L3 method exists, the control nodes can be created: Start nodes (rule 'createStart', Figure 6.41), end nodes (rule 'createEnd', Figure 6.42), decision nodes (rule 'createDecision', Figure 6.43), merge nodes (rule 'createMerge', Figure 6.44), fork nodes (rule 'createFork', Figure 6.45) and join nodes (rule 'createJoin', Figure 6.46).

CUML actions with output are translated into L3 assignment nodes. Rule 'createAssignNode' in Figure 4.10 translates an instance of a CUML *CallOperationAction* into an assignment node.

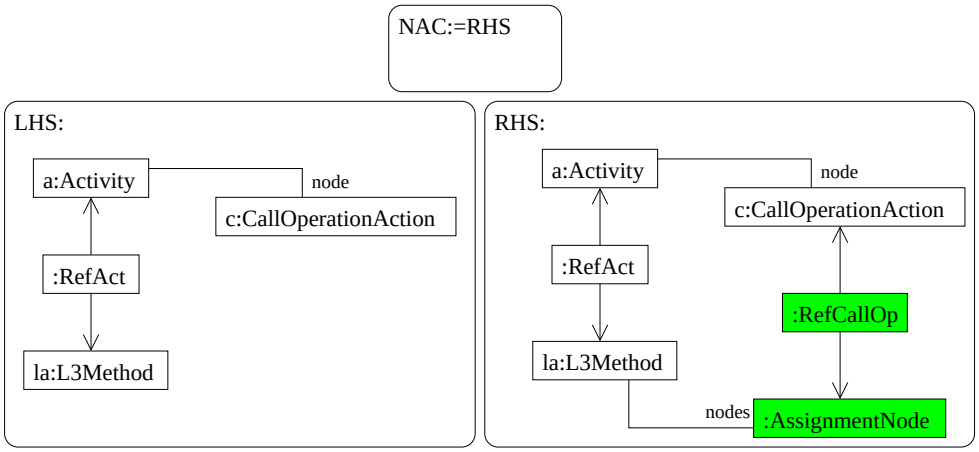

Figure 4.10: Rule: createAssignNode

If the assignment node exists for the operation call, the L3 call expression is created by application of rule 'createSupplier' in Figure 4.11. The call expression will be completed by the rules of Level 3 (Section 4.2.4). The designator is created by application of rule

'createDesignator' (Figure 6.53) for existing assignment nodes.

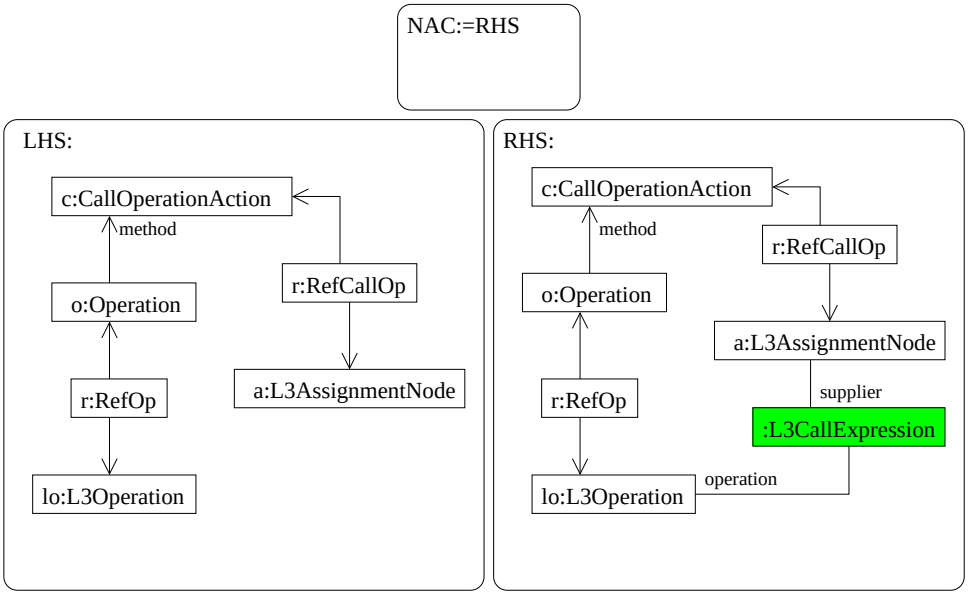

Figure 4.11: Rule: createSupplier

Application of the rules 'createReadFeature' (Figure 6.51) and 'createReadVariable' (Figure 6.47) creates L3 assignment nodes for instances of CUML *ReadStructuralFeatureAction* and *ReadVariableAction* respectively. Instances of CUML *ValueSpecificationAction* are translated to assignment nodes by application of the rule 'createValueSpecification' (see Figure 6.60), instances of *TestidentityAction* are translated by application of the rule 'createTestIdentity' (see Figure 6.61).

The creation of L3 iterators from CNSD foreach statements is a litle more complex. Since CNSD iterators do not have a loop structure consisting of decision/merge nodes, this structure has to be created on the L3 side explicitly. This is done by application of the rule 'createIteratorLoop' (see Figure 4.12): The assignment node delivering the collection is connected to a created L3 merge node. This node merges the control flow of first-time loop entry and the control flow returning from the end of the loop. It is connected to a decision node that branches control flow to the loop body and to the L3 final node that terminates the iteration. Rule 'closeIteratorLoop' (see Figure 6.62) closes the iterator loop by connecting all ends of control flow of the body to the merge node of the loop (see above).

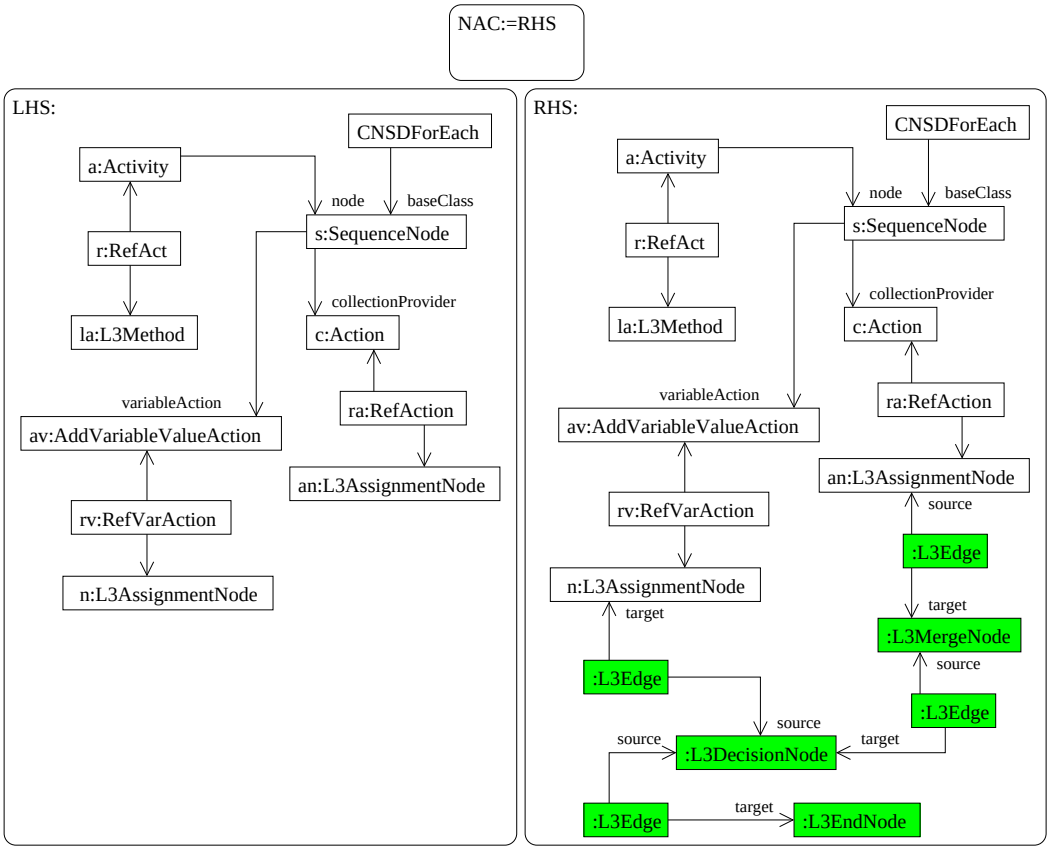

Figure 4.12: Rule: createIteratorLoop

The L3 iterator object is created by application of the rules 'createVariableIterator' (Figure 4.13) for collections provided by variables and 'createAttributeIterator' (Figure 6.63) for collections provided by attributes.

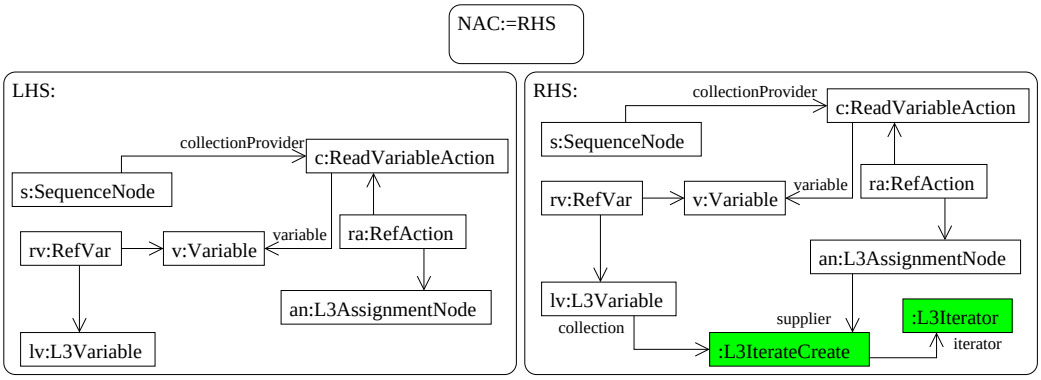

Figure 4.13: Rule: createVariableIterator

The outgoing edges of the decision node created by the rule 'createIteratorLoop' have to be annotated with guard expressions. Rule 'createIteratorHasNext' (Figure 4.14) creates an instance of *L3IteratorHasNext* as guard for the outgoing edge leading to the iteration body. The edge leading to the L3 end node terminating the iteration is guarded by an L3 else expression.

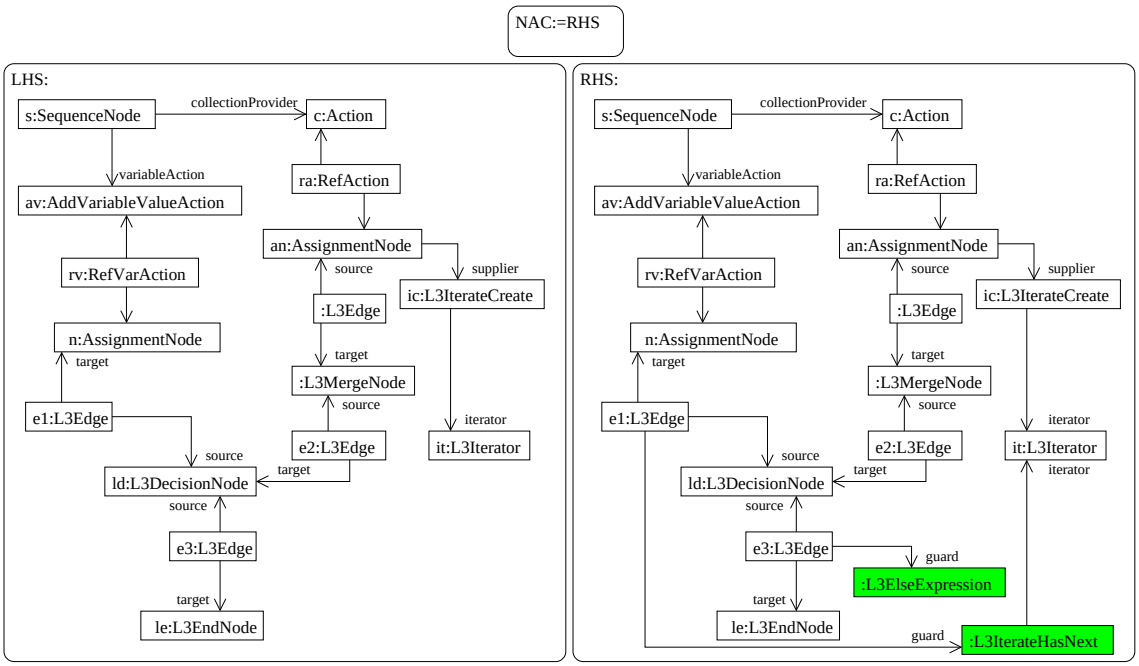

Figure 4.14: Rule: createIteratorHasnext

Finally, the L3 assignment that assigns a value of the collection to the iteration variable before each iteration has to be created. Rule 'createIteratorNext' (Figure 4.15) connects the existing L3 assignment node that corresponds to the instance of *AddVariableValueAction* on the CUML side that assigns the iteration variable to the corresponding L3 variable and a created instance of *L3IterateNext*.

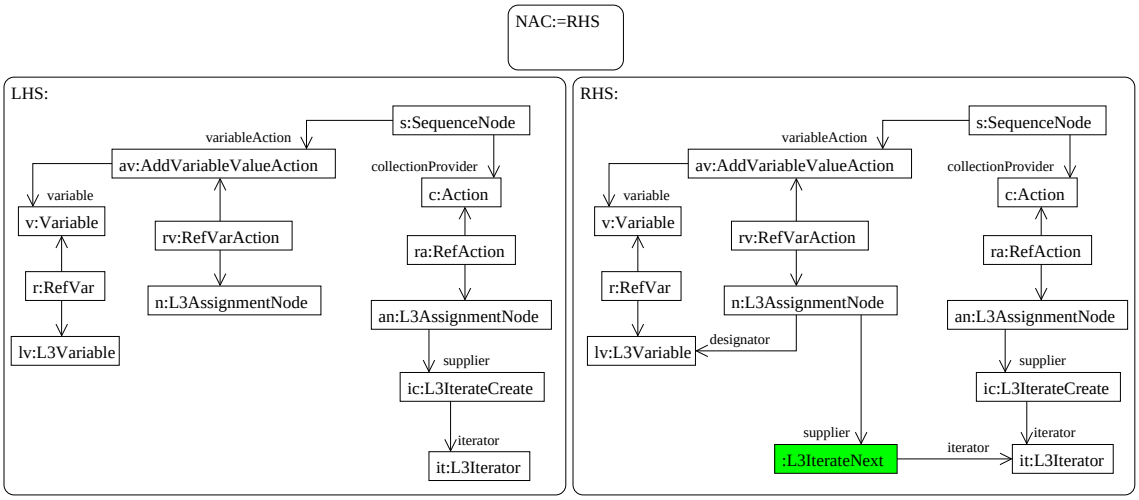

Figure 4.15: Rule: createIteratorNext

4.2.4 Level 3 - Generation of L3 Expressions

Level 3 consists of the rules that create L3 expressions. Most of the CUML actions are translated to L3 expressions.

The CUML actions that allow manipulation of local variables of activities and attributes of objects are translated to the concrete subtypes of *L3CollectionExpression*. There are rules for translating instances of subclasses of *StructuralFeatureAction*: 'createAddValueSFeature' (see Figure 6.57) and 'createRemoveValueSFeature' (see figure 6.59). The rules for translating variable actions are 'createAddValueVariable' (figure 6.54) and 'createRemoveValueVariable' (Figure 6.56).

The CUML literal specifications of values of the primitive data types Boolean, String and Integer[3] are translated to the value specifications of L3 expressions (see Section 3.3). The rules are: 'createBoolSpec' (Figure 6.64), 'createIntSpec' (Figure 6.65), 'createStringSpec' (Figure 6.66) and the rule 'createNullSpec' (Figure 6.67) for the specification of the absence of any value.

[3]The UML profile for CUML ensures that in a CUML model values of the type *UnlimitedNatural* are used for structural concerns only, i.e. upper multiplicity bounds, capacity of object nodes, weight of activity edges etc.

CUML expressions that are UML expressions extended by one of the CUML stereotypes are of course translated to L3 expressions as well: If a CUML expression is extended by the stereotype *ValueExpression*, application of the rule 'createValueExpFromVariable' (Figure 4.16) - if the CUML expression refers to an activity variable - or the rule 'createValueExpFromProperty' (Figure 6.68) - if the expression refers to a property - leads to creation of an L3 value expression.

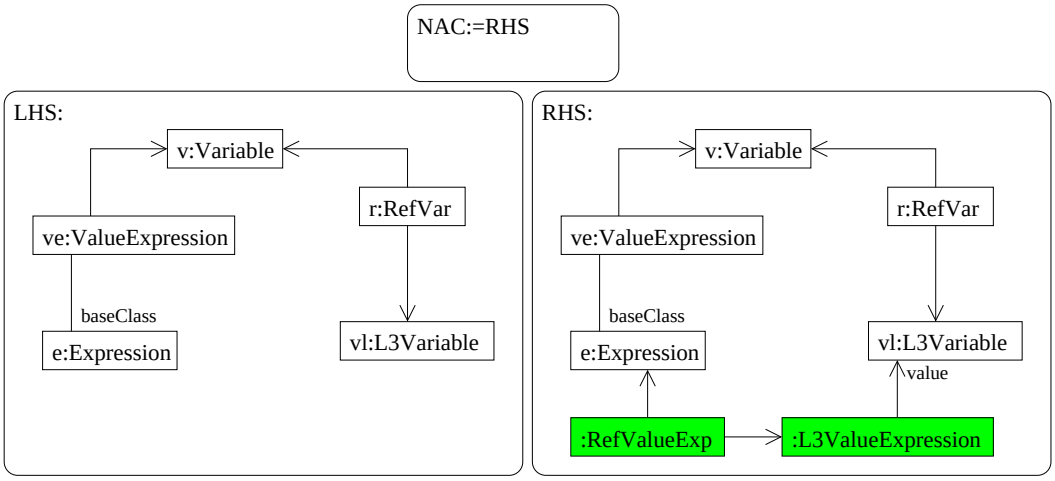

Figure 4.16: Rule: createValueExpFromVariable

If a CUML expression represents the call of an operation it is extended by the stereotype *CallExpression*. These expressions are translated to L3 call expressions. There are three rules that create call expressions: 'createCallExpFromProperty' (Figure 4.17) is applicable if the target object of the operation call is accessed via an attribute.

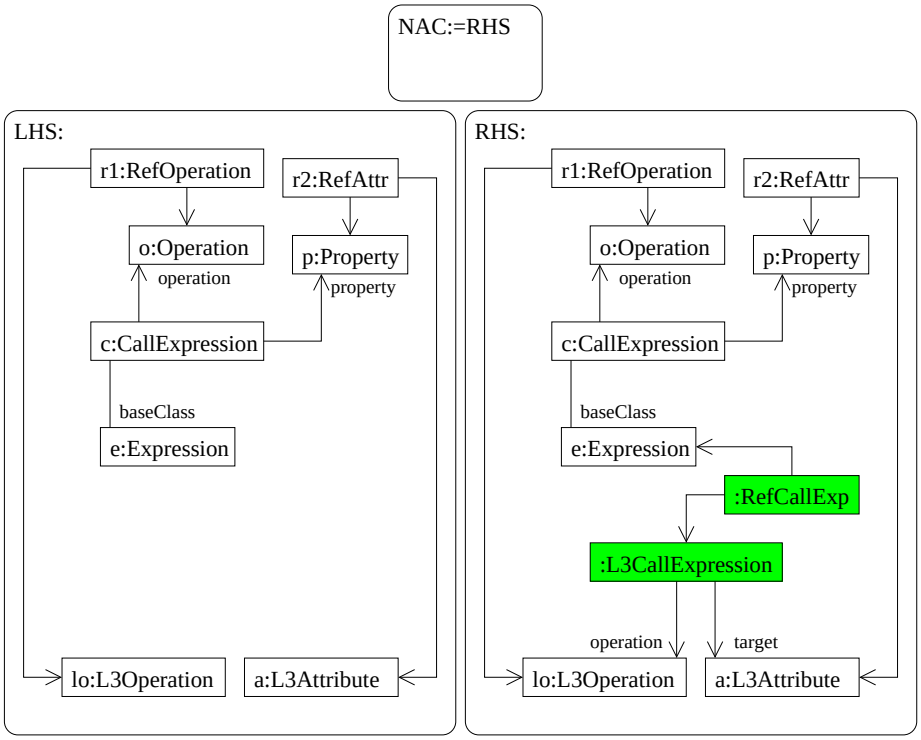

Figure 4.17: Rule: createCallExpFromProperty

If the operation is called on a variable, rule 'createCallExpFromProperty' (Figure 4.17) is applicable. Rule 'createCallExpStatic' (Figure 6.70) creates calls for static operations.

If a child of a call expression in a CUML expression tree is translated to L3, the structure that identifies the child as an argument of the operation call can be created on the L3 side: this is done by application of the rule 'createArgumentExp' (Figure 4.18). This rule makes use of the stereotype *ArgumentExpression*, that makes explicit the relation between the child expression of a call expression and a parameter of the operation that is called.

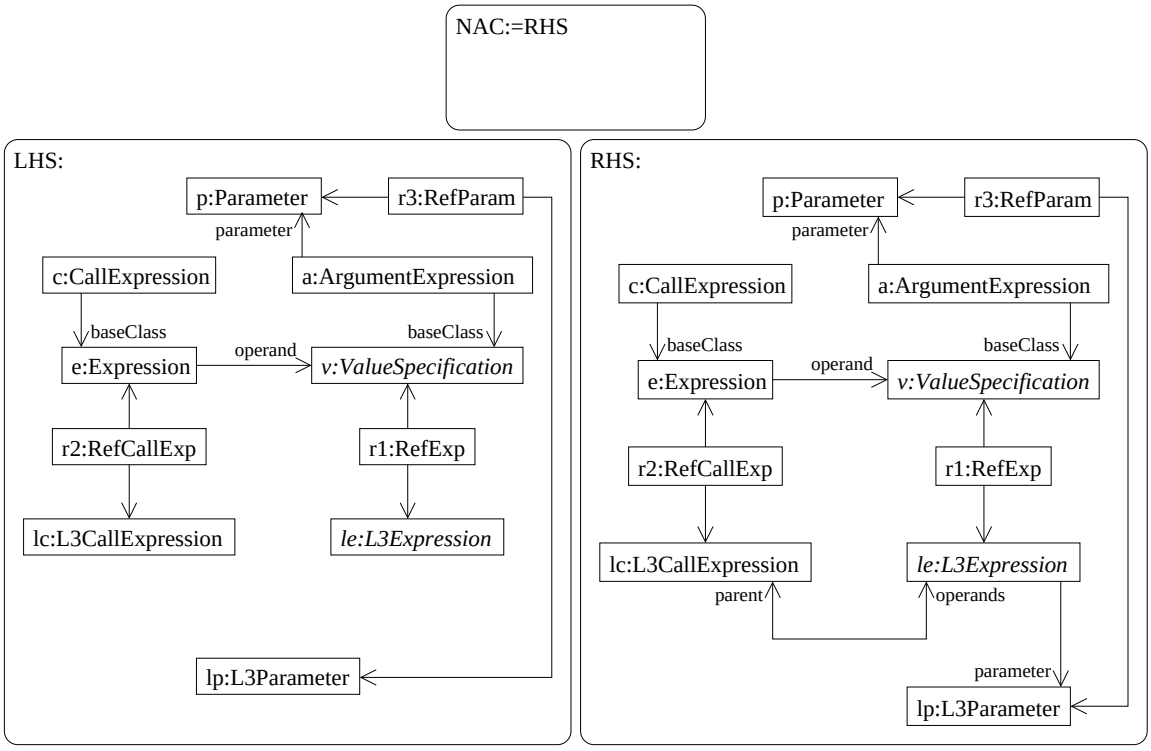

Figure 4.18: Rule: createArgumentExp

But L3 call expressions can be the result of the translation of an instance of a CUML *CallOperationAction* as well. In this case the values for the arguments of the operation call are provided by the data flow of the enclosing activity. Rule 'createActionArgument' (see Figure 4.19) creates an instance of *L3ValueExpression* that relates the corresponding L3 variable of this data flow to the correct L3 parameter of the called operation.

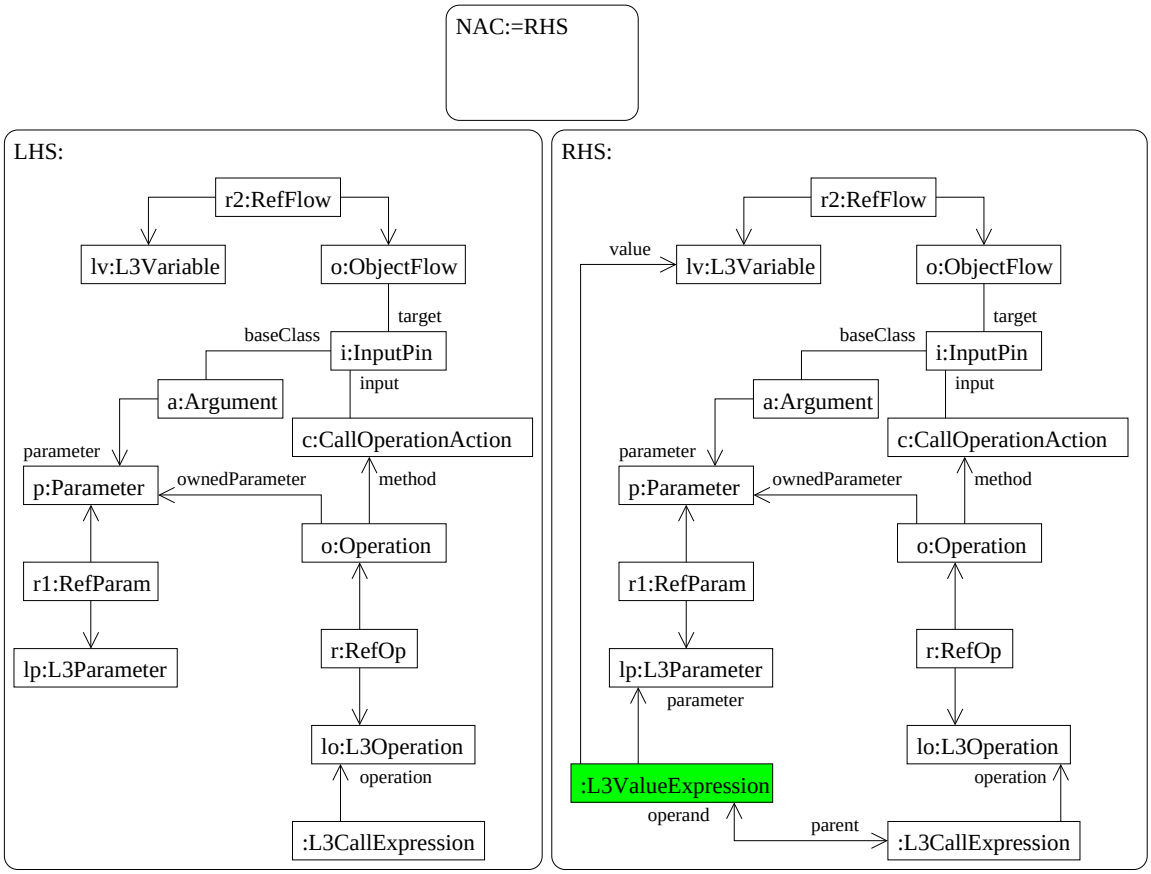

Figure 4.19: Rule: createActionArgument

Level 3 contains the rules for translating control and data flow of CUML activities. Rule 'createEdge' (figure 6.48) creates an L3 edge for every instance of *ControlFlow*. Since in CUML activities there can be implicit control flow - an action starts execution when all input pins hold a number of tokens that allow execution, even if there is no incoming control flow - this control flow has to be made explicit in L3. This is done by application of the rule 'createImplicitControl' (see Figure 4.20).

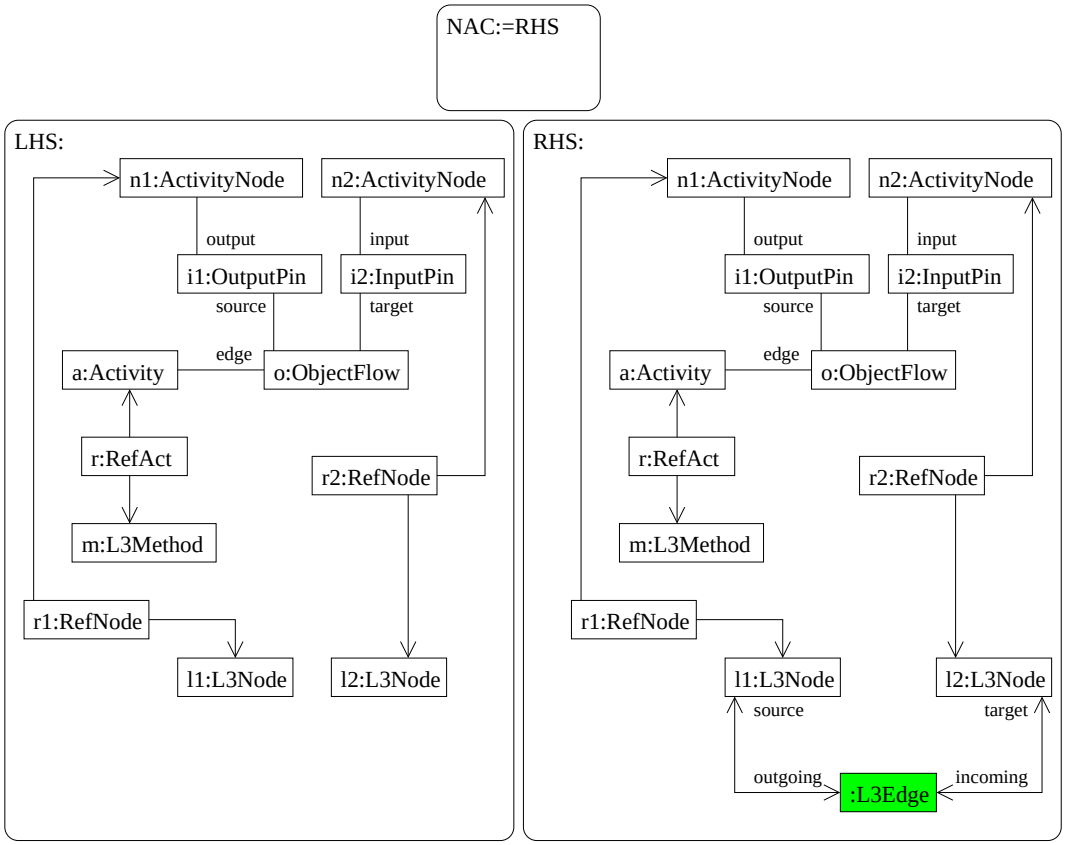

Figure 4.20: Rule: createImplicitControl

The creation of explicit L3 control flow from implicit CUML control flow may lead to L3 method nodes having more than one incoming edge. Since this may occur even without application of rule 'createImplicitControl', since CUML, as well as UML, allows implicit merge[4] , elimination of implicit merge is necessary in L3, due to its explicit character. The explicit L3 merge node is created by application of the rule 'createImplicitMerge' (see Figure 6.49). Incoming edges of an L3 node having a merge node as predecessor are connected to this merge node by application of rule 'mergeEdges' (see Figure 6.50).

Since in L3 methods access to objects and values is managed via variables, the object flow of an activity is translated to local variables of the corresponding L3 method by application of the rule 'createVariable' (see Figure 6.47).

[4]Multiple incoming control flows of actions are implicitly merged by an assumed merge node that is target of the control flows and predecessor of the action.

5 Conclusion

The motivation for the approach presented in this diploma thesis is the need for a formally defined complex modelling technique, that integrates different modelling techniques by mapping them to a common semantical domain.

On the one hand with the UML there exists a complex modelling technique for object-oriented software systems, that includes structural modelling techniques like Class Diagrams, constructive techniques like Activities and descriptive techniques like Sequence Diagrams. But the semantics of the UML defined in [6] is incomplete and informal. On the other hand there exist many approaches that define formal semantics for single sublanguages of the UML by mapping to a semantical domain. But a combination of these approaches covering structural, contructive and descriptive techniques would therefore result in a composite semantical domain, whereas the problem of interoperability arises.

Besides verification of models, automated code generation is one of the main purposes of Model Driven Architecture. A common semantical domain for the sublanguages of a complex modelling technique that allows code generation from the models of the semantical domain language could make the development of code generation more convenient.

5.1 Summary

In Chapter 2 I introduced the complex modelling technique CUML. CUML is formally described as a restriction to UML by the UML profile for CUML. This way CUML benefits from the wide tool support for UML. Since my thesis concentrates on behaviour modelling techniques, I introduced just a basic version of Class Diagrams as structural modelling technique of CUML. The behaviour modelling techniques introduced are CActivities as a UML Activities subset, and CUML Nassi-Shneiderman Diagrams (CNSDs). Nassi-Shneiderman Diagrams were first proposed in [15] informally and used as a structured form of pseudo-code since. CNSDs are an enhancement of these and feature concepts of modern programming languages, e.g. iteration over collections. Since CUML is based on the UML, the abstract syntax of CNSDs had to be expressed by means of the UML meta model. So I achieved a syntactical definition for CNSDs by using them as concrete syntax for a subset of UML Activities. This fact emphasizes the role of CNSDs as "syntactical sugar" within CUML. Where CActivities - similar to UML Activities - feature modelling of data and control flow by token flow, CNSDs offer ways of modelling data and control flow that are more convenient for modelers who are used to develop a software system by writing program source code directly. The introduction of an alternative concrete syntax is no violation of the UML standard, since [6] calls the proposed notation for UML Activities optional.

Consisting of Class Diagrams for modelling the class structure of a system and CActivitities and CNSDs for modelling the behaviour by specifying the operations of classes, CUML is a complex modelling technique in the sense I pointed out above. Figure 5.1 shows, that a complete model consisting of CClassDiagrams, CActivities and CNSDs is translated to the semantical domain.

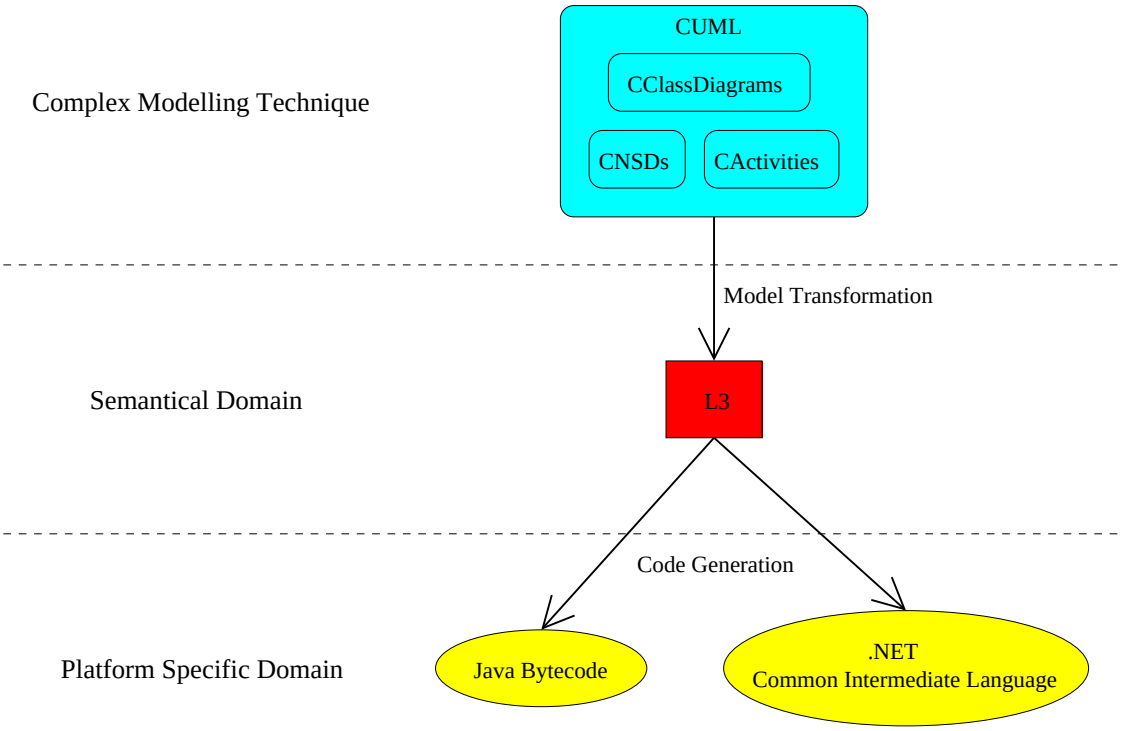

Figure 5.1: From CUML to program code.

In Chapter 3 that introduces the semantical domain language L3 I pointed out 4 main properties of L3: explicit contents, seperation of aspects, low-level concepts and model tracing. These properties make L3 fulfill the requirements of model verification and code generation: Explicit contents ensure that all information contained in the model is easily accessible for verification. Separation of structural, constructive and descriptive aspects structure the model for different verification techniques: The constructive part is checked against the requirements of the descriptive part, both parts communicate over the structural part. Model tracing allows the identification of the corresponding CUML model element for every L3 model element, what is useful for the user feedback of the L3 model verification. Finally, the low-level concepts of L3 make it easy to develop code generation routines from L3 to arbitrary object-oriented platforms. The schema in Figure 5.1 depicts Java and .NET CIL as possible target platforms of code generation.

In Chapter 4 I introduced the model transformation from the complex modelling technique CUML to the semantical domain language L3 by a graph transformation system. This transformation is an endogenous model transformation: The meta model, that is interpreted as the type graph, is an intermediate structure, that imports both the CUML and the L3 meta model. This intermediate structure consists of meta-classes that relate a CUML meta-class and an L3 meta-class, that correspond to eachother in the sense of model tracing mentioned above.

The discussion of the transformation rules has shown, that implicit contents of the CUML model are translated to explicit constructs of the L3 model (e.g. collection types). Further the separation of aspects has become obvious: E.g., during transformation of CUML class diagrams into the L3 class structure I pointed out how the information concerning CUML associations may lead to both structural and descriptive contents of the L3 model.

5.2 Possible Extensions

Since my thesis deals with the syntactical integration of the constructive behaviour modelling techniques of CUML only, namely CActivities and CNSDs, there are a number of possible extensions to CUML. CUML contains just a basic version of class diagrams. A full featured version should cover polymorphy, interfaces, abstract classes and operations, and visibility. Further the constructive part could be completed by adding transformation rules and OCL body constraints for query operations. CUML Activities and CNSDs could be extended to exception handling. Possible techniques for the descriptive part of CUML are UML sequence diagrams and OCL constraints.

Enhancement of CUML class diagrams would lead to corresponding increments of the L3 class structure. Additional constructive CUML techniques can be mapped to L3 methods in their current version introduced here. CActivities and CNSDs featuring exceptions would require an exception concept for L3. For the descriptive contents on the CUML side the corresponding L3 part would have to be developed.

Semantical correctness is out the scope of my thesis. Nevertheless I like to mention that, although the semantics of UML and therefore that of CUML is defined informally by natural language description, semantical correctness of the model transformation CUML $\Rightarrow$ L3 will have to be discussed once formal semantics of L3 are defined.

5.3 Implementation Notes

Finally I like to sketch an implementation for the approach introduced in this thesis as plugin for the Eclipse integrated development environment ([1]). Such an implementation will map to the schema presented in Figure 5.1.

One of the main reasons for defining the complex modelling technique by use of the UML profiling mechanism was to benefit from tool support for UML. The data model for implementation of an Eclipse plugin is typically provided by an instantiation of the Eclipse Modelling Framework (EMF, [2]). The UML2 Eclipse plugin ([5]) provides a

complete UML 2.0 meta model based on EMF. Since the EMF plugin includes a mechanism to create a meta model for UML profiles by use of the UML2 plugin, a meta model for CUML can be easily build from the UML profile for CUML in Section 6.2. The editor for CUML diagrams can be implemented using the Graphical Editing Framework (GEF, [3]), which allows convenient development of graphical editors for the Eclipse IDE.

The meta model for the semantic domain language L3 can be created as an EMF-based model.

An important part of a tool for CUML will be the model transformation from CUML to L3. The Tiger EMF Transformation Project provides a framework for graph-based EMF transformation, so that the transformation specified by the graph transformation system in Section [4] can be easily implemented.

The implementation of the routines for verification of semantical domain models highly depend on the formal verification techniques used. Since EMF provides access to models via Java interfaces, they probably will be implemented in Java.

The implementation of code generation from L3 models should make use of existing tools, e.g. the Java Emitter Templates (JET) framework, a generic template engine that can be used to create source code.

6 Appendix

6.1 German Summary

Obwohl durch die Definition der UML keine formale Semantik vorgegeben ist, existieren viele Ansätze, die eine formale Semantik für Teilsprachen der UML durch Abbildung in einen semantischen Bereich definieren. Trotzdem gibt es keinen solchen Ansatz für eine Menge von UML-Teilsprachen, die die komplette Modellierung objektorientierter Softwaresysteme ermöglicht. Ein solcher Ansatz müsste die semantische Integration verschiedener Verhaltensmodellierungstechniken beinhalten. In dieser Diplomarbeit definiere ich die komplexe Modellierungstechnik CUML, die die Verhaltensmodellierungstechniken CActivities und CNSDs enthält. Diese Techniken werden durch Modelltransformation in einen gemeinsamen semantischen Bereich integriert. Modelltransformation und die Sprache L3, die den semantischen Bereich syntaktisch definiert, sind Teil dieser Diplomarbeit.

6.2 UML-Profile for CUML

6.2.1 Constraints on UML

[1] Every instance of *Classifier* is instance of *Class*.

```
context Classifier inv:
  self.allInstances()->forAll(oclIsTypeOf(Class))
```

[2] Every *Operation* must be owned by a class.

```
context Operation inv:
  self.class->size() = 1
```

[3] Every *Property* must be owned by a class.

```
context Property inv:
  self.class->size() = 1
```

[4] Lower and upper bound of a *MultiplicityElement* have to be expressed as integer and unlimited natural respectively.

```
context Kernel::MultiplicityElement inv:
  self.lowerValue.oclIsTypeOf(LiteralInteger) and
  self.upperValue.oclIsTypeOf(LiteralUnlimitedNatural)
```

[5] Instances of *LiteralUnlimitedNatural* are only allowed as specification of a *MultiplicityElement*s upper bound, the capacity of an *ObjectNode* or the weight of an *ActivityEdge*.

```
context LiteralUnlimitedNatural inv:

  self.allInstances()->forAll(n |
  MultiplicityElement.allInstances()->exists(upperValue = n) or
  ActivityEdge.allInstances()->exists(weight = n) or
  ObjectNode.allInstances()->exists(upperBound = n))
```

[6] Every *Parameter* is owned by an *Operation*.

```
context Parameter inv:

  self.allInstances().operation->size = 1
```

[7] *ParameterDirectionKind* is restricted to: **in**, **inout** and **return**.

```
context Parameter inv:

  self.direction = in or self.direction = inout or
  self.direction = return
```

[8] Every *TypedElement* has a type.

```
context TypedElement inv:

  self.allInstances().type->size = 1 or
  self.oclIsTypeOf(InputPin)
```

[9] There is exactly one *Activity* for every particular pairing of an implementing *Class* and an *Operation*.

```
context Communications::Operation:
inv:  self.class->size() = 1 implies
self.method->select(m | m.oclIsTypeOf(Activity))->size() = 1
```

[10] There are no autonomous activities, i.e. every activity is associated with a classifier as its context.

```
context StructuredActivities::Activity inv:

   self.context->size() = 1
```

[11] Every *Classifier* has at most one generalization.

```
context Kernel::Classifier inv:

  self.generalization->size() < 2
```

[12] Every *SequenceNode* has one *ExecutableNode* of type *InitialNode*.

```
context SequenceNode inv:

  self.executableNode->size() = 1 and
  self.executableNode.oclIstTypeOf(InitialNode)
```

[13] Visibility has to be specified for all *NamedElement*s.

```
context Kernel::NamedElement inv:
  self.visibility->size() = 1
```

[14] The direction of a *Parameter* is either **in** or **return**.

```
context Parameter inv:
  (direction = in) or (direction = out)
```

[15] Every instance of *Type* is instance of *Class*.

```
context Type inv:
  self.allInstances()->forAll(oclIsTypeOf(Class))
```

[16] Instances of *Action* are instances of *CallOperationAction*, *StructuralFeatureAction*, *ValueSpecificationAction*, *TestIdentityAction*, *VariableAction*, *SequenceNode*.

```
context Action inv:
  self.oclIsTypeOf(CallOperationAction) or
  self.oclIsTypeOf(StructuralFeatureAction) or
  self.oclIsTypeOf(ValueSpecificationAction) or
  self.oclIsTypeOf(TestIdentityAction) or
  self.oclIsTypeOf(VariableAction) or
  self.oclIsTypeOf(SequenceNode)
```

[17] A *Variable* must be owned by an *Activity*.

```
context Variable inv:
  self.activityScope->oclIsTypeOf(Activity)
```

[18] The input pins of a *TestIdentityAction* have the type *Object*.

```
context TestIdentityAction inv:
  self.first.type = Object and
  self.second.type = Object
```

6.2.2 Stereotypes

«stereotype» Argument

- **Metaclass** *Pin*
- **Description** If the pin is owned by a *CallOperationAction*, it has to correspond to a parameter of the operation.
- **Tagged Values**

parameter: Parameter	The parameter the pin provides the values for.

- **Constraints**

  [1] The associated parameter has the same type as the pin.

  ```
  context Argument inv:
    self.baseClass.type = self.parameter.type
  ```

  [2] The associated parameter has the same multiplicity as the pin.

  ```
  context Argument inv:
    self.baseClass.multiplicity =
    self.parameter.multiplicity
  ```

«stereotype» ArgumentExpression

- **Metaclass** *Expression*
- **Description** If an *Expression* represents an argument of an operation call, it is extended by this stereotype.
- **Tagged Values**

parameter: Parameter[0..1]	Parameter of the *Operation* of which the expression is an argument.

- **Constraints**

  [1] The associated parameter is parameter of the operation of which this expression is an argument.

  ```
  context ArgumentExpression inv:
    self.operand.operation.parameter->includes(self.parameter)
  ```

«stereotype» CallExpression

- **Metaclass** *Expression*
- **Description** If an *Expression* represents an operation call, it is extended by this stereotype.
- **Tagged Values**

operation: Operation[1]	The Operation which is called.
property: Property[0..1]	Attribute of the enclosing *Operation*s *Class* designating the object the operation is called upon.
variable: Variable[0..1]	*Variable* of the enclosing *CActivity* designating the object the operation is called upon.

- **Constraints**

  [1] If the corresponding *Operation* is not static, one of the following Tagged Values is set: parameter, property, variable.

```
context CallExpression inv:
  not(self.operation.isStatic) implies
  (parameter->size() + property->size() + variable->size() = 1)
```

[2] If the corresponding *Operation* is static, none of the following Tagged Values is set: parameter, property, variable.

```
context CallExpression inv:
  self.operation.isStatic implies
  (parameter->size() + property->size() + variable->size() = 0)
```

«stereotype» ValueExpression

- **Metaclass** *Expression*
- **Description** If an *Expression* represents an attribute, parameter or variable, it is extended by this stereotype.
- **Tagged Values**

property: Property[0..1]	Attribute of the enclosing *Operation*s *Class* containing the value.
variable: Variable[0..1]	*Variable* of the enclosing *CActivity* containing the value.

- **Constraints**

[1] One of the following Tagged Values is set: parameter, property, variable.

```
context ValueExpression inv:
  self.operation.isStatic implies
  (parameter->size() + property->size() + variable->size() = 1)
```

«stereotype» CNSDActivity

- **Metaclass** *Activity*
- **Description** An Activity extended by this stereotype is graphically represented with the conrete syntax of CNSDs.
- **Constraints**

[1] The directly contained activity nodes have one of the following types: *InitialNode*, *ActivityFinalNode*, *SequenceNode*.

```
context CNSDActivity inv:
  self.baseClass.node->forAll(oclIsTypeOf(InitialNode) or
  oclIsTypeOf(ActivityFinalNode) or
  oclIsTypeOf(SequenceNode))
```

«stereotype» CNSDAssignment

- **Metaclass** *SequenceNode*
- **Description** A SequenceNode extended by this stereotype is the abstract syntax of a CNSD assignment statement.
- **Tagged Values**

designator: Action[1]	The action that writes the value to an attribute or variable.
supplier: Action[1]	The action that provides the input value for the designator action.

- **Constraints**

[1] The designator action has one of the following types: *WriteVariableAction*, *WriteStructuralFeatureAction*.

```
context CNSDAssignment inv:
  self.designator.oclIsTypeOf(WriteVariableAction) or
  self.designator.oclIsTypeOf(WriteStructuralFeatureAction)
```

[2] The supplier action provides input for the designator action.

```
context CNSDAssignment inv:
  self.supplier.result.outgoing.target =
  self.designator.value
```

[3] Supplier and designator are directly contained by the base class.

```
context CNSDAssignment inv:
  self.baseClass.node.include(self.supplier) and
  self.baseClass.node.include(self.designator)
```

[4] designator is directly contained by the extended sequence node.

```
context CNSDAssignment inv:
  self.baseClass.node->includes(designator)
```

[5] supplier is directly contained by the extended sequence node.

```
context CNSDAssignment inv:
  self.baseClass.node->includes(supplier)
```

«stereotype» CNSDOperationCall

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD operation call statement.
- **Tagged Values**

opAction: CallOperationAction[1]	The action that executes the operation.

- **Constraints**

  [1] opAction is directly contained by the extended sequence node.

```
context CNSDOperationCall inv:
  self.baseClass.node->includes(opAction)
```

«stereotype» CNSDIfElse

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD IfElse statement.
- **Tagged Values**

decisionNode: DecisionNode[1]	The DecisionNode evaluating the if condition.
elseNode: SequenceNode[1]	Contains the abstract syntax of the CNSD else-trace.
ifCondition: ValueSpecification[1]	The guard expression of decisionNode.
ifNode: SequenceNode[1]	TContains the abstract syntax of the CNSD if-trace.

- **Constraints**

  [1] The decisionNode has 2 outgoing edges.

```
context CNSDIfElse inv:
  self.decisionNode.outgoing->size() = 2
```

  [2] elseNode and ifNode are successors of decisionNode.

```
context CNSDIfElse inv:
  self.decisionNode.outgoing->collect(source)->
  includes(ifNode->union(elseNode))
```

  [3] The ifCondition is guard of the edge from decisionNode to ifNode.

```
context CNSDIfElse inv:
  self.decisionNode.outgoing->exists(guard = ifCondition and
  source = ifNode)
```

  [4] decisionNode is directly contained by the extended sequence node.

```
context CNSDIfElse inv:
  self.baseClass.node->includes(decisionNode)
```

  [5] elseNode is directly contained by the extended sequence node.

```
context CNSDIfElse inv:
  self.baseClass.node->includes(elseNode)
```

  [6] ifNode is directly contained by the extended sequence node.

```
context CNSDIfElse inv:
  self.baseClass.node->includes(ifNode)
```

«stereotype» CNSDWhile

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD While statement.
- **Tagged Values**

decisionNode: DecisionNode[1]	The DecisionNode evaluating the loop condition.
body: SequenceNode[1]	Contains the abstract syntax of the CNSD loop body.
loopCondition: ValueSpecification[1]	The guard expression of decisionNode.

- **Constraints**

[1] The body node is predecessor of the decisionNode.

```
context CNSDDoWhile inv:
  self.body.outgoing.target = decisionNode
```

[2] The body node is successor of the decisionNode.

```
context CNSDDoWhile inv:
  self.body.incoming.source = decisionNode
```

[3] The loopCondition is guard of the edge from decisionNode to body.

```
context CNSDDoWhile inv:
  self.body.incoming.guard = loopCondition
```

[4] The decision node is successor of the *InitialNode*.

```
context CNSDDoWhile inv:
  self.decisionNode.incoming.source.oclIsTypeOf(InitialNode)
```

[5] decisionNode is directly contained by the extended sequence node.

```
context CNSDDoWhile inv:
  self.baseClass.node->includes(decisionNode)
```

[6] body is directly contained by the extended sequence node.

```
context CNSDDoWhile inv:
  self.baseClass.node->includes(body)
```

«stereotype» CNSDWhile

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD While statement.

- **Tagged Values**

decisionNode: DecisionNode[1]	The DecisionNode evaluating the loop condition.
body: SequenceNode[1]	Contains the abstract syntax of the CNSD loop body.
loopCondition: ValueSpecification[1]	The guard expression of decisionNode.

- **Constraints**

[1] The body node is predecessor of the decisionNode.

```
context CNSDDoWhile inv:
  self.body.outgoing.target = decisionNode
```

[2] The body node is successor of the decisionNode.

```
context CNSDDoWhile inv:
  self.body.incoming.source = decisionNode
```

[3] The loopCondition is guard of the edge from decisionNode to body.

```
context CNSDDoWhile inv:
  self.body.incoming.guard = loopCondition
```

[4] The decision node is successor of the *InitialNode*.

```
context CNSDDoWhile inv:
  self.decisionNode.incoming.source.oclIsTypeOf(InitialNode)
```

[5] decisionNode is directly contained by the extended sequence node.

```
context CNSDDoWhile inv:
  self.baseClass.node->includes(decisionNode)
```

[6] body is directly contained by the extended sequence node.

```
context CNSDDoWhile inv:
  self.baseClass.node->includes(body)
```

«stereotype» CNSDForEach

- **Metaclass** *CompleteStructuredActivities::SequenceNode*
- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD ForEach statement.
- **Tagged Values**

collectionProvider: Action[1]	Action that provides the elements of the collection.
buffer: CentralBufferNode[1]	Buffer that stores the elements of the collection.
variableAction: AddVariableValueAction[1]	Action that writes the value of the variable before each iteration.
body: SequenceNode[1]	SequenceNode encapsulating the body of the foreach statement.

- **Constraints**

  [1] CollectionProvider is of one of the following types: ReadVariableAction, ReadStructuralFeatureAction.

  ```
  context CNSDForEach inv:

    self.collectionProvider.oclIsTypeOf(ReadVariableAction)
    or self.collectionProvider.
    oclIsTypeOf(ReadStructuralFeatureAction)
  ```

  [2] Edge weight from collectionProvider's result pin to buffer is '*'.

  ```
  context CNSDForEach inv:

    self.collectionProvider.result.outgoing.weight = *
  ```

«stereotype» CNSDCase

- **Metaclass** *CompleteStructuredActivities::SequenceNode*

- **Description** A *SequenceNode* extended by this stereotype encapsulates the abstract syntax of a CNSD case statement.

- **Tagged Values**

decisionNode: DecisionNode[1]	The decision node implementing the case distinction.
traces: SequenceNode[*]	The sequence nodes that encapsulate the abstract syntaxes of the different traces of the case statement.

- **Constraints**

  [1] The decision node is predecessor to all traces.

  ```
  context CNSDCase inv:

    self.traces->collect(incoming)->forAll(source = decisionNode)
  ```

  [2] decisionNode is directly contained by the extended sequence node.

  ```
  context CNSDCase inv:

    self.baseClass.node->includes(decisionNode)
  ```

  [3] All traces are directly contained by the extended sequence node.

  ```
  context CNSDCase inv:

    self.baseClass.node->includes(traces)
  ```

6.3 Model Transformation

6.3.1 Intermediate Structure

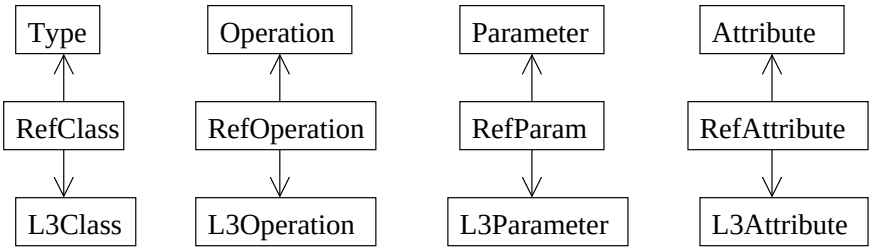

Figure 6.1: Intermediate Structure: Class Structure References

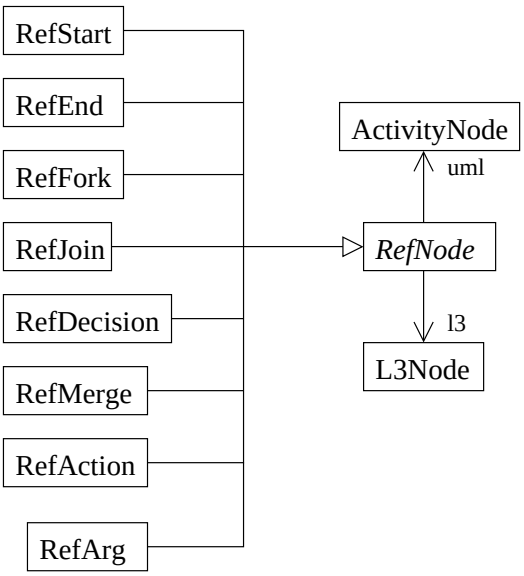

Figure 6.2: Intermediate Structure: Node References

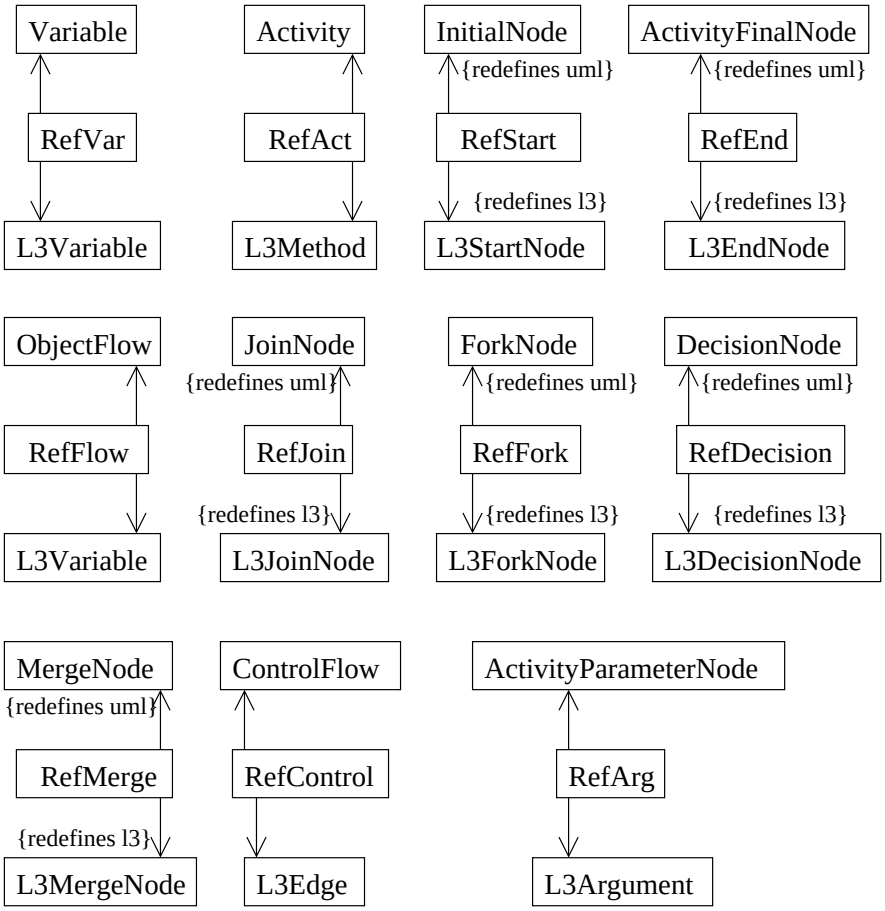

Figure 6.3: Intermediate Structure: Control Structures

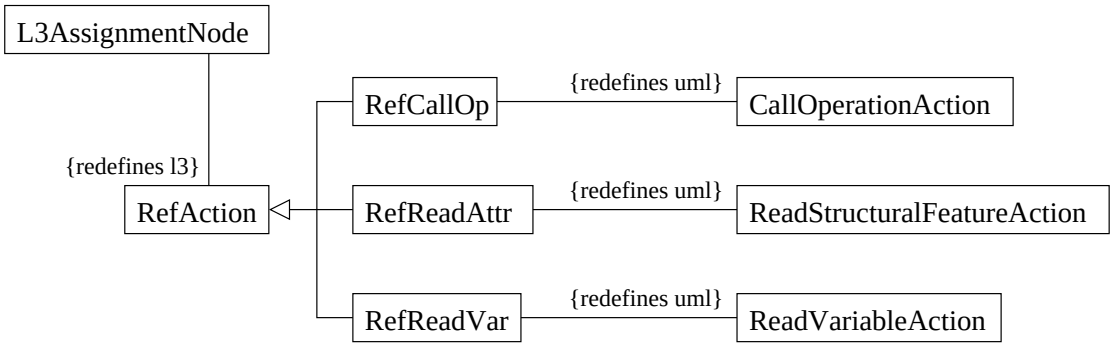

Figure 6.4: Intermediate Structure: Actions with Output

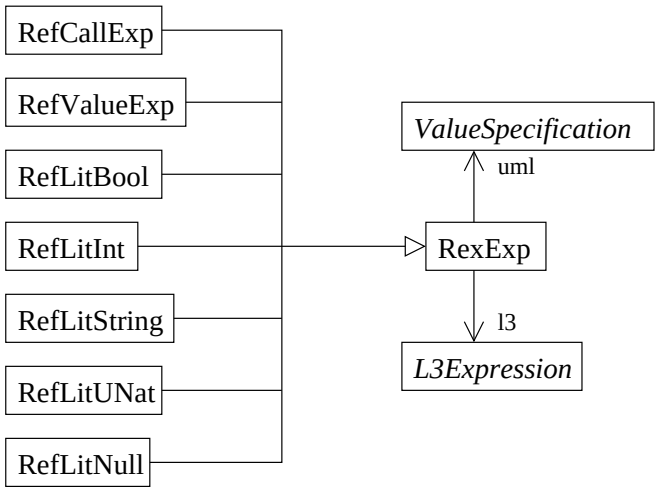

Figure 6.5: Intermediate Structure: Expressions

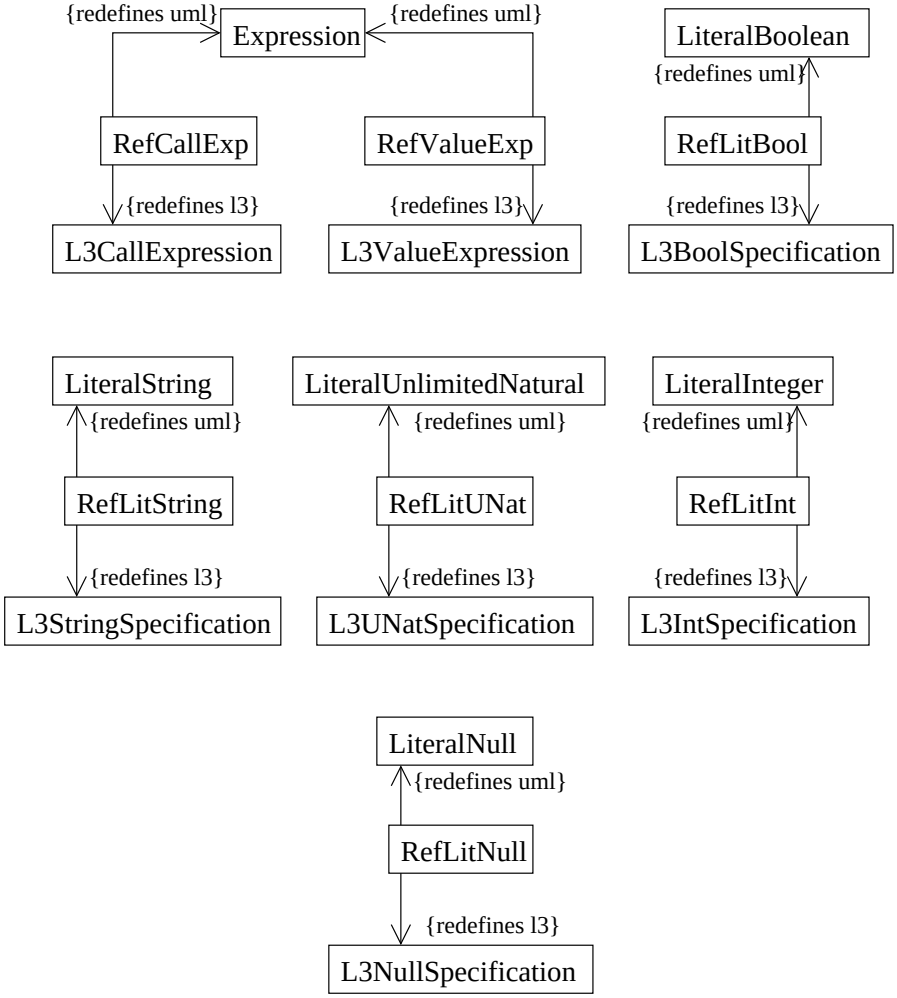

Figure 6.6: Intermediate Structure: Expression References

6.3.2 Transformation Rules

Preparation of CUML Model

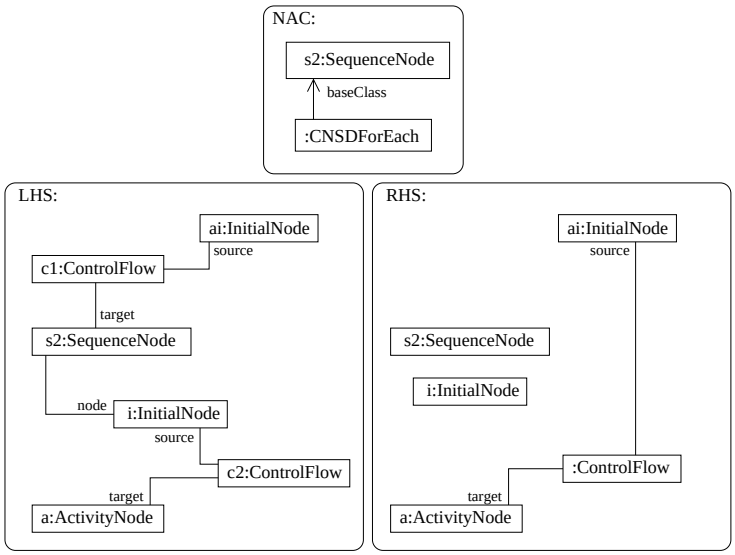

Figure 6.7: Rule: connectInitialSequence

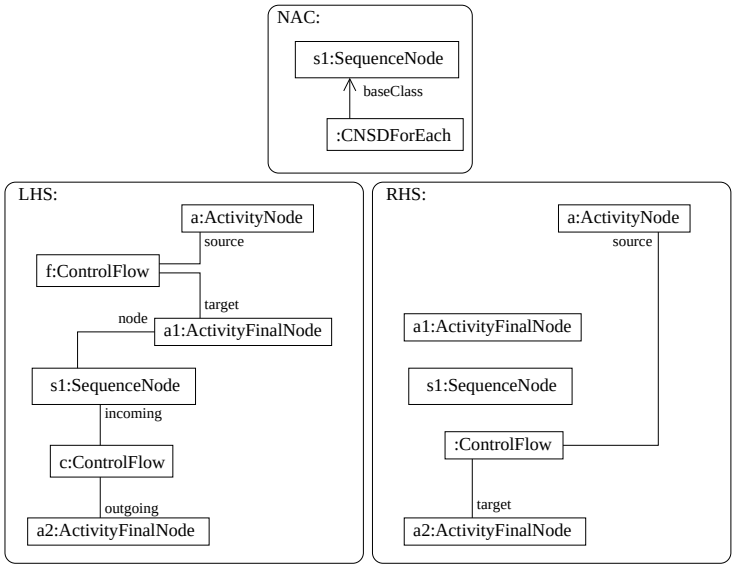

Figure 6.8: Rule: connectSequenceFinal

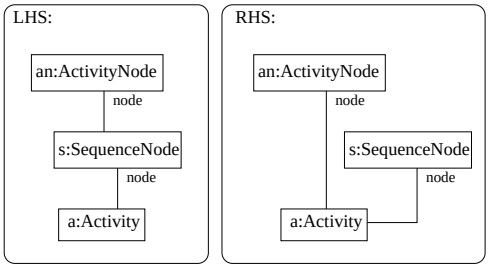

Figure 6.9: Rule: addNodeActivity

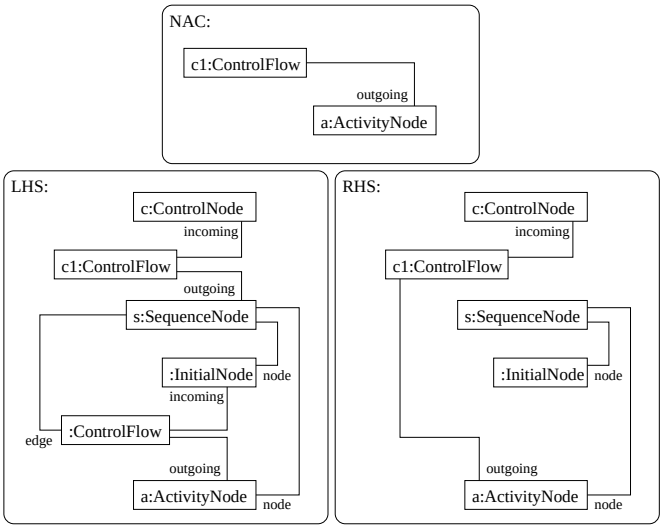

Figure 6.10: Rule: controlSequenceNode

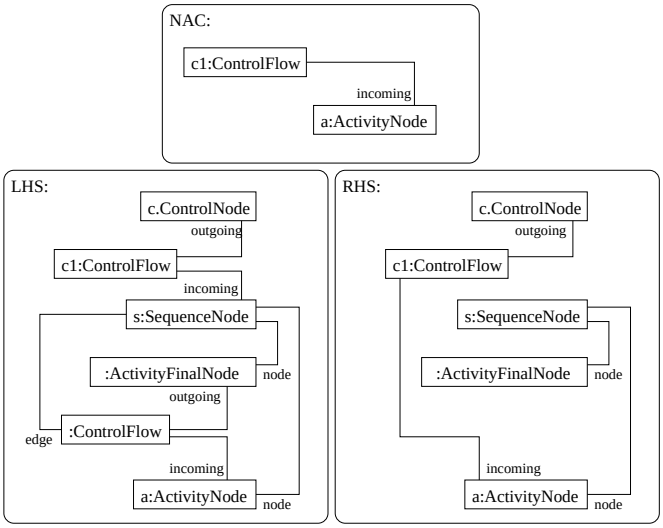

Figure 6.11: Rule: sequenceControlNode

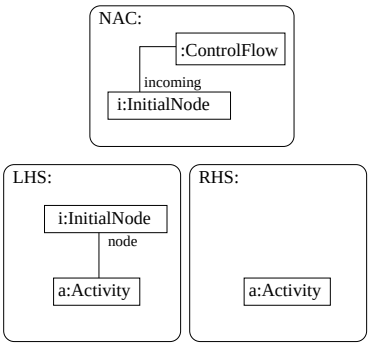

Figure 6.12: Rule: deleteInitial

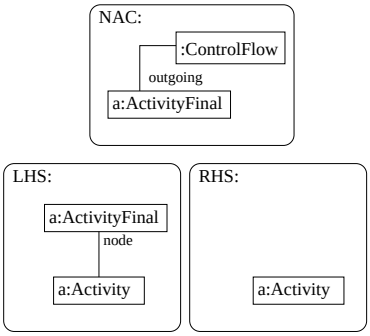

Figure 6.13: Rule: deleteFinal

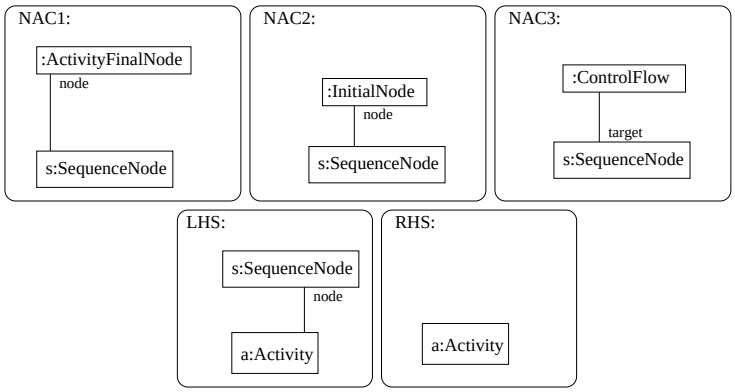

Figure 6.14: Rule: deleteSequence

Create L3Model

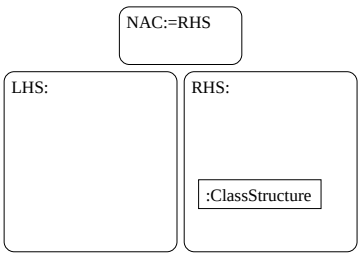

Figure 6.15: Rule: createClassStructure

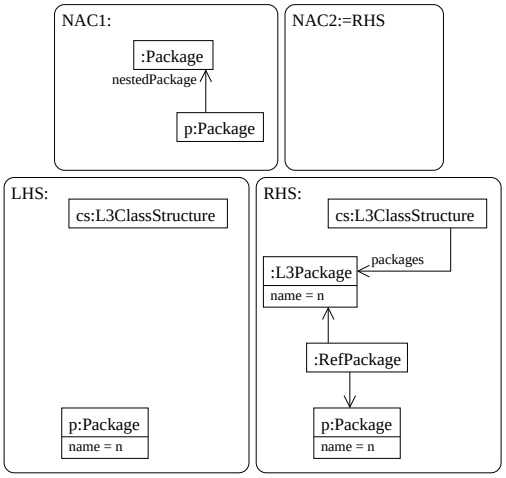

Figure 6.16: Rule: createPackage

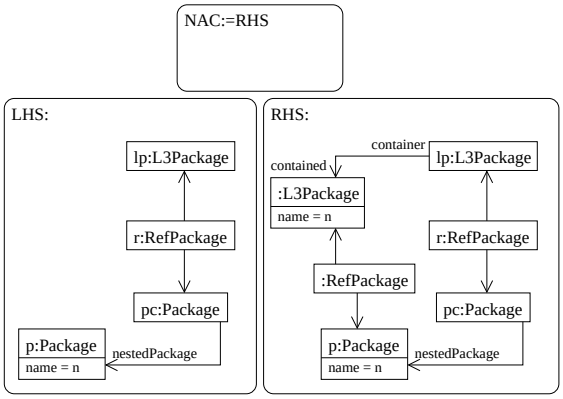

Figure 6.17: Rule: createNestedPackage

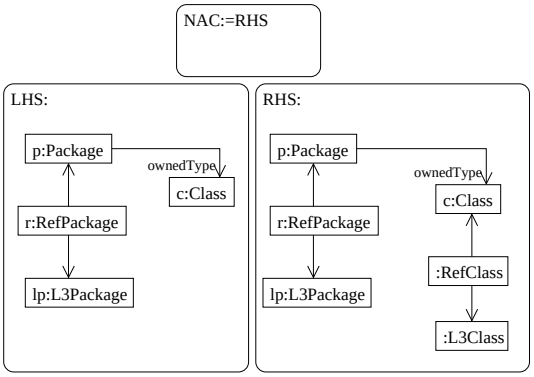

Figure 6.18: Rule: createClass

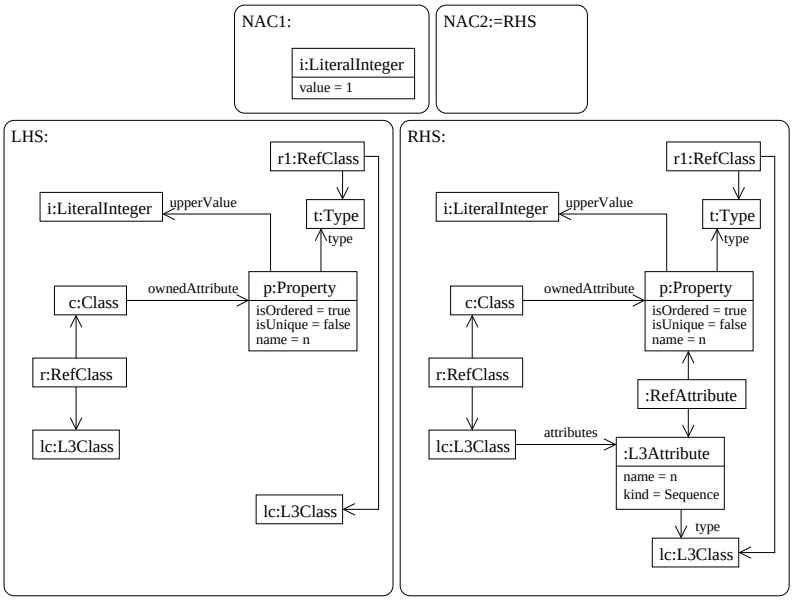

Figure 6.19: Rule: createAttributeSequence

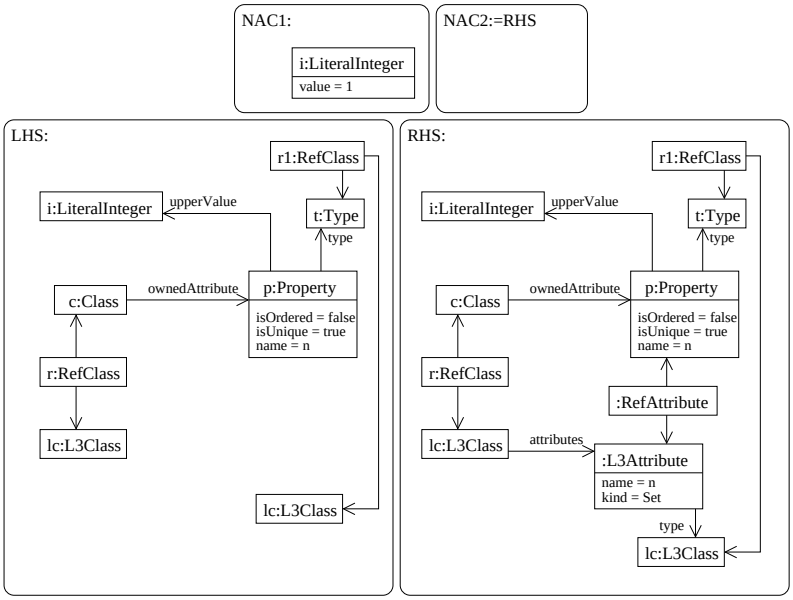

Figure 6.20: Rule: createAttributeSet

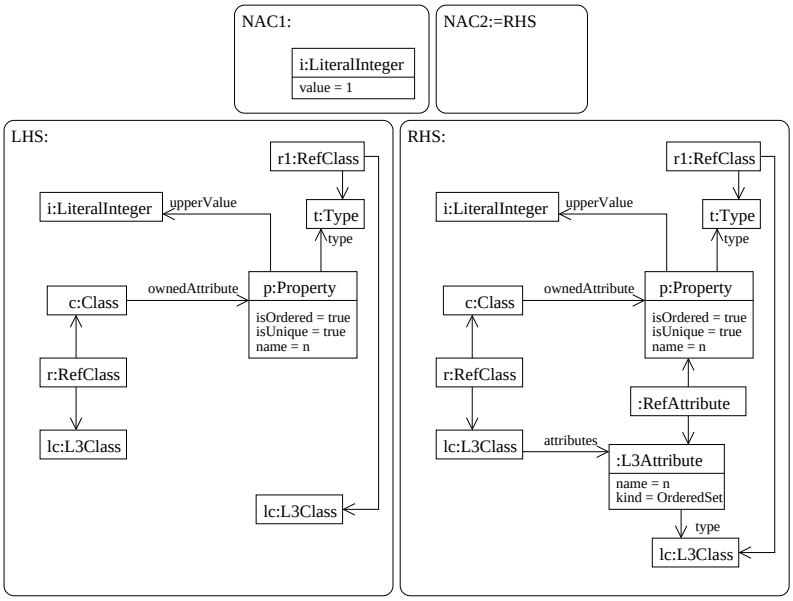

Figure 6.21: Rule: createAttributeOrderedSet

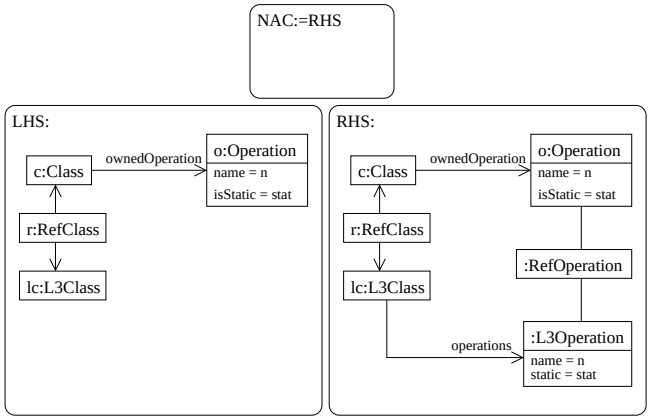

Figure 6.22: Rule: createOperation

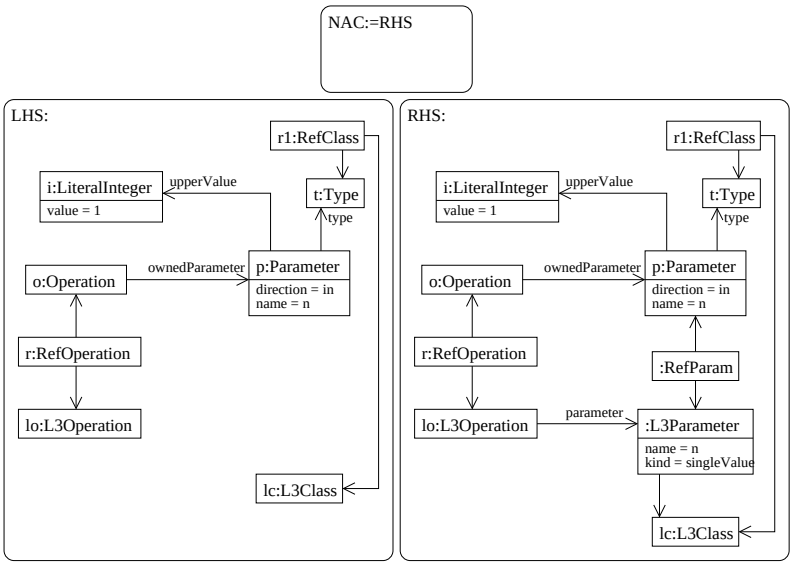

Figure 6.23: Rule: createParameter

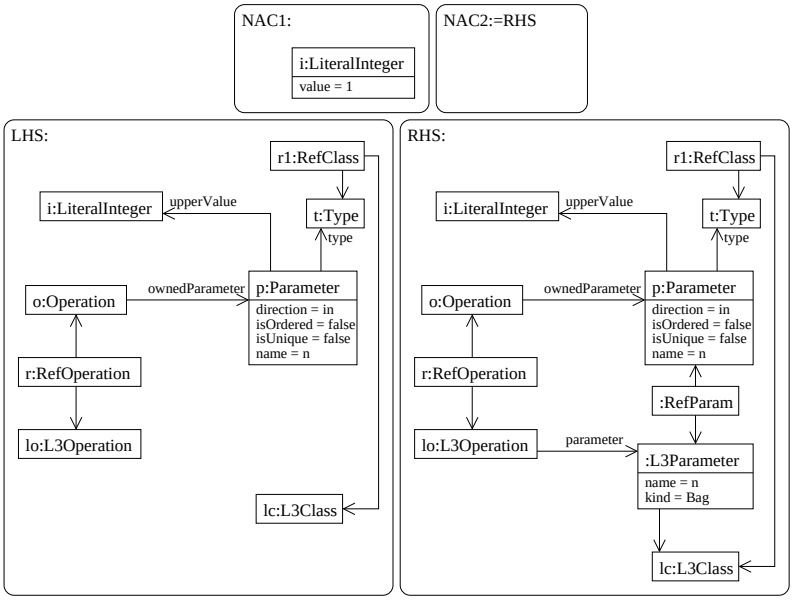

Figure 6.24: Rule: createParameterBag

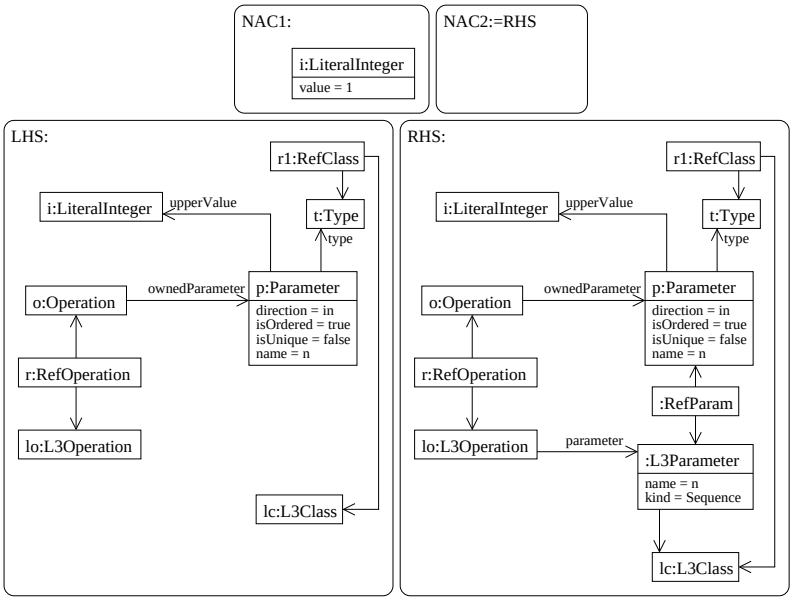

Figure 6.25: Rule: createParameterSequence

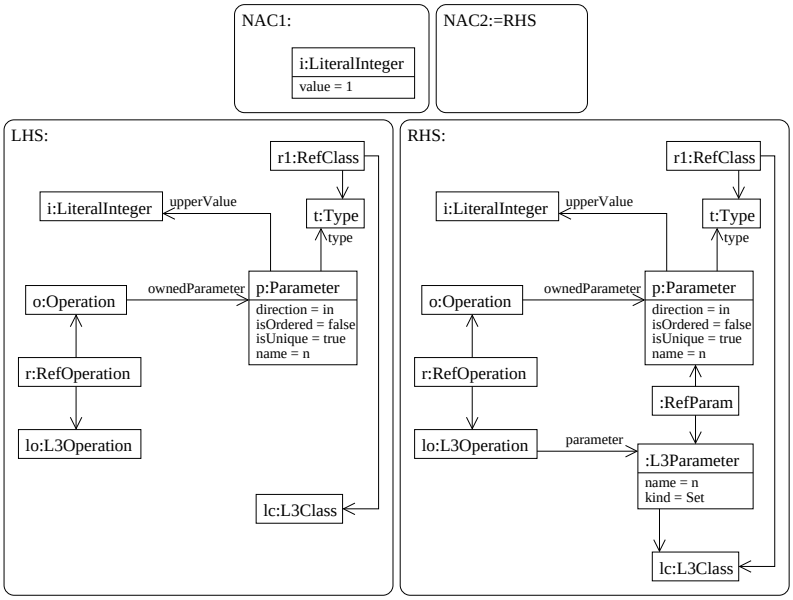

Figure 6.26: Rule: createParameterSet

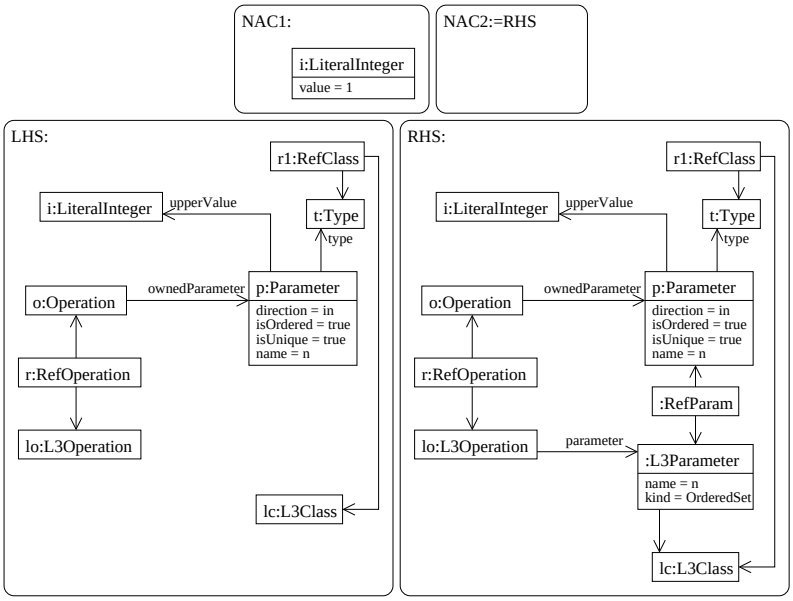

Figure 6.27: Rule: createParameterOrderedSet

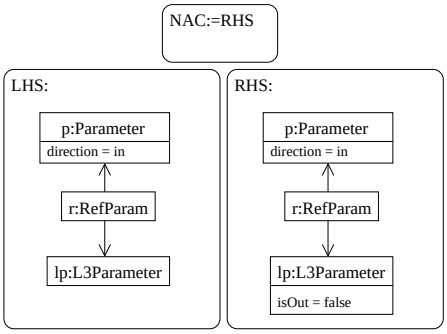

Figure 6.28: Rule: setParamIn

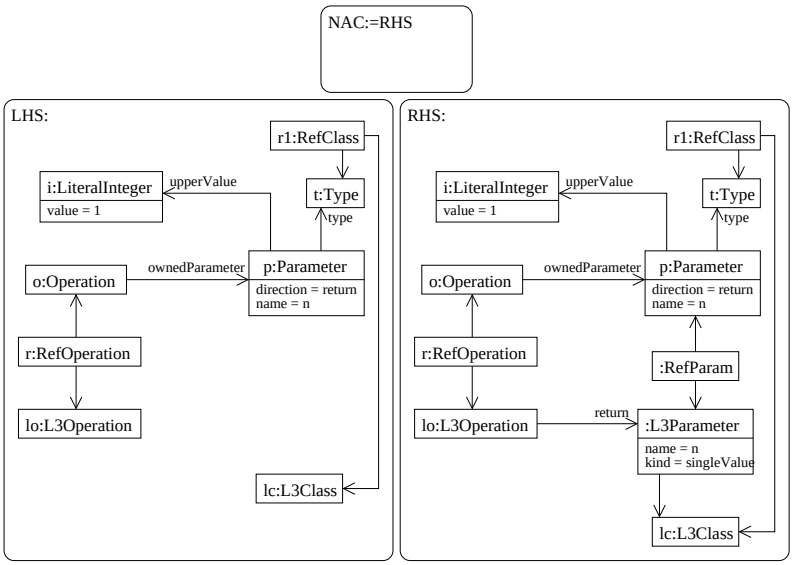

Figure 6.29: Rule: createReturn

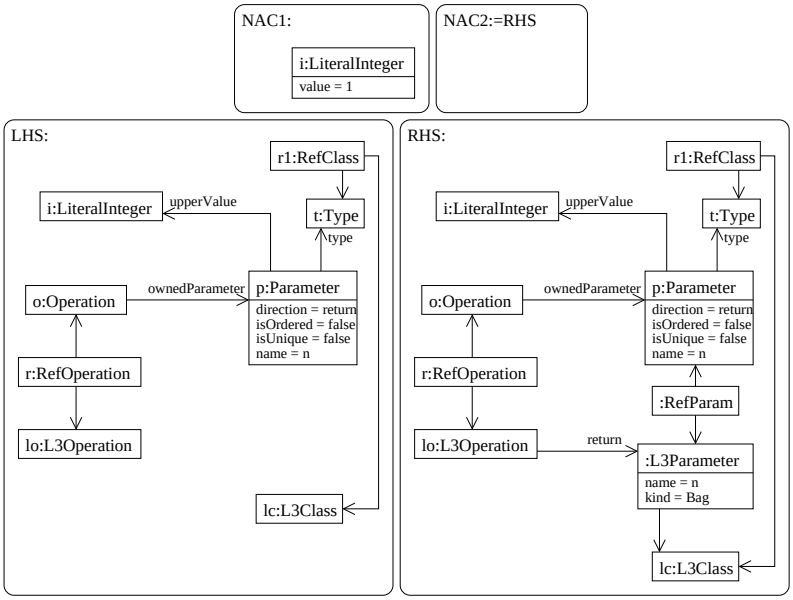

Figure 6.30: Rule: createReturnBag

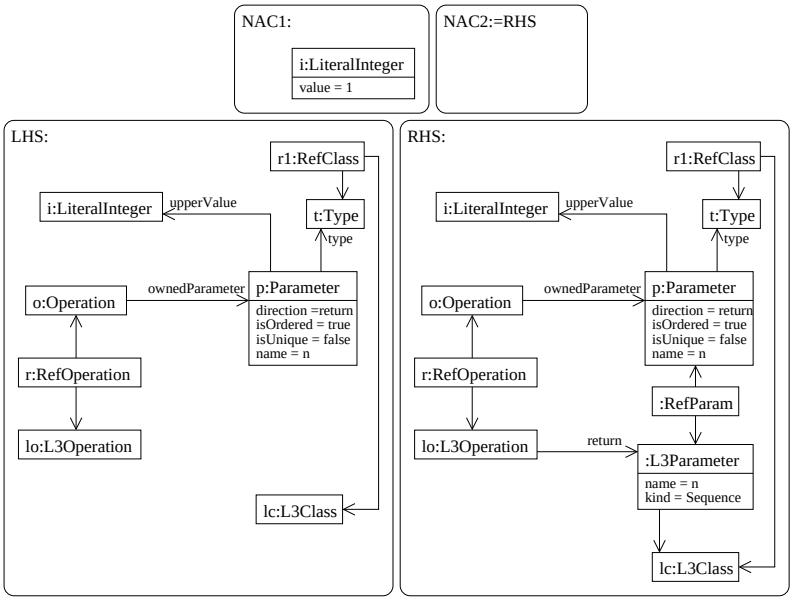

Figure 6.31: Rule: createReturnSequence

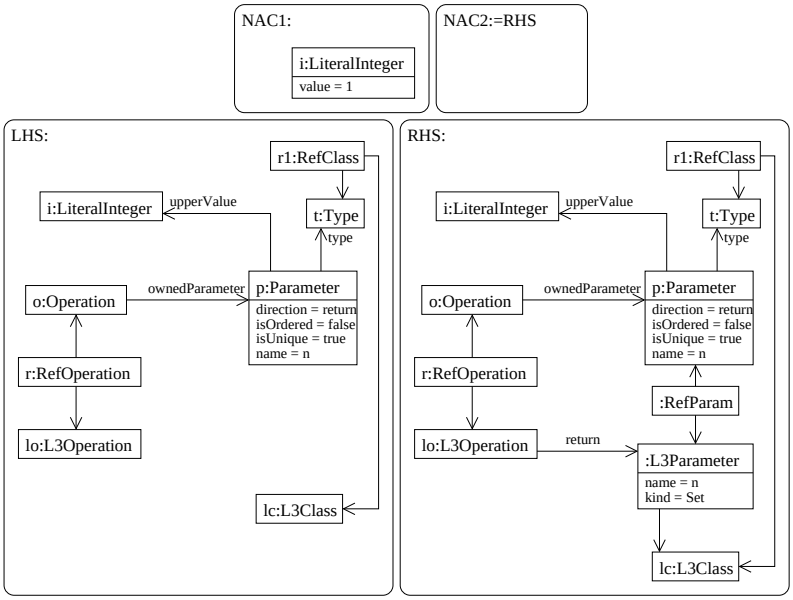

Figure 6.32: Rule: createReturnSet

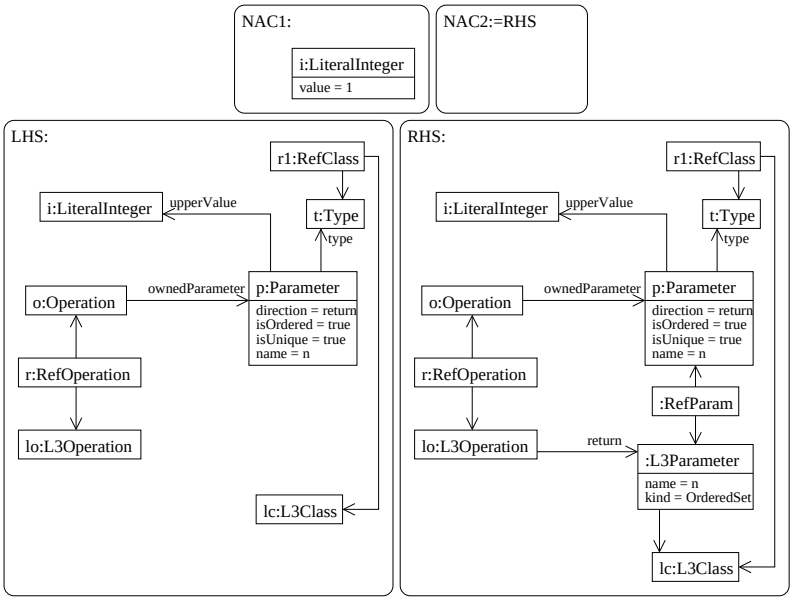

Figure 6.33: Rule: createReturnOrderedSet

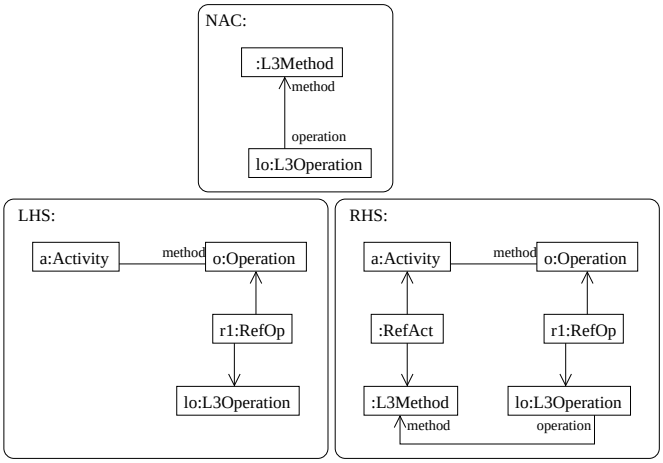

Figure 6.34: Rule: createMethod

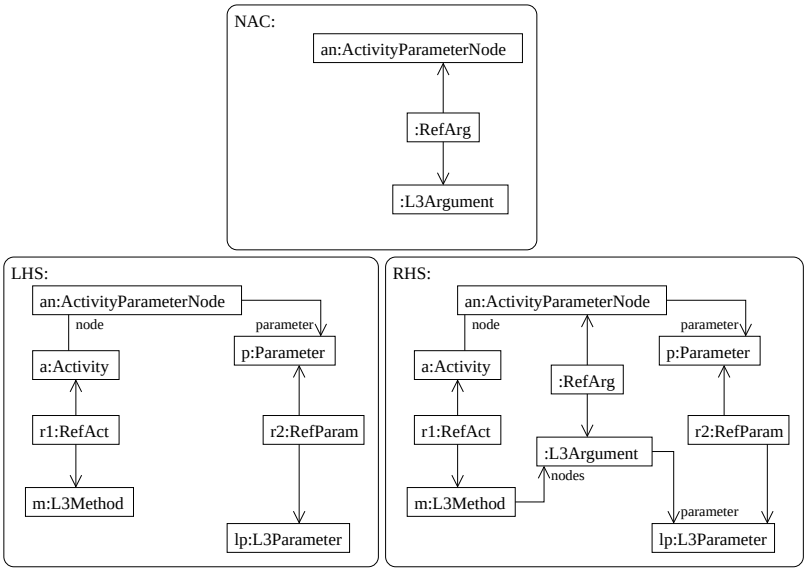

Figure 6.35: Rule: createArgument

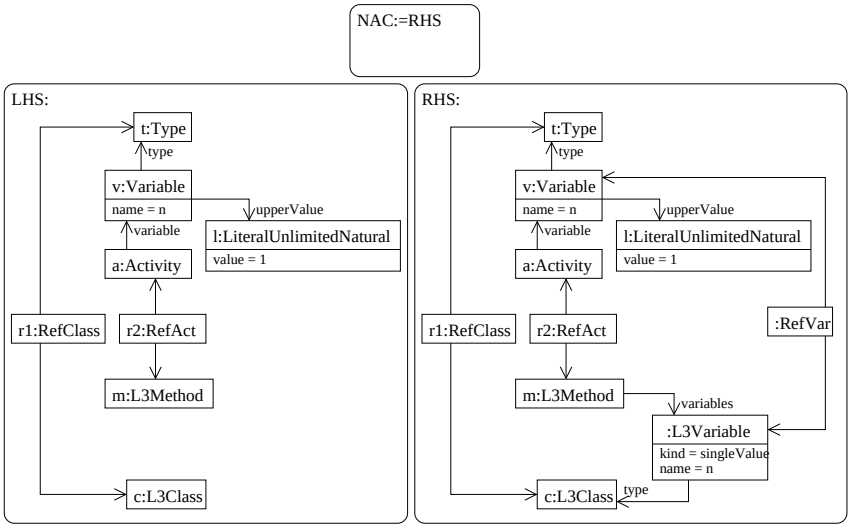

Figure 6.36: Rule: createVarSinglevalue

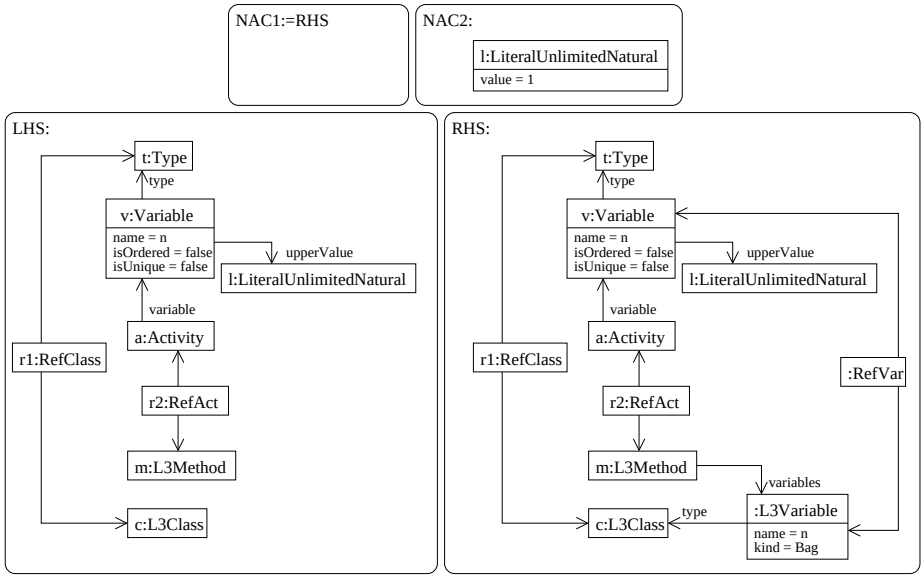

Figure 6.37: Rule: createVarBag

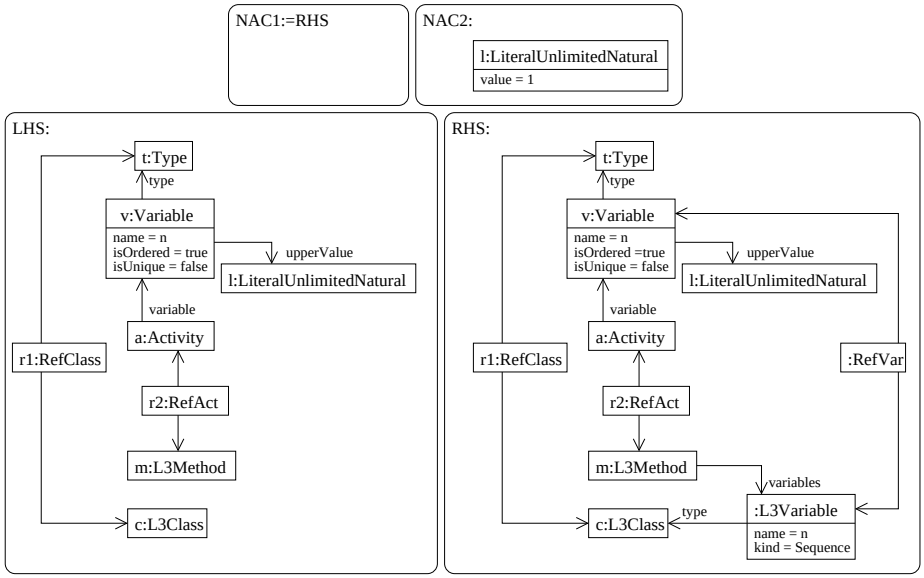

Figure 6.38: Rule: createVarSequence

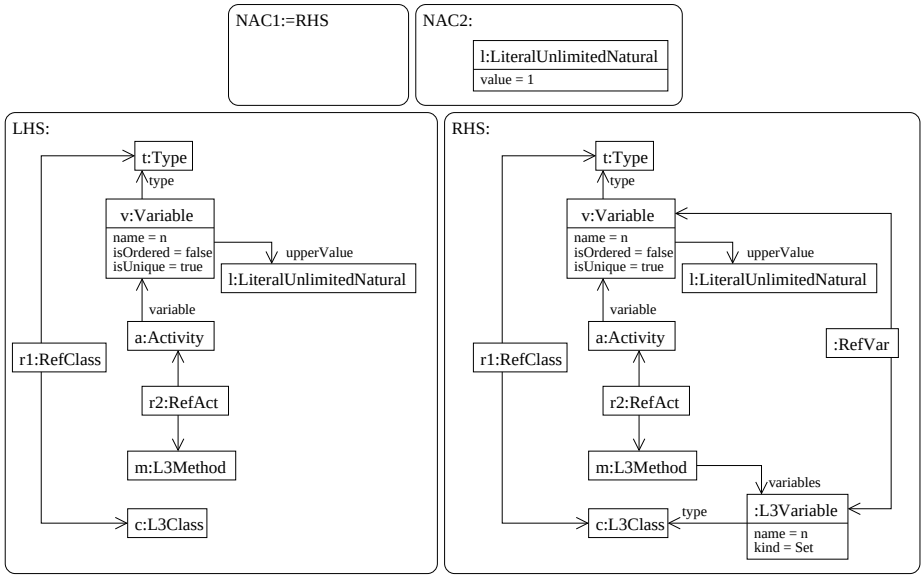

Figure 6.39: Rule: createVarSet

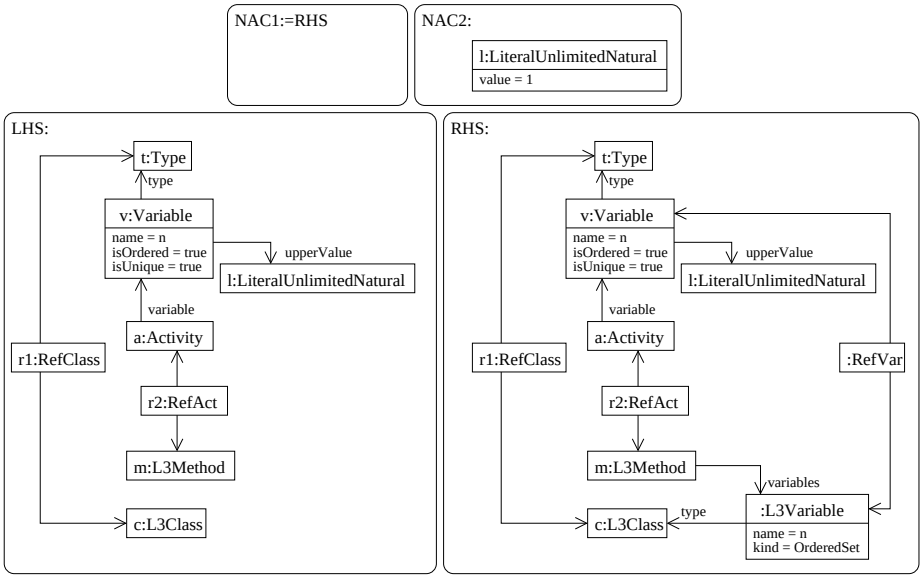

Figure 6.40: Rule: createVarOrderedSet

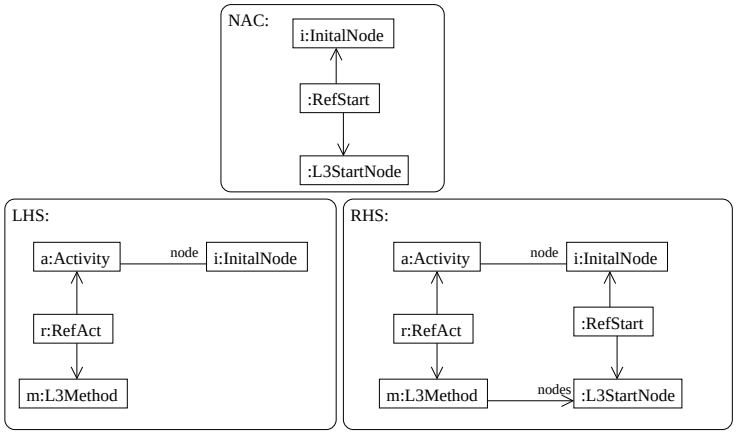

Figure 6.41: Rule: createStart

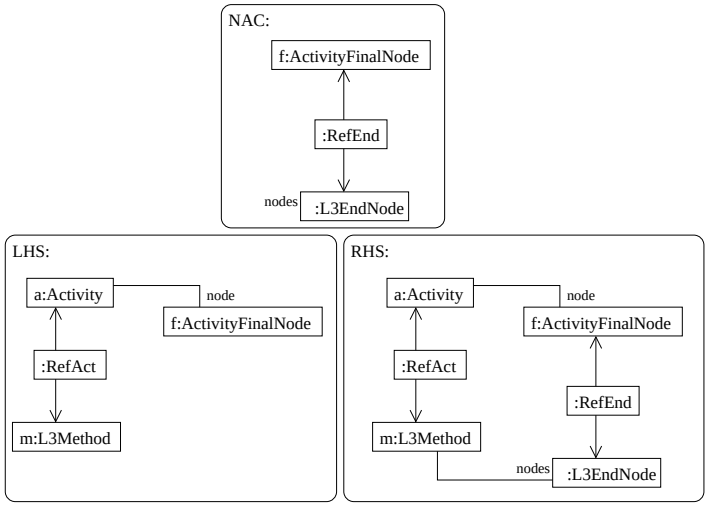

Figure 6.42: Rule: createEnd

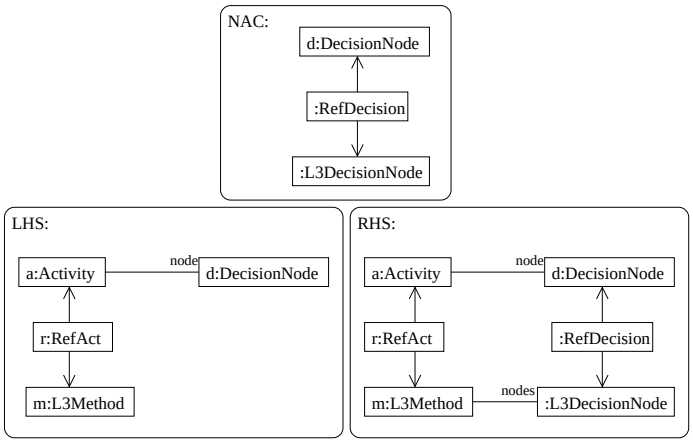

Figure 6.43: Rule: createDecision

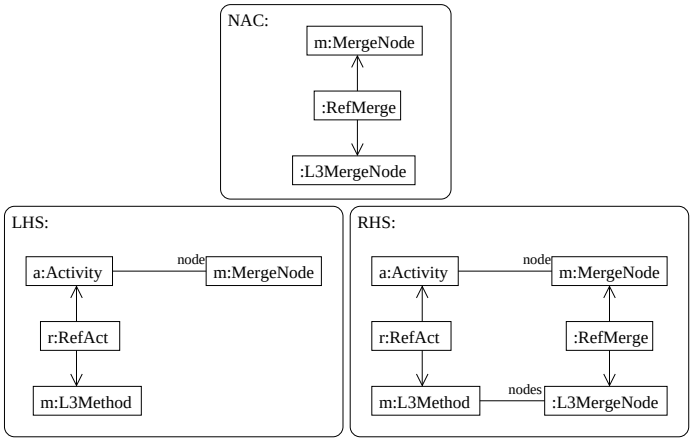

Figure 6.44: Rule: createMerge

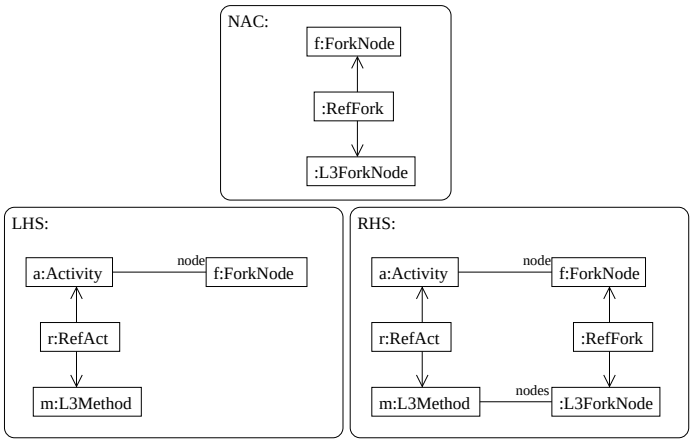

Figure 6.45: Rule: createFork

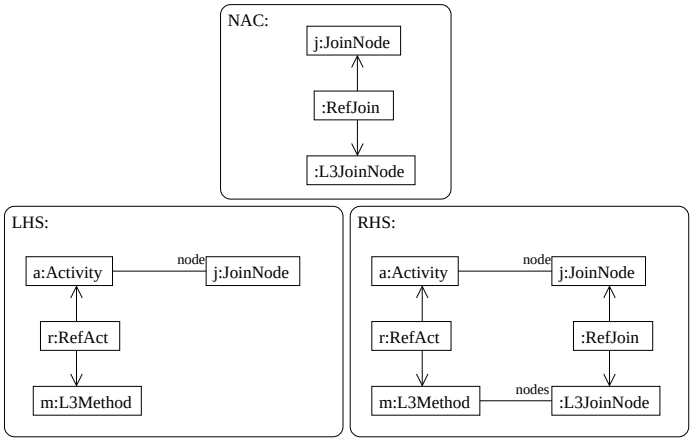

Figure 6.46: Rule: createJoin

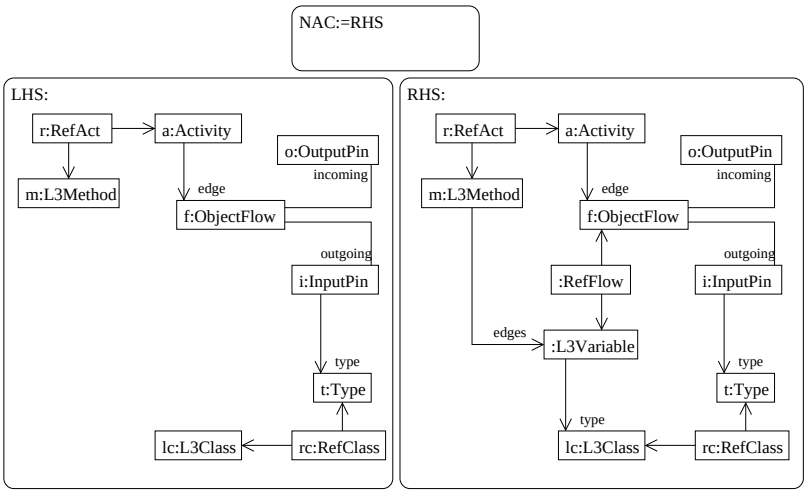

Figure 6.47: Rule: createVariable

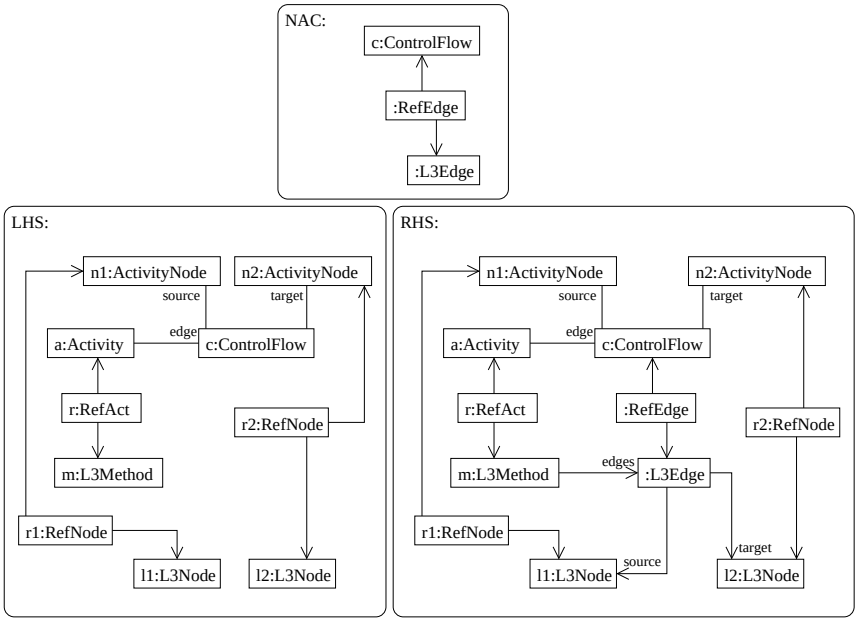

Figure 6.48: Rule: createEdge

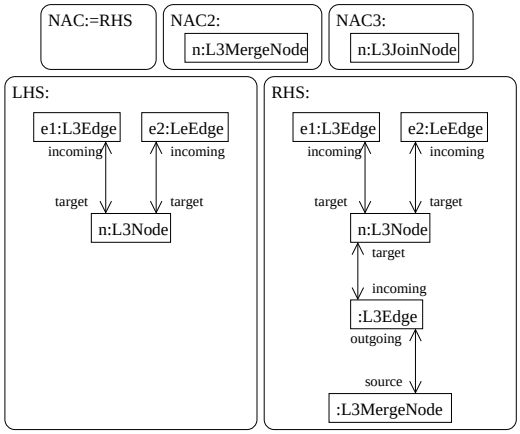

Figure 6.49: Rule: createImplicitMerge

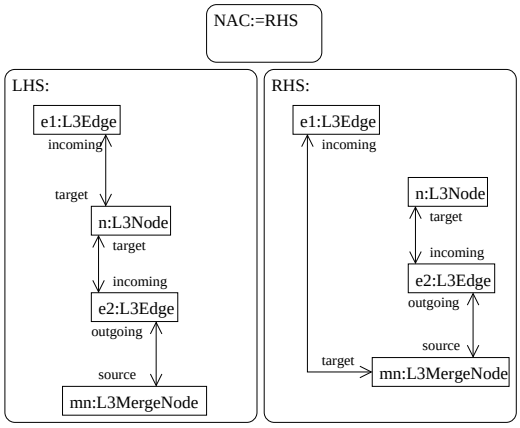

Figure 6.50: Rule: mergeEdges

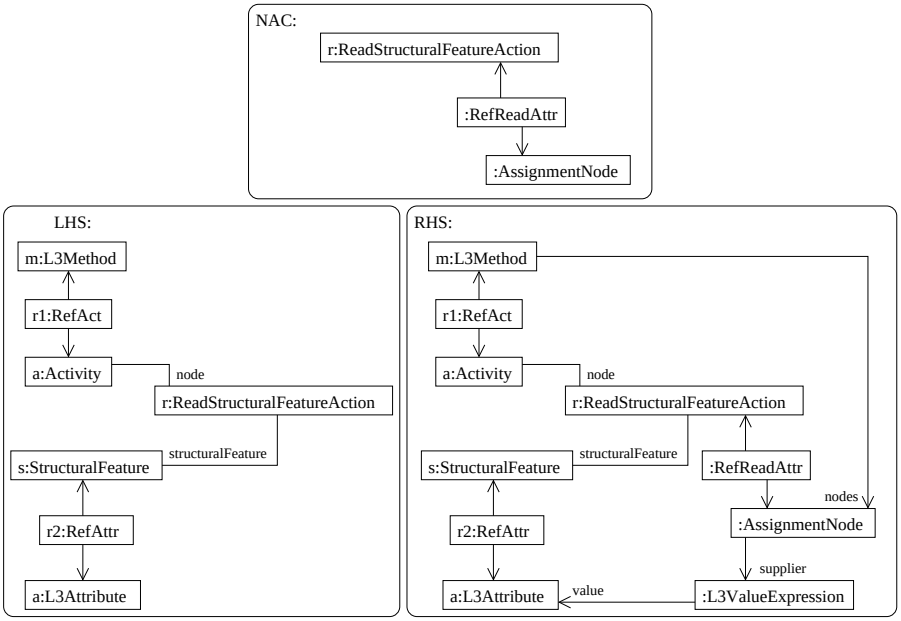

Figure 6.51: Rule: createReadFeature

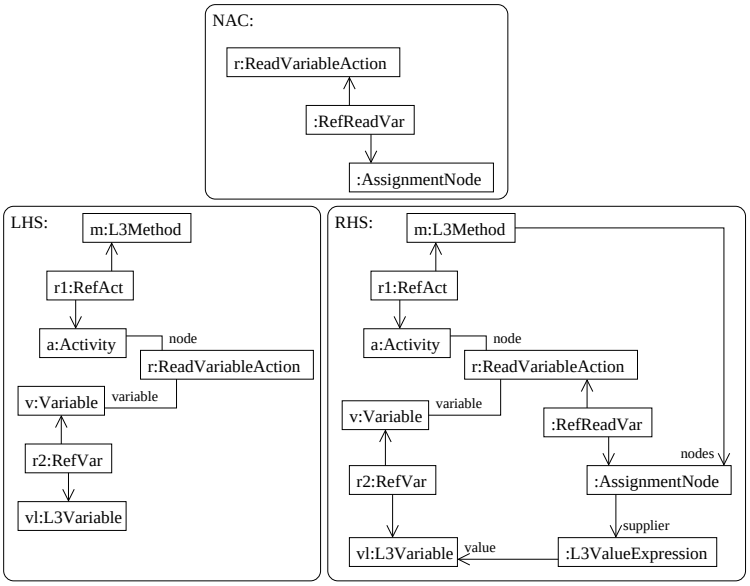

Figure 6.52: Rule: createReadVariable

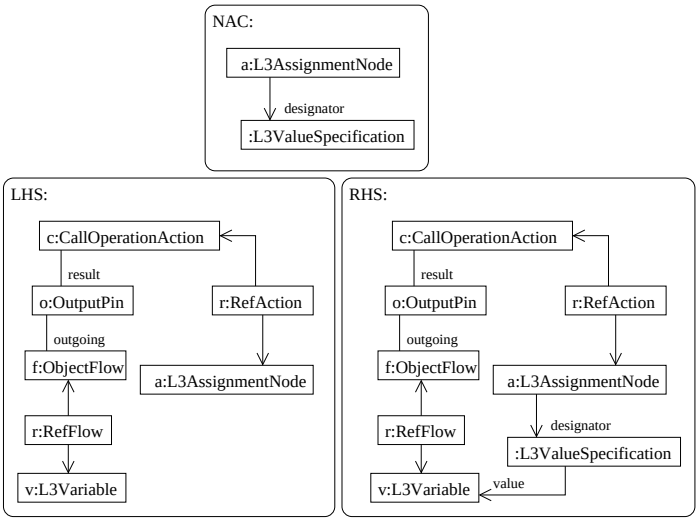

Figure 6.53: Rule: createDesignator

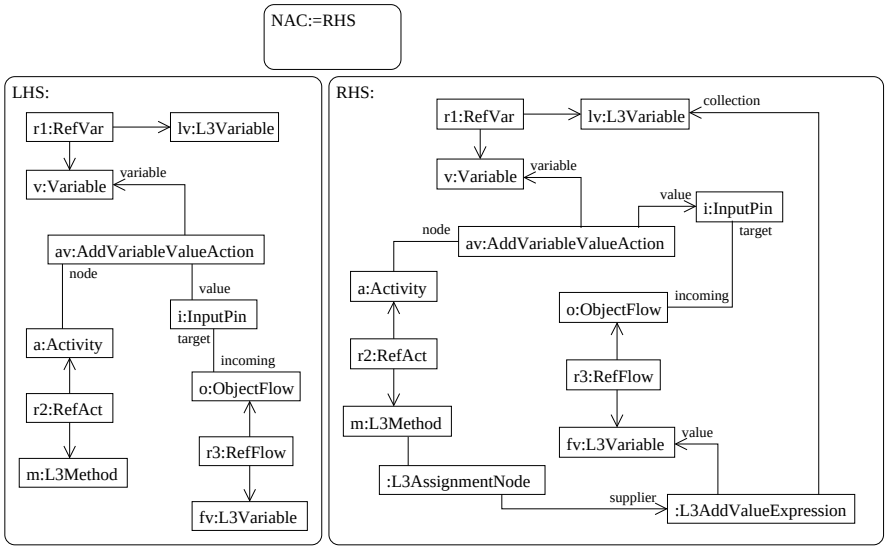

Figure 6.54: Rule: createAddValueVariable

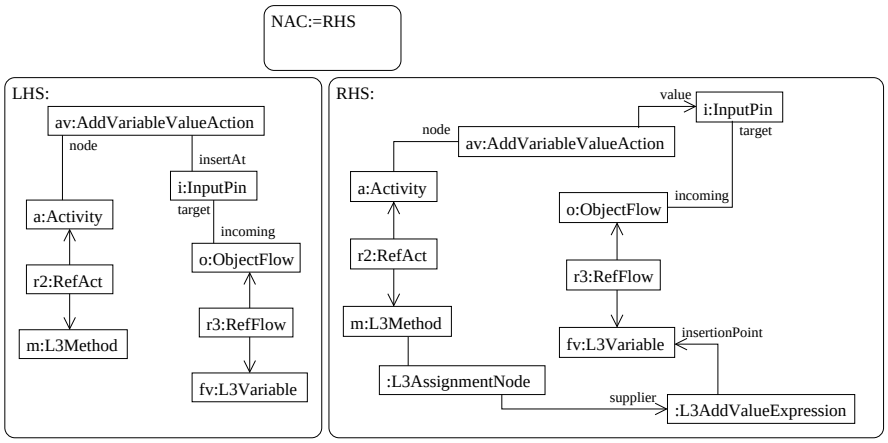

Figure 6.55: Rule: createInsertionVar

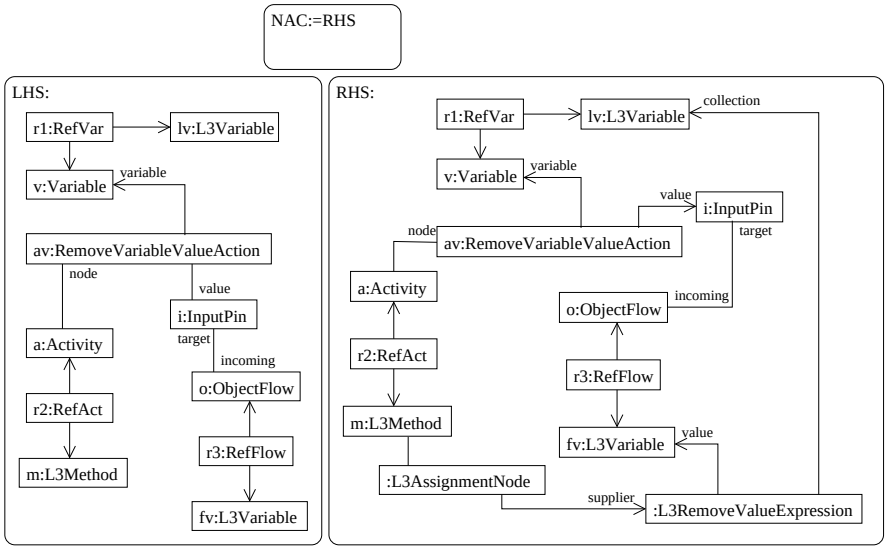

Figure 6.56: Rule: createRemoveValueVariable

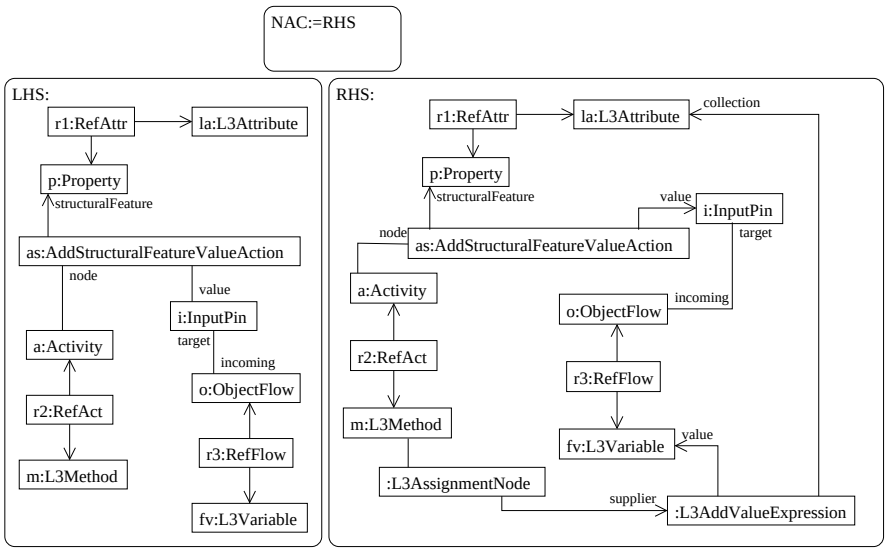

Figure 6.57: Rule: createAddValueSFeature

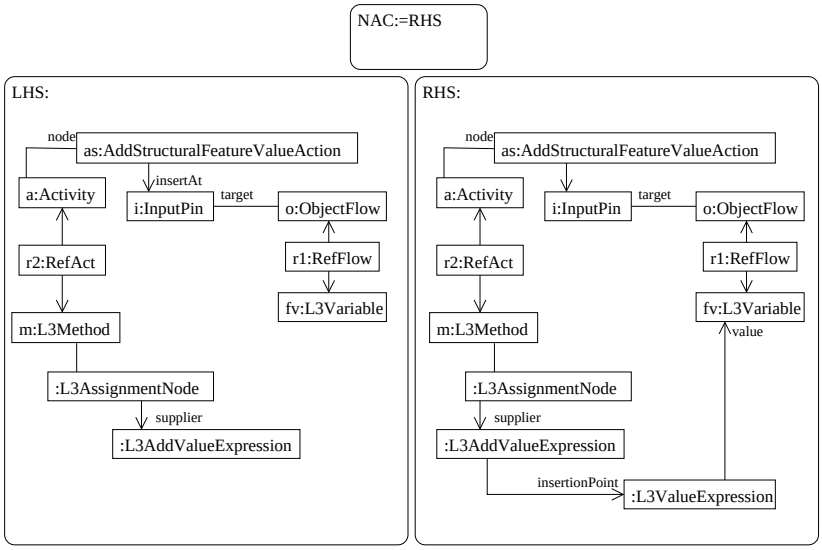

Figure 6.58: Rule: createInsertionSF

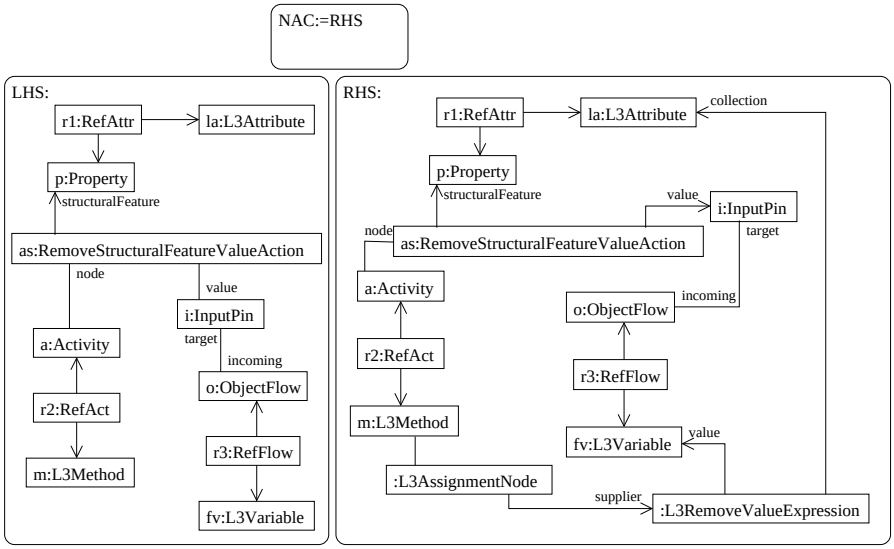

Figure 6.59: Rule: createRemoveValueSFeature

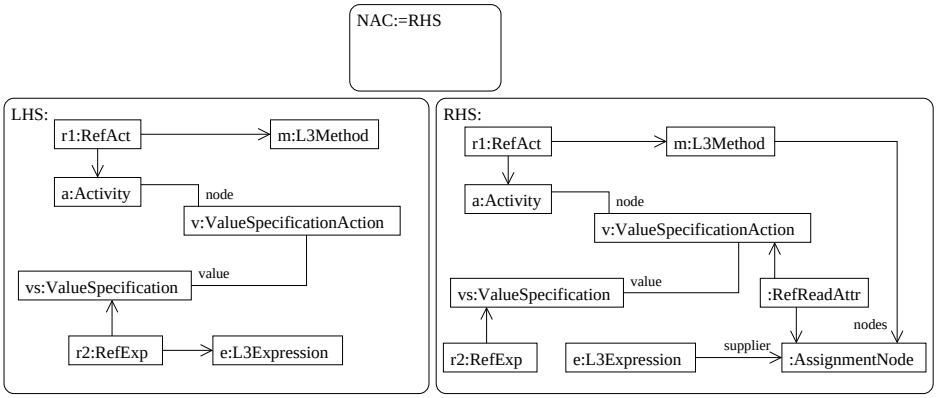

Figure 6.60: Rule: createValueSpecification

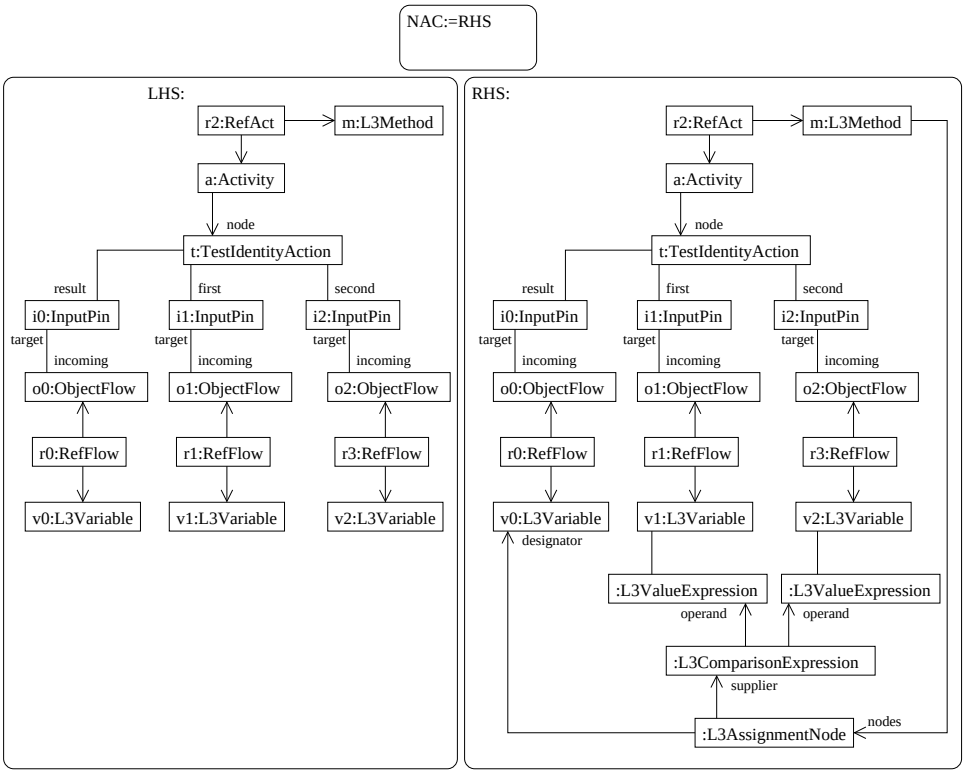

Figure 6.61: Rule: createTestIdentity

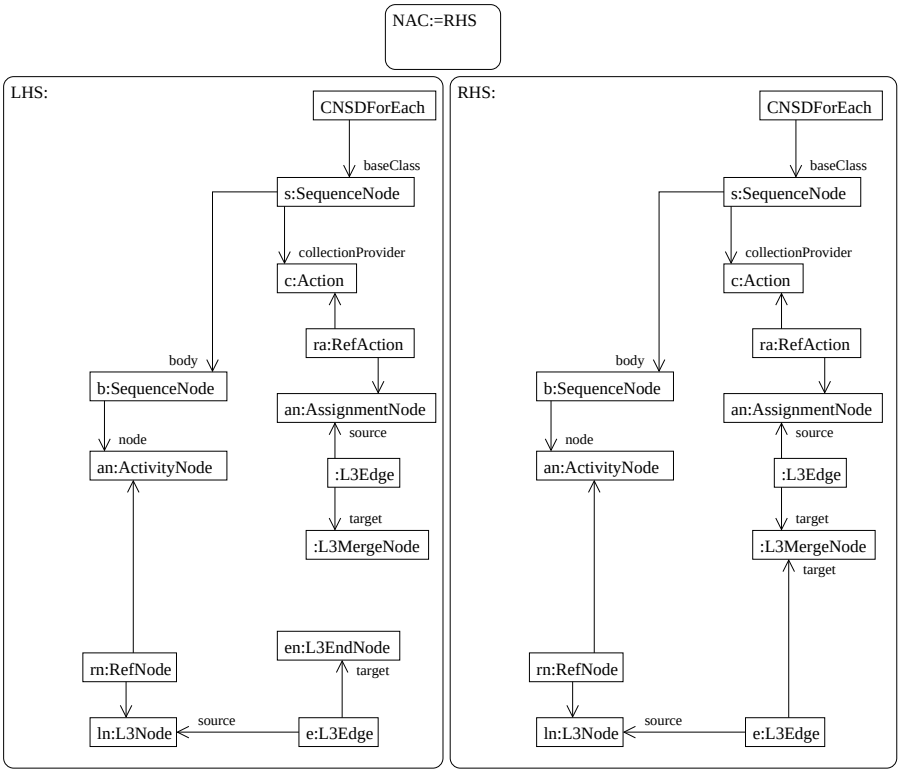

Figure 6.62: Rule: closeIteratorLoop

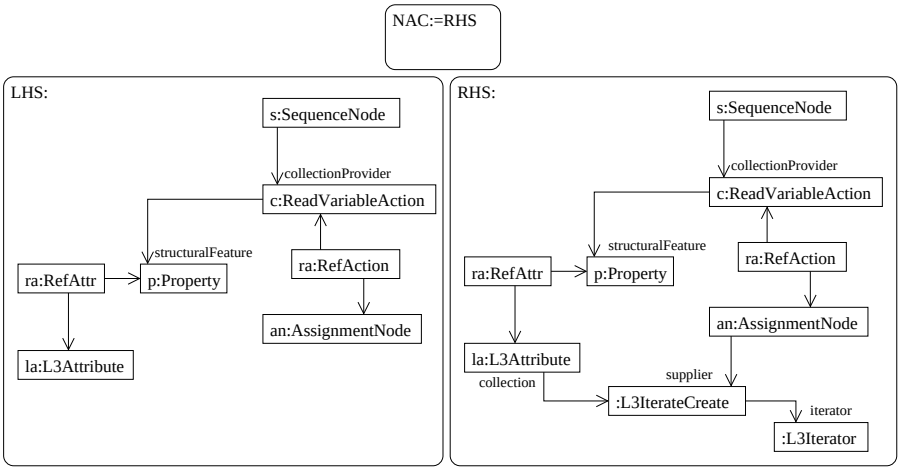

Figure 6.63: Rule: createAttributeIterator

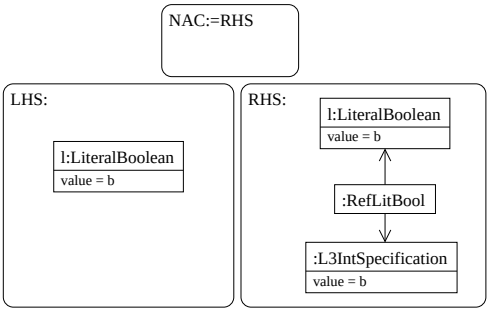

Figure 6.64: Rule: createBoolSpec

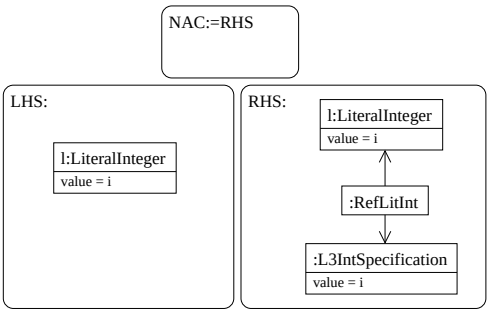

Figure 6.65: Rule: createIntSpec

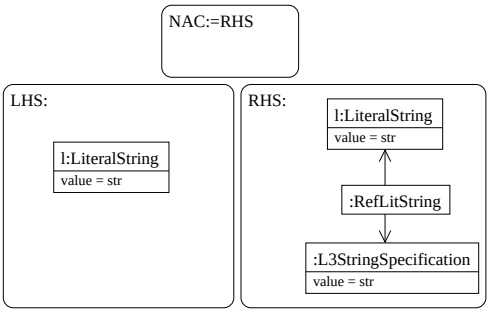

Figure 6.66: Rule: createStringSpec

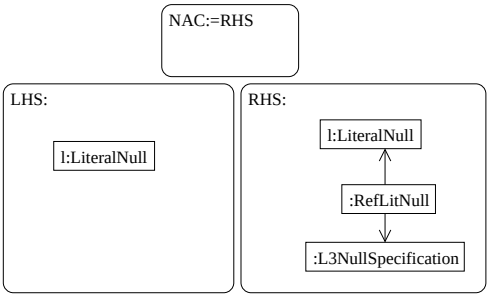

Figure 6.67: Rule: createNullSpec

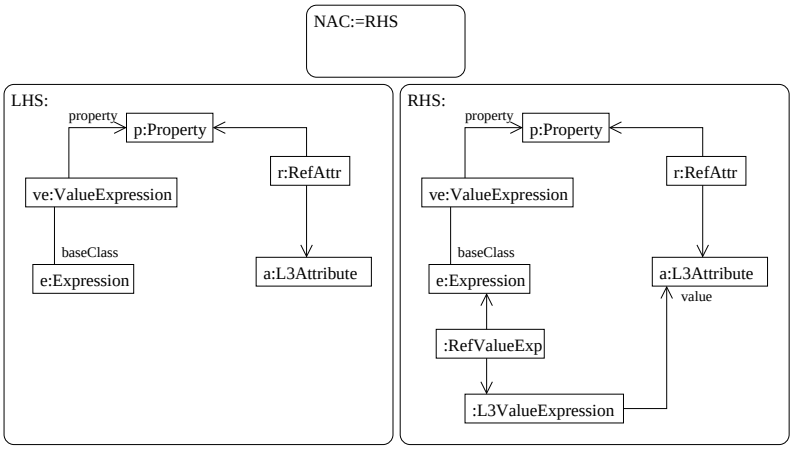

Figure 6.68: Rule: createValueExpFromProperty

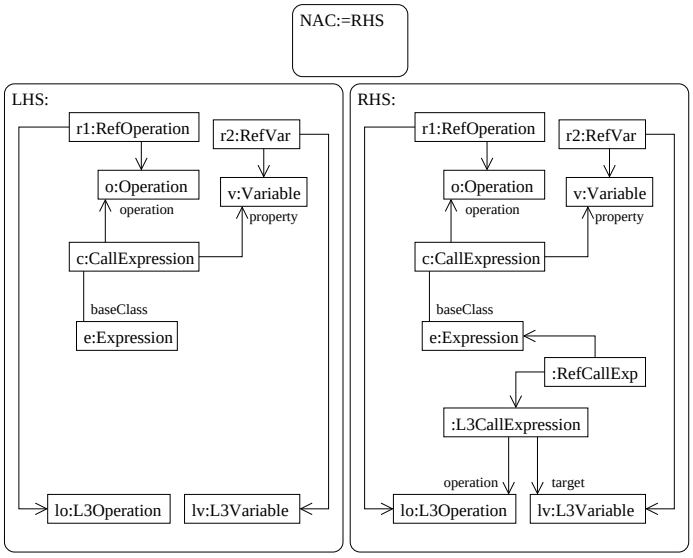

Figure 6.69: Rule: createCallExpFromVariable

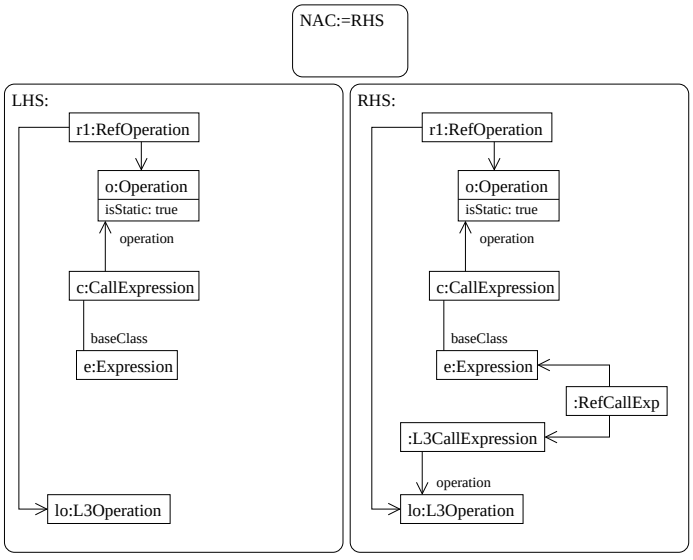

Figure 6.70: Rule: createCallExpStatic

Bibliography

[1] Eclipse IDE. *http://www.eclipse.org.*

[2] Eclipse Modelling Framework. *http://www.eclipse.org/emf.*

[3] Graphical Editing Framework. *http://www.eclipse.org/gef.*

[4] Tiger EMF Transformation Project. *http://tfs.cs.tu-berlin.de/emftrans.*

[5] UML2 Eclipse Plugin. *http://www.eclipse.org/modeling/mdt/?project=uml2.*

[6] *Unified Modeling Language: Superstructure version 2.1.* OMG, 2006.

[7] R. Bardohl, H. Ehrig, J. de Lara, and G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. *LNCS*, 2984:214–228, 2004.

[8] Benjamin Braatz. An Integration Concept for Complex Modelling Techniques. *BME-DAAI Technical Report Series Volume 2006/1*, pages 81–93, 2006.

[9] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. *LNCS 1816*, pages 239–306, 2000.

[10] H. Ehrig, K.Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation.* Monographs in Theoretical Computer Science. Springer Verlag, 2006.

[11] Evans, Cook, Mellor, Warmer, and Wills. Advanced Methods and Tools for a Precise UML. *LNCS 1723*, 1999.

[12] Soon-Kyeong Kim and David Carrington. Formalizing the UML Class Diagram Using Object-Z. *LNCS 1723*, pages 83–98, 1999.

[13] J. M. Küster, S. Sendall, and M. Wahler. Comparing Two Model Transformation Approaches. *Proceedings UML 2004 Workshop OCL and Model Driven Engineering, Lisbon, Portugal*, 2004.

[14] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. *LNCS 2185*, pages 241–256, 2001.

[15] I. Nassi and B. Shneiderman. Flowchart Techniques for Structured Programming. *ACM SIGPLAN Notices 8 (1973)*, pages 12–26, 1973.

[16] H. Störrle. Semantics of UML 2.0 Activities with Data-Flow. *www.pst.informatik.uni-muenchen.de/personen/stoerrle/V/AD2-DataFlow_ UML04.pdf*.

[17] H. Störrle. Towards a Petri-Net Semantics of Data Flow in UML 2.0 Activities. *University of Munich, Technical Report 04*, 2005.

[18] H. Störrle and J.H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. *Software Engineering, SE 2005*, 2005.