Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification

by

Michelle Love Crane

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
January 2009

Abstract

One of the main notational contexts in which model-driven software development has been studied is the Unified Modeling Language (UML), the *de facto* standard in software modelling. The current trend in software development is not just towards the use of models, but the use of *executable* models. In 2006, the Object Management Group issued a Request for Proposal (RFP), soliciting the definition of an Executable UML Foundation, with a fully specified executable semantics. The purpose of such a version of UML is to make the advantages of executable models available to UML users by enabling "a chain of tools that support the construction, verification, translation, and execution" of models.

An oft-voiced criticism of UML is its lack of a formal, unambiguous description of its semantics. In an effort to improve the support for model-driven development, especially with respect to executable modelling, the UML 2 specification introduced a novel three-layer semantics architecture. This architecture provides a stratification of the description of UML models that clearly separates 'low-level' behavioural specification mechanisms, such as actions, from 'high-level' behavioural formalisms, such as activities, state machines and interactions. Although UML describes the effect of actions, it does not provide either the concrete syntax or the formal semantics of an action language.

Our research focuses on a top-to-bottom slice of the three-layer architecture. We formally define the execution semantics of two-thirds of UML actions, including the most complicated actions—invocation actions. Our formal definition is expressed in terms of state changes to a global state machine representing an executing UML model. Our work provides an alternate formalization to that of the current submission to the RFP and could be used to enhance that submission.

To validate our formal semantics and to determine the usefulness of the three-layer architecture, we have created an interpreter for UML actions and activities. This interpreter was designed in accordance with the complex token passing semantics of UML and provides analysis capabilities that have been successfully used to identify problems even in published activity diagrams. In effect, we have created a tool that supports the construction, verification and execution of a subset of UML models, namely activities. Our handling of this slice of the three-layer architecture is a preliminary step to realizing the grander vision of general executable (and analyzable) models.

Co-Authorship

Some parts of Chapters 3, 4 and 5 were written in collaboration with my supervisor Dr. Juergen Dingel and were presented [19] at the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008). The majority of Chapter 7 was written in collaboration with my supervisor and was presented [20] at the 2008 conference of the Centre for Advanced Studies on Collaborative research (CASCON 2008). The grammar in Appendix C appears as part of a technical report [31] at the National Institute of Standards and Technology (NIST). The grammar was created in isolation, although the technical report itself was written in collaboration with two other authors, David Flater, of NIST, and Philippe Martin, of Griffith University.

Acknowledgments

This research is supported by the Centre of Excellence for Research in Adaptive Systems (CERAS) and IBM. An ongoing project, it has also been supported over the years by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Ontario Graduate Scholarship (OGS) program.

Formally, I would like to thank Juergen Dingel for being an inspiring supervisor and mentor. I also appreciate his commitment to excellence and amazing turnaround time, both of which are helping me achieve my goals. I would also like to thank Bran Selic of Malina Software Corporation and Conrad Bock of the National Institute of Standards and Technology for their invaluable assistance with the intricacies of UML.

Informally, I would like to thank the professors, students and staff of the School of Computing for making it such an interesting experience. Most especially, I'd like to thank Bob Tennent, tongue-in-cheek, for scaring me so badly on the very first day of my prep year. Come up with a perfect binary search in 30 minutes, indeed.

I'd like to thank my family and Queen's friends: Mom, the folks, Amber, Dean, Jeremy, Richard, and anyone else I've ever had a long rambling conversation with.

To Val, once again, I thank you for your continued support. We're almost there!

And finally...most of all, this one's for me.

Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. The research was conducted under the supervision of Dr. Juergen Dingel. Any published (or unpublished) ideas and/or techniques of others are fully acknowledged in accordance with the standard referencing practices.

Michelle L. Crane

January 2009

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

There is much interest in the use of models in software development in the hopes of improving productivity by increasing the levels of abstraction, automation, and analysis. One of the main notational contexts in which model-driven software development has been studied is the Unified Modeling Language (UML), the *de facto* standard in software modelling. UML has been very successful: in terms of industrial acceptance, UML is by far the most successful software modelling language. For instance, activity modelling via activity diagrams "was embraced by business-process modelers and by systems engineers, who tend to view many of their systems as interconnecting signal processors" [73].

The current trend is not just towards the use of models, but the use of *executable* models. Early software development methods incorporating the idea of executable models include STATEMATE [38], the Shlaer-Mellor method [75] and ROOM [74]. In all of these methods, executable code is generated automatically from models of behaviour. This feature makes them particularly suitable for the construction of automated testing and simulation environments and explains their popularity for the

development of embedded systems, where the testing of the software on a specific target can be particularly difficult. The purpose of an executable version of UML is to make the advantages of executable models available to UML users by enabling "a chain of tools that support the construction, verification, translation, and execution of computationally complete executable models" [58].[1] Although there are several commercial flavours of executable UML, e.g., xUML [65] and xtUML [50], the official sanction of an executable subset of UML is still an ongoing process [58]. The Object Management Group (OMG) has published a Request for Proposal [58], soliciting a definition of an "Executable UML Foundation". This foundation would be a computationally complete and compact subset of UML, with fully specified executable semantics, which would "serve as a shared semantics foundation that would support additional semantics...covering the higher-level formalisms defined in UML." [58]

1.1 Problem

A key to understanding UML is to remember that entire purpose of a *unified* modeling language is to provide support to multiple paradigms, specifically, "it is intended for use with all development methods, lifecycle stages, application domains, and media" [67]. UML was original developed in an effort to "simplify and consolidate" [67] many emerging object-oriented development methods. However, in recent years, a focus has been on "procedures and processes" [2]. By definition, UML is intended to be a general-purpose visual modelling language and is not confined to any particular

[1] In this case, *computationally complete* is defined such that "the subset shall be sufficiently expressive to allow definition of models that can be executed on a computer either through interpretation or as equivalent computer programs generated from the models through some kind of automated transformations" [58].

design or programming methodology. UML is not a programming language, and does not rely on any existing programming language to specify behaviour. Instead, the very 'low-level' concept *actions* is introduced, where an action is a "fundamental unit of behaviour" [59, §11.1]. These actions can be combined to form larger behaviours in different ways. UML supports three 'high-level' behavioural formalisms: activities, state machines and interactions. Activities are heavily influenced by business process modeling notations. State machines, used often in computer science and electrical engineering, are based on Harel's statecharts [37]. Finally, while state machines describe the behaviour of individual objects, interactions are used to model the behaviour of a set of objects [67].

An oft-voiced criticism of UML is its lack of a formal, unambiguous description of its semantics. This lack prevents UML from affording its users more automation and analysis gains. In an effort to improve the support for model-driven development, especially with respect to executable modelling, the UML 2 specification introduced a novel three-layer semantics architecture [59, 71]. This architecture, discussed in further detail in Section 2.1, provides a stratification of the description of UML models that clearly separates 'low-level' (i.e., primitive) behavioural specification mechanisms, such as actions, from 'high-level' (i.e., derived) behavioural formalisms, such as activities, state machines and interactions. While this architecture helps clarify the runtime semantics of UML, a comprehensive formalization of the semantics is still an open research problem.

One complicating factor is the large number of features that each diagram type typically supports. Activities, for instance, allow for different kinds of flow (e.g., data, control), different kinds of behaviour invocation (e.g., direct or indirect, synchronous

or asynchronous), a variety of different kinds of control node (e.g., initial, final, fork, join, decision, merge), structured concepts (e.g., loop, conditional), and mechanisms for unstructured control flow (e.g., interruptible regions, exceptions). Additionally, the semantics of activities has changed from a special kind of state machine to a rather complex token offer semantics, which generalizes Petri net semantics and contains features ssuch as deadlock avoidance rules and competition.

1.2 Thesis and Scope of Research

Thesis Statement: The layered semantic architecture (as described in the UML specification [59]) defines a useful framework for formally[2] defining and implementing behavioural semantics in UML.

To this end, we flesh out, and indeed implement, a top-to-bottom slice of the three-layer architecture. At the 'high-level', we implement an interpreter for UML activities, one of UML's behavioural formalisms. Activities are a natural fit for composing UML actions, which represent the 'low-level' behavioural mechanisms of UML. We formally define the execution semantics of actions with respect to a particular semantic domain, and incorporate both actions and the semantic domain into our interpreter. Our handling of this slice of the three-layer architecture is a preliminary step to realizing the grander vision of general executable (and analyzable) models.

The scope of our research is constrained by the following:

- Our chosen semantic domain, referred to as the *System Model* [13, 14, 15], is

[2]By *formally*, we simply mean *precisely* and *unambiguously* [39].

described in greater detail in Chapters 2 and 3. It is the culmination of a two-year project [79] researching the semantics of UML, in which we were involved. The System Model was designed by our research partners as a deliberately general representation of the structural foundation of UML. This generality made it possible to implement the System Model as part of our interpreter. Our research provides an excellent opportunity to evaluate the System Model itself.

- UML actions represent the fundamental units of behaviour in UML. We formally define two-thirds of these actions and implement slightly fewer in our interpreter. Several other actions have been discounted due to limitations in the System Model; a few more are not included for simplicity.

- Of the three high-level behavioural formalisms provided by UML, we chose activities, as they are a natural fit for composing actions into larger executions. We support a large subset of activities, including: actions, pins, all control nodes, sequential and concurrent execution, and the rather delicate token passing semantics peculiar to UML 2 activities.

1.3 Organization of Thesis

The remaining chapters of the thesis are organized as follows: Chapter 2 provides background, including briefly describing UML actions and activities. Chapter 3 describes the semantic domain, i.e., the System Model. Chapter 4 formally describes the structure of UML activities. Chapter 5 details the formal description of UML

actions, focusing on the semantics of action execution. Chapter 6 describes the execution of activities as a whole, including token passing. Chapter 7 describes our action and activity interpreter, including running examples to demonstrate its analysis capabilities. Chapter 8 continues the discussion about our interpreter, with several illustrative examples. Chapter 9 provides a discussion of related work. Finally, Chapter 10 concludes with a discussion of our contributions and future work.

Chapter 2

Background

2.1 UML Semantics Architecture

The Unified Modeling Language (UML) is a "general-purpose visual modeling language" [67] that can be used in the analysis and design of software systems. Although well-documented, UML does not yet have a formal semantics. The abstract syntax is carefully laid out in almost a thousand pages of specification, but the meaning of the syntactic elements is discussed in prose, with a smattering of Object Constraint Language (OCL) constraints. The runtime semantics of UML is defined as a "mapping of modeling concepts into corresponding execution" [59, §6.3]. This semantics is not formally defined in the UML specification. Instead, the specification outlines a three-layer semantics architecture, which "identifies key semantic areas...and how they relate to each other" [59, §6.3.2].

The three-layer architecture is shown in Figure 2.1. Each layer depends on those below it, but not vice versa. The bottom layer represents the structural foundations of UML, including concepts such as values, objects, links, messages, etc. The middle

layer is the behavioural base, which contains mechanisms for individual object behaviour, as well as behaviour involving more than one object. More importantly, this layer also contains the description of a set of UML actions. The top layer represents different behavioural formalisms in UML, all of which rely on the behavioural base. Activities, state machines, and interactions all make use of the actions in order to express behaviour; these actions are explained in terms of constructs in the structural foundation.

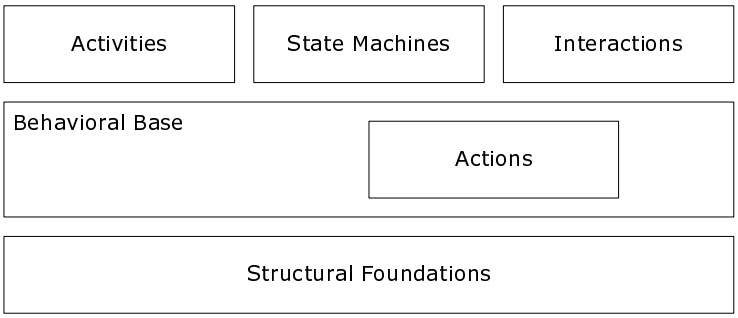


Figure 2.1: The UML three-layer semantics architecture

The key to this architecture lies in the middle layer, i.e., actions. A fundamental premise of UML behavioural semantics is the assumption that "all behavior...is ultimately caused by actions" [59, §6.3.1]. Actions are "fundamental units of behavior" [59, §6.3.2]. As an analogy, actions are comparable to "executable instructions in traditional programming languages" [59, §6.3.2].

The advantage of this approach to defining the runtime semantics is the fact that once UML actions are clearly mapped to the structural foundation, it should be relatively easy to define the semantics of different behavioural formalisms.

2.2 Semantic Domain

Our formalization leverages a previously developed semantic domain called the *System Model*. Generally, the meaning of a UML specification is given by "the constraints that models place on the runtime behavior of the specified system" [13]. Described in a series of three documents [13, 14, 15], the goal of the System Model is to allow for these constraints to be captured and collected in their purest and most general form. Therefore, the System Model description intentionally avoids the use of existing formal specification notations such as Z [76] or Abstract State Machines [12] and relies solely on simple mathematical concepts such as sets, relations, and functions.

The System Model contains a formalization of UML's structural foundation (i.e., the bottom layer of the architecture in Figure 2.1), which results in a formal notion of state. The meaning of a single behavioural diagram is captured by a (possibly timed) state machine. The meaning of a collection of behavioural UML diagrams is given by the composition of the state machines for each diagram.

We discuss the System Model in more detail in Chapter 3.

2.3 UML Actions

Actions were originally introduced in UML 1.5 as a "set of primitive actions that model manipulation of objects and links as well as computation and communication among objects" [67]. In UML 2.0, activities were brought more in line with these actions to produce a more procedural model [67]. There are a total of 45 actions defined in UML, 36 of which are concrete.[1] As mentioned earlier, a UML action is a

[1]UML defines nine *abstract* actions, such as Action, InvocationAction, CallAction, etc., which are used to organize all actions into a hierarchy and not intended to be instantiated. The remaining

"fundamental unit of behavior" [59, §11.1]. Equivalent to an executable programming language statement, an action can be thought of as a black box, taking in a set of inputs and converting them into a set of outputs.

UML actions are divided into four packages, roughly corresponding to level of complexity:

- **Basic actions** are those that perform operation calls, send signals, and invoke behaviour.
- **Intermediate actions** are primitive actions that either perform a computation or access object memory. This group includes actions that create and delete objects and structural features, as well as read values and test identity.
- **Complete actions** deal with the "relation between object and class and link objects" [59, §11.1]. These actions can read the instances of a classifier, determine the classifier of a particular object, and work with links and associations. In addition, the actions used to accept events or calls are included in this package.
- **Structured actions** "operate in the context of activities and structured nodes" [59, §11.1], and include actions that operate on variables (e.g., local to an activity) and raise exceptions.

Another way to categorize actions is by what part of the model they affect. For instance, the UML actions can be segregated into the following functional groups:

- **Communication** actions, used for communication between objects.

actions are concrete and can be instantiated.

- **Object** actions, used to create, destroy, and manipulate objects.
- **Structural feature** actions, used to modify the attributes of objects.
- **Link** actions, used to create, modify and destroy associations and links.
- **Variable** actions, used to create, modify and destroy variables.
- **Other** actions, encompassing miscellaneous functionality.

Table 2.1 shows the 45 UML actions, categorized along two dimensions: the official UML package in which the actions are specified, and the informal grouping according to function [72].

Table 2.1: UML's 45 actions, categorized along two dimensions: by package (columns) and by function (rows). Abstract actions are shown in italics

	Basic	**Intermediate**	**Complete**	**Structured**
Communication	*InvocationAction* *CallAction* CallBehaviorAction CallOperationAction SendSignalAction	BroadcastSignalAction SendObjectAction	ReplyAction UnmarshallAction AcceptEventAction AcceptCallAction	
Object		CreateObjectAction DestroyObjectAction ReadSelfAction TestIdentityAction	ReadExtentAction ReadIsClassifiedObjectAction ReclassifyObjectAction StartClassifierBehaviorAction	
Structural Feature		*StructuralFeatureAction* ClearStructuralFeatureAction ReadStructuralFeatureAction *WriteStructuralFeatureAction* AddStructuralFeatureValueAction RemoveStructuralFeatureValueAction		
Link		*LinkAction* ReadLinkAction *WriteLinkAction* CreateLinkAction DestroyLinkAction ClearAssociationAction	CreateLinkObjectAction ReadLinkObjectEndAction ReadLinkObjectEndQualifierAction	
Variable				*VariableAction* *ClearVariableAction* ReadVariableAction *WriteVariableAction* AddVariableValueAction RemoveVariableValueAction
Other	*Action* OpaqueAction	ValueSpecificationAction	ReduceAction	RaiseExceptionAction

2.3.1 Description of Actions

All actions ultimately inherit from the abstract Action class. Here, we show the actions in each functional group, and provide a brief description of each. Abstract actions are shown in italics. Not all actions will be formally defined. Those that will not be defined are marked with an 'x'. The others are discussed in further detail in Chapter 5.

2.3.1.1 Communication Actions

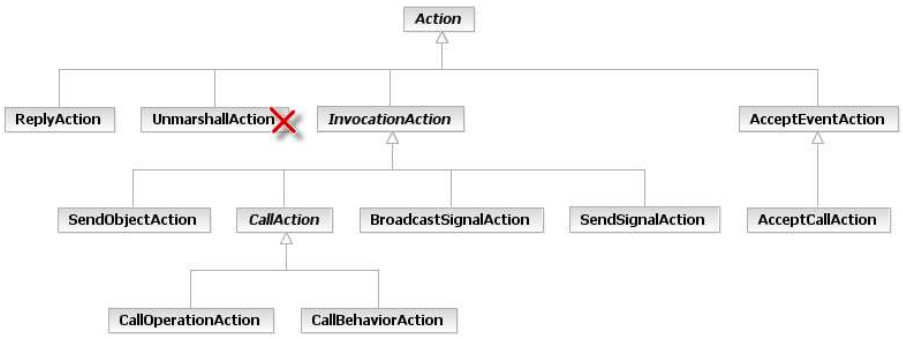


Figure 2.2: Hierarchy of communication actions

AcceptCallAction This action represents the receipt of a synchronous call request, i.e., it waits for a synchronous call [59, §11.3.1].

AcceptEventAction This action waits for an event, such as a signal or an asynchronous call [59, §11.3.2].

BroadcastSignalAction This invocation action broadcasts a signal [59, §11.3.7].

CallAction This is an abstract class for actions that invoke behaviour and receive return values [59, §11.3.8].

CallBehaviorAction This call action invokes behaviour directly. It may be used synchronously or asynchronously [59, §11.3.9].

CallOperationAction This call action invokes behaviour indirectly, e.g., a call request is submitted to a target, where it may invoke behaviour. It may be used synchronously or asynchronously [59, §11.3.10].

InvocationAction This is an abstract class for invocation actions [59, §11.3.20].

SendObjectAction This invocation action sends an object to a target [59, §11.3.44].

SendSignalAction This invocation action sends a signal to a target [59, §11.3.45].

ReplyAction This action returns values to a calling action [59, §11.3.43].

UnmarshallAction This action breaks an object into its component structural feature parts [59, §11.3.49]. For simplicity, this action will not be formally defined.

2.3.1.2 Object Actions

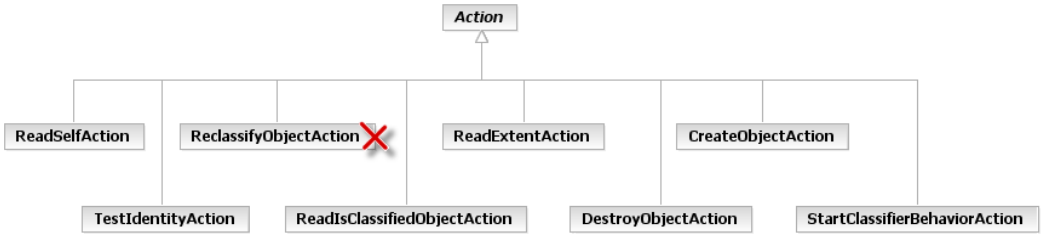


Figure 2.3: Hierarchy of object actions

CreateObjectAction This action creates an object [59, §11.3.16].

DestroyObjectAction This action destroys an object [59, §11.3.18].

ReadExtentAction This action retrieves the current instances of a classifier [59, §11.3.31].

ReadIsClassifiedObjectAction This action determines whether or not a given object is an instance of a particular classifier [59, §11.3.32].

ReadSelfAction This action retrieves the host object of an action [59, §11.3.36].

ReclassifyObjectAction This action changes the classifier(s) of an object [59, §11.3.39]. For simplicity, this action will not be formally defined.

StartClassifierBehaviorAction This action starts the classifier behaviour of an object [59, §11.3.46].

TestIdentityAction This action tests if two objects are identical [59, §11.3.48].

2.3.1.3 Structural Feature Actions

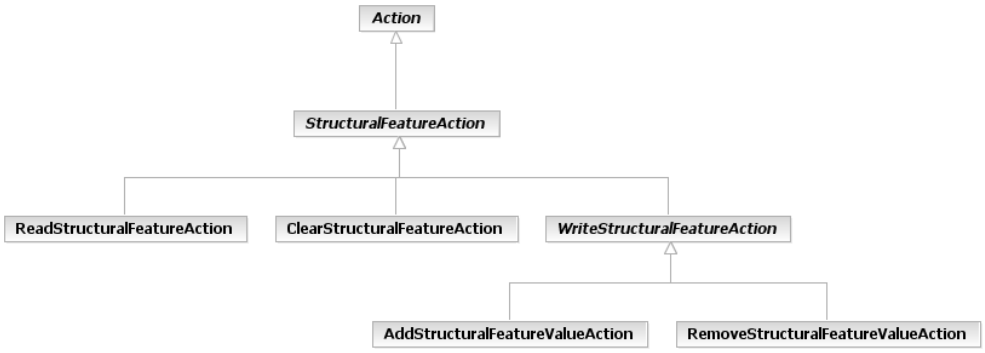


Figure 2.4: Hierarchy of structural feature actions

AddStructuralFeatureValueAction This action adds a value to a structural feature [59, §11.3.5].

ClearStructuralFeatureAction This action removes all values from a structural feature [59, §11.3.12].

ReadStructuralFeatureAction This action retrieves the values of a structural feature [59, §11.3.37].

RemoveStructuralFeatureValueAction This action removes values from a structural feature [59, §11.3.41].

StructuralFeatureAction This is an abstract class for all actions dealing with structural features, i.e., attributes of objects [59, §11.3.47].

WriteStructuralFeatureAction This is an abstract class for actions modifying structural features [59, §11.3.54].

2.3.1.4 Link Actions

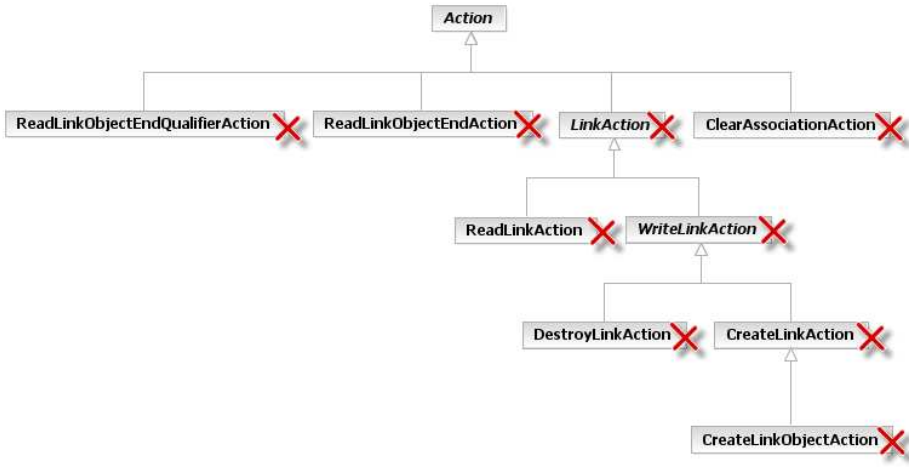


Figure 2.5: Hierarchy of link actions

The System Model does not support the concept of links as instances of associations. For that reason, none of the link actions will be formally defined.

ClearAssociationAction This action destroys all links of an association [59, §11.3.11].

CreateLinkAction This action creates a link [59, §11.3.14].

CreateLinkObjectAction This action creates a link object; it used only for creating links of association classes [59, §11.3.15].

DestroyLinkAction This action destroys links and link objects [59, §11.3.17].

LinkAction This is an abstract class for all actions dealing with links [59, §11.3.21]. Links are instances of associations and association classes.

ReadLinkAction This action navigates across an association to retrieve objects at one end [59, §11.3.33].

ReadLinkObjectEndAction This action retrieves an end object from a link object [59, §11.3.34].

ReadLinkObjectEndQualifierAction This action retrieves a qualifier end value from a link object [59, §11.3.35].

WriteLinkAction This is an abstract class for actions that modify links [59, §11.3.53].

2.3.1.5 Variable Actions

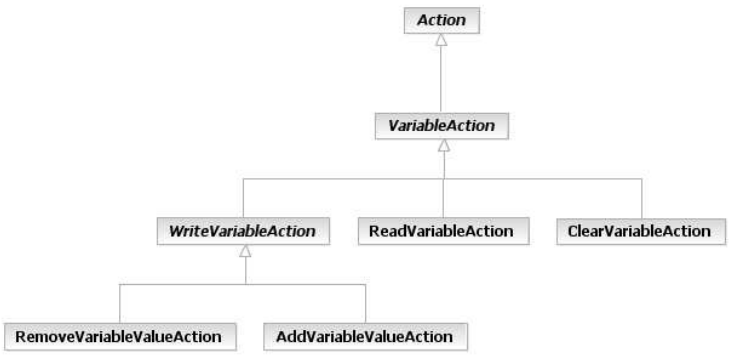


Figure 2.6: Hierarchy of variable actions

AddVariableValueAction This action adds values to a variable [59, §11.3.6].

ClearVariableAction This action removes all values from a variable [59, §11.3.13].

ReadVariableAction This action retrieves the values of a variable [59, §11.3.38].

RemoveVariableValueAction This action removes values from a variable [59, §11.3.42].

VariableAction This is an abstract class for actions that act on statically specified variables [59, §11.3.52].

WriteVariableAction This is an abstract class for actions that modify variable values [59, §11.3.55].

2.3.1.6 Other

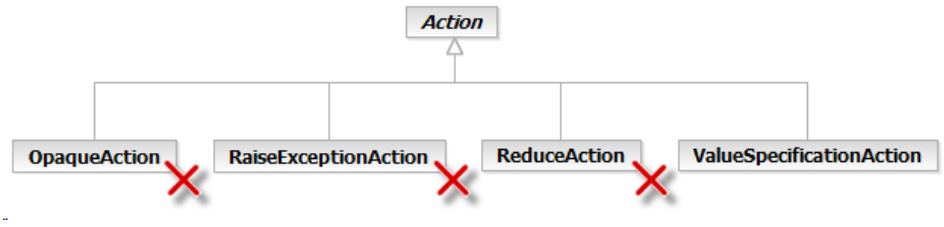


Figure 2.7: Hierarchy of other actions

OpaqueAction This action represents user-defined or implementation-specific behaviour [59, §11.3.26]. For simplicity, this action will not be formally defined.

RaiseExceptionAction This action causes an exception to occur [59, §11.3.30]. We do not support the concept of exceptions, so this action will not be formally defined.

ReduceAction This action reduces a collection to a single value [59, §11.3.40]. For simplicity, this action will not be formally defined.

ValueSpecificationAction This action evaluates a value specification [59, §11.3.51].

2.3.2 Anatomy of an Action

Actions are used to change the global state of the modelled system. They can be thought of as black boxes, with inputs and outputs, and sometimes having side effects. The inputs and outputs are represented by pins, specifically InputPins and OutputPins. Pins are typed elements that "provide values to actions and accept result values from them" [59, §11.3.28]. Pins are used for passing dynamic information, i.e., information that becomes available or changes as the activity is executed. On the other hand, an action may also have attributes or associations, which provide it with static information. This static information is necessary for the execution of the activity.

Consider, for instance, the ReadIsClassifiedObjectAction. Figure 2.8 shows the UML metamodel of this action, indicating its static attributes and associations and dynamic pins. The ReadIsClassifiedObjectAction takes as input an object, compares that object to a statically-defined classifier and returns a result based on whether or not that object is an instance of that classifier. There is an additional static attribute, isDirect; when true, the action only returns a positive result if the object is a direct instance of the classifier (as opposed to an instance of one of the classifier's subclasses).

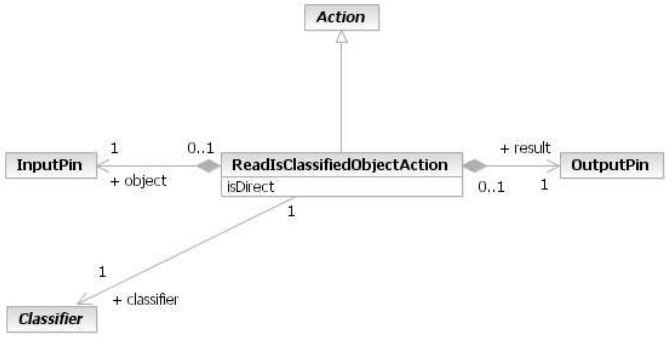


Figure 2.8: UML metamodel of ReadIsClassifiedObjectAction. This action has a static attribute (isDirect), a static association (classifier), a dynamic input pin (object) and a dynamic output pin (result)

2.3.3 Surface Action Language

The UML specification describes the *action semantics*, which describes the effects of the actions discussed above; however, it does not define a formal semantics or a concrete syntax [33] for actions. In fact, the action semantics could be viewed as an assembly language and therefore would be more useful as a foundation, but needing a *surface action language* at a higher level of abstraction [67]. This action language would "encompass both primitive actions and the control mechanisms" [59, §11.1] of UML's behavioural formalisms. The surface action language could "implement each semantic construct one-to-one, or it could define higher-level, composite constructs" [59, §11.1] to simplify modelling.

As there is no standard action language, many have been created, both for commercial use (where the focus is mainly on executable UML) and academic research (where the focus tends to be on examining semantics or analyzing UML models). We are interested in semantics, specifically of the individual actions; in order to examine the execution of actions, we too have required the use of an action language. UML

activities are a natural fit as our surface language, as they encompass both actions (be they seemingly high-level abstractions, such as "Pay Bills" or low-level primitives, such as CreateObjectAction) and control mechanisms. In this case, the surface language does not offer higher-level abstraction, but rather the one-to-one mapping discussed above.

2.4 UML Activities

Activity modelling focuses on the "sequence and conditions for coordinating lower-level behaviors" [59, §12.1]. An activity is a "graph of nodes and flows that shows the flow of control (and optionally data) through the steps of a computation" [67]. In UML 1.5, activities were considered a special form of state machine, albeit with a modified notation in order to appear like a flow diagram [2]. However, the underlying state machine semantics was restrictive and confusing. UML 2.0 redefined activities to be true flow diagrams, and integrated them with actions, thereby supporting true data and control flow modelling [2].

2.4.1 Activity Notation

Activity definitions are shown in *activity diagrams*. These diagrams contain action nodes, control nodes, object nodes, as well as control flow and data flow edges. The descriptions provided here are simply intended to introduce the reader to the basic notation of activities; additional constraints and semantics are discussed in the UML specification [59, Chapter 12].

2.4.1.1 Action Nodes

Action nodes are shown as rounded rectangles (see Figure 2.9) and refer to 'predefined' actions, such as CreateObjectAction. Although it may seem that many activities include high-level abstractions, such as "Create Car" or "Pay Bills", these actions are actually predefined UML actions. For instance, action nodes can be given labels that are more descriptive than the predefined names, such as "Create Car" instead of CreateObjectAction [2]. Also, the label "Pay Bills" simply represents an invocation of a user-defined behaviour, e.g., a CallBehaviorAction [2].

Figure 2.9: Action nodes

2.4.1.2 Control Nodes

There are several control nodes, used to start and stop flow, synchronize concurrency, as well as provide branching. Note that both data (including objects) and control may flow through control nodes.

The *initial node* (Figure 2.10) is used to start flow; there may be several initial nodes in an activity.

Figure 2.10: Initial node

Activity final nodes (Figure 2.11) and *flow final nodes* (Figure 2.12) are used to stop flows. When an activity final node is reached, all flows in the activity cease. On the other hand, when a flow final node is reached, only that particular flow is halted. There may be multiple activity final and flow final nodes in an activity.

Figure 2.11: Activity final node

Figure 2.12: Flow final node

Fork nodes (Figure 2.13) provide explicit support for concurrency. When a token (data or control) reaches a fork, it is duplicated[2] and sent along all outgoing edges. There is a restriction in that the flows into and out of a fork must be consistent, i.e., the edges coming into and out of a fork node must be either all data flow edges or all control flow edges.

Join nodes (Figure 2.14) provide support for synchronization in an activity. Tokens do not flow through a join unless it receives tokens from all of its incoming edges. If all edges carry control tokens, then one control token is offered to the outgoing edge. If there are some incoming data tokens as well, then these (but no control tokens) are offered to the outgoing edge, in the order in which they reached the join.

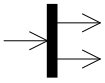


Figure 2.13: Fork node

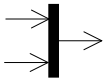


Figure 2.14: Join node

[2]Although not clear from the specification, the duplication is not a 'deep' copy; rather, the reference to an object is duplicated [8].

Decision nodes (Figure 2.15) are used for branching between alternate flows; guards on the outgoing edges are evaluated to determine which edge to take—a token may travel along only one outgoing edge. The flows into and out of a decision must be consistent, i.e., the edges coming into or out of a decision must be either all data flow edges or all control flow edges.

Merge nodes (Figure 2.16) are used to bring together alternate flows. Unlike the join node, there is no concurrency synchronization. Rather, any token coming into the merge will be passed to the outgoing edge. Again, the flows into and out of the merge must be consistent, i.e., the edges coming into or out of a merge must be either all data flow edges or all control flow edges.

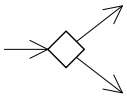


Figure 2.15: Decision node

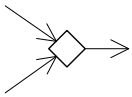


Figure 2.16: Merge node

2.4.1.3 Object Nodes

Object nodes are shown as rectangles; see Figure 2.17. They can be used to hold data (including objects) temporarily and encompass the notion of *pins*, which are object nodes used to provide "inputs and outputs to actions" [59, §12.3.44].

(a) Object node

(b) Object node as pin

Figure 2.17: Object nodes

Action nodes with pins attached are commonly found in activity diagrams. In reality, these structures are composed of an action node and an object node connected by an edge.[3] In order to make this relationship explicit, we make use of the *standalone pin notation* [59, Figure 12.120], which is equivalent to the standard pin notation. Consider the two activities in Figure 2.18. In the standalone pin notation, object nodes in an activity represent pins, i.e., an object node represents both the output pin of one action and the input pin of another action.

(a) Standard notation (b) Standalone notation

Figure 2.18: The standalone pin notation explicitly defines pins as object nodes between actions

2.4.1.4 Control and Object Flows

An activity coordinates the execution of actions using a "control and data flow model" [59, §12.3.4]. Control flows sequence the execution of nodes while object flows sequence data produced by one node and used by another [59, §12.3.4]. Control nodes are used to structure both types of flow.

Both control and object flow are represented as edges in an activity diagram; they appear identical but are "distinguished by usage" [2]. Consider the activities in Figure 2.19. Figure 2.19(a) shows a control flow edge; it connects two actions.

[3]These specific edges are not actually represented in the current UML specification, although they are necessary for transferring tokens between actions and their pins [8].

Figure 2.19(b) shows an object flow edge; it connects two pins. Note that the two arrows in Figure 2.18(b) "denote a single object flow edge between two pins in the underlying model" [59, §12.3.37].

(a) Control flow edge (b) Object flow edge

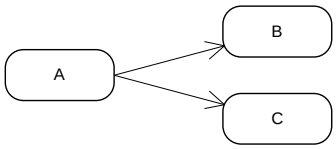
Figure 2.19: Control and object flow edges are distinguished by usage

2.4.2 Token Semantics

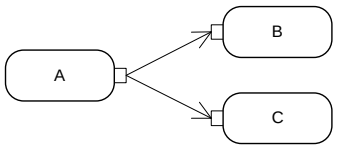
The semantics of activity execution depends on the *flow* of *tokens*. Each token is unique, contains "an object, datum, or locus of control" [59, §12.3.4], and is associated with a particular node in the activity. *Flow* means that the execution of each node in the activity "affects, and is affected by, the execution of other nodes" [59, §12.3.4] in the activity. It is important to note that tokens are not pushed through an activity. Instead, they are offered by a source node and either accepted or rejected by target nodes. In this manner, they move through an activity in accordance to a "traverse-to-completion" [5] semantics. In essence, a token only moves to a target if it can be used there immediately; the purpose of this strategy is to help avoid deadlock. The details of token semantics are quite complicated; some of the constraints on tokens are:

- Tokens move between action nodes. With few exceptions, they do not rest on control or object nodes.
- An action cannot begin execution until it has received sufficient token offers along each of its incoming edges.
- Tokens can only move along an edge if they satisfy the restrictions of the source node, the target node, and the edge itself. Restrictions include concepts such as multiplicities on the input/output pins and weights along the edge.
- It is possible that a single token is offered to multiple edges; however, it will only traverse one edge.
- Tokens can be duplicated, e.g., by a fork node. They can also be combined, e.g., by a join node.

Competition One of the reasons to distinguish between control and object flows is because competition is possible with object flows, but not with control flows. Consider the activities in Figure 2.20. The activity in Figure 2.20(a) contains only control flows. When action `A` finishes execution, it offers control tokens to both actions `B` and `C`. These control tokens can be accepted by both targets, i.e., both `B` and `C` can execute. On the other hand, the activity in Figure 2.20(b) contains only object flows. When `A` finishes execution, it offers a token (through its output pin) to both targets `B` and `C`, through their input pins. In this case though, *only one* of the targets can accept the offer and thus receive the token (with its data). The other outstanding offer will be immediately revoked. In other words, when an object node has multiple targets, competition occurs.



(a) Action with control flow to two targets; there is no competition. Both `B` and `C` can execute



(b) Action with object flow to two targets; there is competition. Only one of `B` and `C` can execute

Figure 2.20: Two similar activities, with vastly different token passing

2.4.3 Activities vs. Other Notations

2.4.3.1 Activities vs. Flowcharts

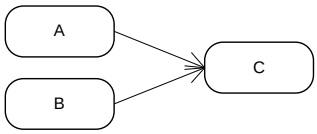
As of UML 2.0, activity diagrams are now considered true flow diagrams. As such, they are similar in appearance to other flow diagram notations, such as flowcharts. In fact, the activity notation was "heavily influenced by various business process modeling notations" [67]. That said, UML activities are not identical to any other notation, since "no single business process modeling notation was dominant" [67].

The similarity between activities and flowcharts can cause readers to misinterpret activities if they are not familiar with the subtle semantics of activities. Specifically, UML activities support concurrent execution (with or without synchronization), as well as the sequential execution and branching supported by flowcharts [67].

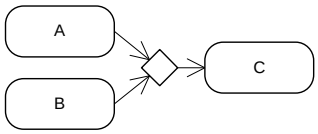
The following rules must be kept in mind when interpreting a UML activity:

- In an activity, when an action has multiple flows incoming, the action will not begin execution until it receives tokens along *each* of the incoming edges. Consider the activity in Figure 2.21(a). Action `C` will only execute after both `A` and `B` have completed execution. If it were desirable to have `C` execute after

either `A` or `B`, then the modeler would have to make use of a merge node, as shown in Figure 2.21(b).



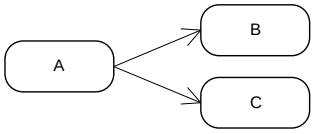
(a) Action `C` can execute only after both `A` and `B` have finished execution



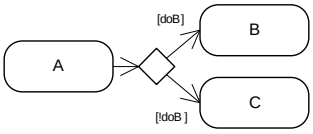
(b) A merge node can be used to trigger `C` after either `A` or `B` have finished execution

Figure 2.21: An action must receive tokens along all incoming edges before it can execute

- As discussed earlier, when there are control flows leaving an action, there is no competition. In other words, control tokens will be offered along all outgoing edges. Consider the action in Figure 2.22(a). Actions `B` and `C` can both execute after `A` has finished execution. If it were desirable to have only one of the two targets execute, the modeler would have to use a decision node, as shown in Figure 2.22(b).



(a) `B` and `C` can both execute after `A` finishes execution



(b) Only one of `B` and `C` can execute after `A` finishes execution

Figure 2.22: An action offers control tokens to all of its outgoing control edges

2.4.3.2 Activities vs. Petri Nets

As of UML 2.0, activities have been "redesigned to use a Petri-like semantics" [59, §12.3.4]. Specifically, the concept of token offering is a generalization of Petri net

transition enablement in the following ways [11] :

- In an activity, tokens move concurrently, and asynchronously. In a Petri net, tokens move all at the same time, i.e., in lock-step.
- In an activity, the decision of which token to move, i.e., which transition to fire, can involve multiple unconnected edges and nodes.

Research has been conducted on relating UML activities to Petri nets. While the mapping of basic constructs, e.g., actions to places and edges to transitions is straightforward, the mapping does not scale to include UML activities at large [77, 70]. Specifically, the concepts of exceptions and streaming do not translate well to Petri nets. More importantly, the notion of traverse-to-completion semantics for tokens cannot be handled by Petri nets. In activities, action nodes can be connected along paths which include numerous control nodes; to translate these to Petri nets, the entire path followed by a token would act as a single Petri net transition. It could be possible to manually create an equivalent Petri net to represent any particular activity. However, to create a generalized transformation would require knowledge of the semantics for all cases and the embedding of that knowledge in the transformation [77].

Chapter 3

System Model

In this chapter we introduce notation conventions used throughout the rest of the document. The remainder of the chapter is devoted to describing the System Model, used as our semantic domain.

3.1 Document Conventions

We make use of the following conventions in this document:

- The *italic* font will be used to indicate specific concepts in our semantic domain and the mapping to it, e.g., *UVAL*.
- The sans serif font will be used to indicate elements from the UML specification, e.g., CreateObjectAction.
- The `typewriter` font will be used to indicate elements from example activities (e.g., action `A`), as well as concepts related to our interpreter (e.g., the `Scheduler` class).

3.1.0.3 Common Sets

We make use of common sets, such as *Boolean* and *String*. These sets are types (called *type names* in the System Model) whose *carrier sets* (i.e., underlying set of values) contain values:

- *Boolean* is a type name with two values $\{true, false\}$ in its carrier set, where $true \neq false$.
- *Identifier* is a type name. We assume that the carrier set of *Identifier* contains all identifiers.
- *String* is a type name. We assume that the carrier set of *String* contains all strings.

3.1.0.4 List Data Structure

We define a list data structure that can contain an ordered collection of elements. For example, $List(UVAL)$ is a list of values and $List(UTYPE)$ is a list of type names. A list may be explicitly specified, e.g., $(v_1, v_2, ..., v_n)$ for a list containing n elements, where $n \geq 0$. A list may contain duplicates. It is possible to access specific elements of the list, e.g., for the list $(v_1, v_2, ..., v_n)$, v_i is the i^{th} element of the list.

3.1.0.5 Dot Notation

Dot (.) notation is used as shorthand for projections on particular fields of a type. For example, an action node *an* is defined as a tuple $(id, in, out, action, annot, waiting, owner, stalled)$. The expression $an.action$ retrieves the *action* element of the tuple, i.e., the action instance represented by the node. The action is itself defined by a tuple

$(name, inPins, outPins, attrs)$. The expression $an.action.inPins$ denotes the function recording the current values on the input pins of the action instance. This convention is used throughout the thesis.

3.1.0.6 Operators

We make use of the following operators:

- The $\oplus$ operator is used in the System Model documentation and is used to add a tuple to a binary function. For instance, $f \oplus [a_1 = b_1][a_2 = b_2]...[a_n = b_n]$ adds the mappings $a_1 \mapsto b_1, a_2 \mapsto b_2, ..., a_n \mapsto b_n$ to the function f. Note that the bindings are from left to right, i.e., first the mapping $a_1 \mapsto b_1$ is added, then $a_2 \mapsto b_2$, etc. Also, if a_i is already in the domain of f, the $\oplus$ operator *overwrites* the old mapping, replacing it with the new one.

- The & operator is used for string concatenation, e.g., the result of $x\&y$, where $x, y \in String$, is the string composed of x immediately followed by y. We assume that all elements have a string representation, e.g., if i is an integer then $x\&i$ is the string composed of x immediately followed by the string representation of i.

In addition, we utilize standard mathematical notation, such as $\in$ for set membership, $\subseteq$ for subset, $\cup$ for set-theoretic union, $\cap$ for set-theoretic intersection, $\setminus$ for set-theoretic complement, $\left| \ldots \right|$ to indicate the size of a set, $\{\}$ to indicate an empty set, $\rightarrow$ as the function arrow, $\mapsto$ for maps to, $\forall$ for universal quantification, $\exists$ for existential quantification, $\exists!$ for uniqueness quantification, ! for negation, etc.

3.2 Structure

The System Model is officially defined in a set of three documents, covering structure [13], control [14], and the underlying state machine [15].

A key rationale in the development of the System Model was to avoid, as much as possible, biases inherent in other specification formalisms. To that end, the System Model is described with mathematics, specifically sets, functions and relations. This lack of even the most basic class diagrams can make deciphering the System Model time-consuming. For that reason, one of our early research objectives was to model the structure of the System Model with UML class diagrams. These diagrams were useful not only in understanding and explaining the System Model, but also in implementing it.

The reader is encouraged to refer to the System Model documentation [13, 14, 15] to get a full, mathematical description of all System Model concepts. In this chapter, we present the essentials required to understand the formal descriptions presented in Chapters 4 and 5. More precisely, we describe the concepts required to understand the notion of *state* used in the System Model.

The UML-based Specification Environment (USE) [80] is a tool used for the specification and analysis of systems. It is discussed in more detail in Chapters 7 and 8. As we modelled the System Model in UML, we utilized the USE tool to visualize instances of the System Model universe. We include several USE diagrams below to illustrate, by example, the structure of the System Model.

3.2.1 Universes

The System Model describes "the universe (set) of all possible semantic structures... The semantic mapping interprets a UML model as a predicate that restricts the universe to a certain set of structures, which represents the meaning of the UML model" [13]. "The system model $SYSMOD$ is the universe of state machines" [13]. In general, we refer to a state machine $sm \in SYSMOD$, where the state machine represents the behaviour of the UML model under examination. There are several universes with which we are concerned. Figure 3.1 shows how these universes relate to each other. The elements composing these universes are discussed in more detail later in this section.

- $SYSMOD$: The universe of state machines. We focus on one state machine, $sm \in SYSMOD$, representing the current model under examination.
- $UTYPE$:[1] The universe of type names. A type name "identifies a carrier set, which contains...data elements" [13] or values associated with the type name. For instance, the carrier set of type name *Int* could contain the values $\{1, 2, 3, ...\}$. The relation $CAR : UTYPE \rightarrow \mathcal{P}\, UVAL$ maps type names to associated carrier sets.
- $UVAL$: The universe of values.
- $UVAR$: The universe of variable names.
- $ULOC$: The universe of locations. $ULOC \subseteq UVAL$.
- $UCLASS$: The universe of class names. $UCLASS \subseteq UTYPE$.

[1] Note that $UTYPE$ is actually shorthand for $sm.UTYPE$, meaning that "$UTYPE$ is the universe of type names of the state machine sm" [13]. This holds true for all other universe names.

- *UOID*: The universe of object identifiers. The universe of object identifiers is the union of all carrier sets of all class names. $UOID = \bigcup_{C \in UCLASS} CAR(C)$. Note that $UOID \subseteq UVAL$.

- *INSTANCE*: While not technically referred to as a universe, *INSTANCE* is the set of 'actual' objects, or object structures. Note that $INSTANCE \subseteq UVAL$. Also, $INSTANCE = \bigcup_{C \in UCLASS} CAR(*C)$. $*C$ is the dereference of class name C, i.e., $*C$ is the record type defining the structure of the class C.

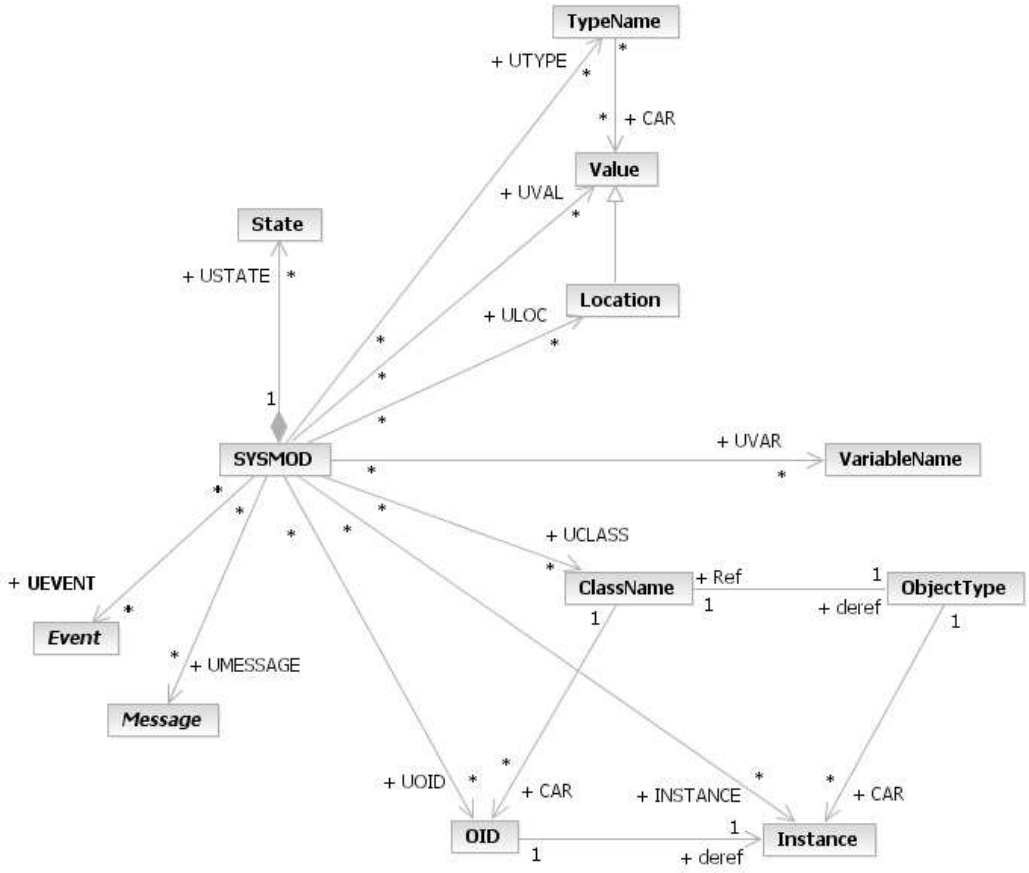


Figure 3.1: Class diagram showing the relationship between the System Model *SYSMOD* and various universes. Role names on associations support the notion that *UTYPE* is shorthand for $sm.UTYPE$, where $sm \in SYSMOD$. Multiplicities of $*$ support the notion that a type, value, variable, etc. could belong to more than one specific state machine *sm*

3.2.2 Types and Values

Types and values are organized into a type system as shown in Figure 3.2. *Values*, on the right, are elements of carrier sets of *TypeNames*, on the left. Much of the System Model structure comes down to either *TypeName* or *Value*.

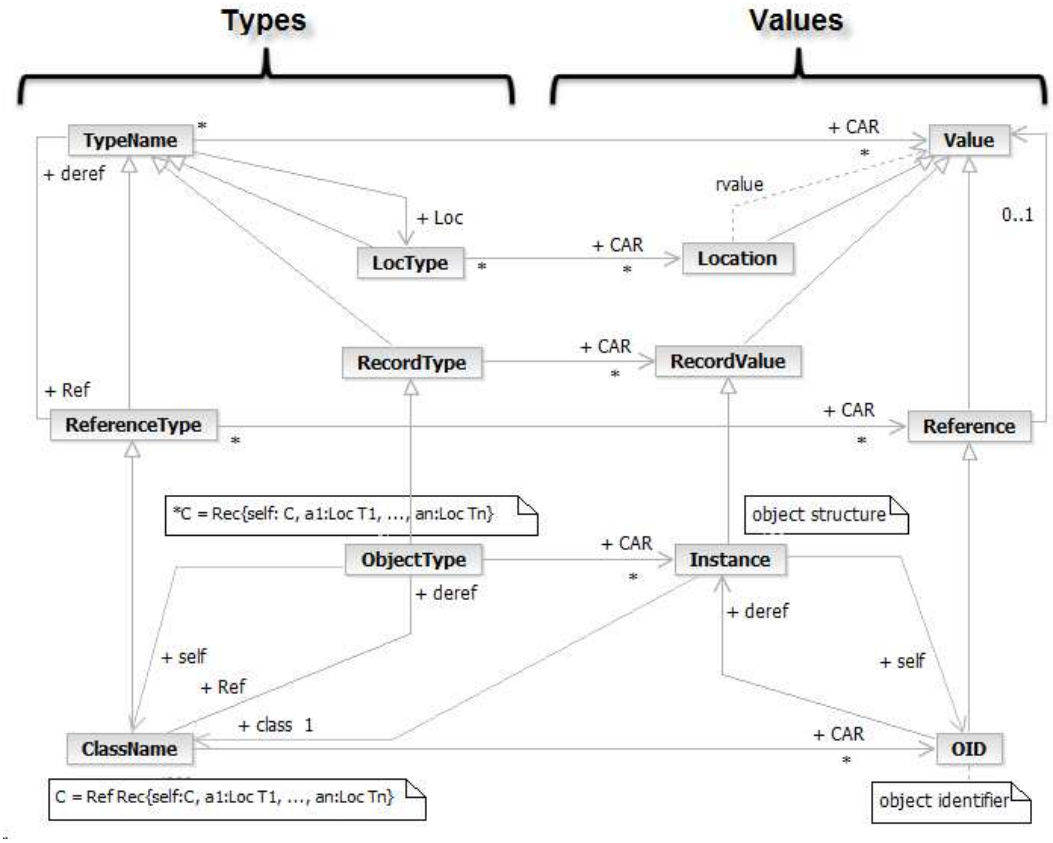


Figure 3.2: Class diagram representing System Model *TypeName* and *Value* concepts. Note the symmetry between the two sides

The USE diagram in Figure 3.3 shows a partial System Model universe, containing two instances *TypeName*, *Int* and *Boolean*, and several instances of *Value*. As can be seen from the diagram, the values are related to the type names through their carrier sets.

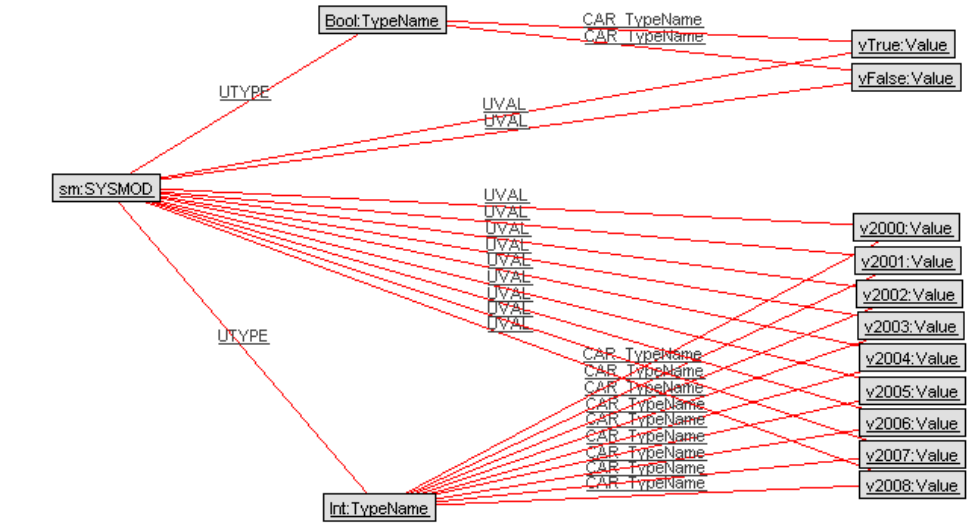


Figure 3.3: Partial System Model universe with *TypeName* and *Value*

3.2.2.1 References and Locations

Essentially, references are 'pointers' to values. As can be seen from Figure 3.2, there is a *ReferenceType* specializing *TypeName*, with a corresponding *Reference* specializing *Value*.

A *ReferenceType Ref T* is a "type name whose carrier set consists of an infinite set of references" [13]. As mathematically defined functions, reference types, such as *Ref T*, do not change over time. In other words, references in *Ref T* always refer to the same value.

The System Model is careful to maintain separation between static and dynamic elements. As references are, by definition, static, it was necessary to introduce an "explicit concept of locations for mutable values" [13]. Essentially, locations can hold values and correspond to the 'slots' referred to in UML's structural foundation [59].

A *LocType Loc T* is the "type of locations that store data of type *T*" [13]. The elements in a *LocType*'s carrier set are *Location* values. Because $ULOC \subseteq UVAL$, locations can be "passed around and stored like ordinary values" [13]. While a reference never changes, the value stored in a location can change. In fact, the current value of a location depends on the current state of the state machine.

The USE diagram in Figure 3.4 shows a partial System Model universe, this time showing the relation between a *LocType*, its *TypeName* and a *Location* value. The *LocType Loc_Int* is a pointer to values of type *Int*. Both *Int* and *Loc_Int* are part of the *UTYPE* universe. In this instance, there is one location value in the location type's carrier set, *locInt1*. By definition, a location is a value, hence *locInt1* being part of both the *UVAL* and *ULOC* universes.

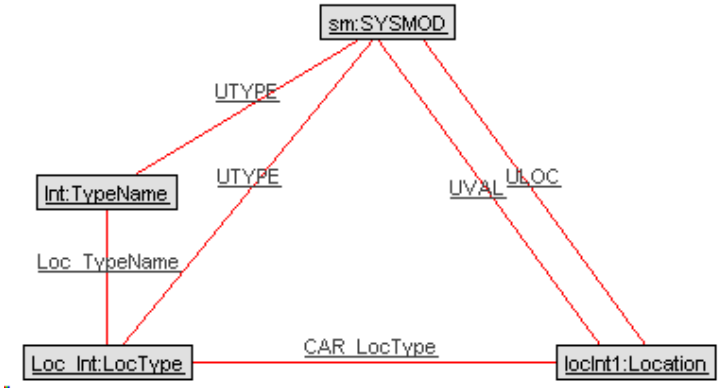


Figure 3.4: Partial System Model universe with *LocType* and *Location*

3.2.2.2 Records

Type names can be composed into record types. The elements of a record type's carrier set are record values. More precisely, a record type is defined as a set of

UVAR–UTYPE pairs, e.g., $Rec\{x : Int, b : Bool\}$. Note that the ordering of the pairs is irrelevant and thus $Rec\{x : Int, b : Bool\}$ is considered to be the same record type as $Rec\{b : Bool, x : Int\}$. A record value is defined as a set of *UVAR–UVAL* pairs, e.g., $[x : 5, b : true]$.

3.2.2.3 Classes and Objects

A *ClassName* is a type name, defined as a reference to a record type. The System Model documentation describes[2] a class name

$$C = Ref\ \ Rec\{a_1 : Loc\ \ T_1, ..., a_n : Loc\ \ T_n\}$$

In other words, the class name C is a reference to a record that consists of a set of variable name–location pairs. The variable names are the attributes of the class. The locations refer to values, which can change over the life of the model.

The actual record structure to which the class name C refers is only described as $*C$ (the dereference of C). We have defined the concept of *ObjectType*, which specializes *RecordType*, to describe the record type of a class. i.e., $*C$ is an instance of *ObjectType*. The relationships between *ClassName*, *ObjectType* and *ReferenceType*, *RecordType* can be seen in Figure 3.2 on page 39.

On the value side, an *OID* (object identifier) is defined as a reference to a record value. In other words, the carrier set of *ClassName* is made up of elements of *OID*, and the carrier set of *ObjectType* is made up of elements of *Instance*. Again, Figure 3.2 shows how *OID* and *Instance* relate to the rest of the elements already described.

The System Model documentation includes Figure 3.5, which shows the structure of an object, as defined by the System Model. We have annotated the drawing with

[2]The original documentation includes a *self* term, which was later deemed to be unnecessary.

additional information.

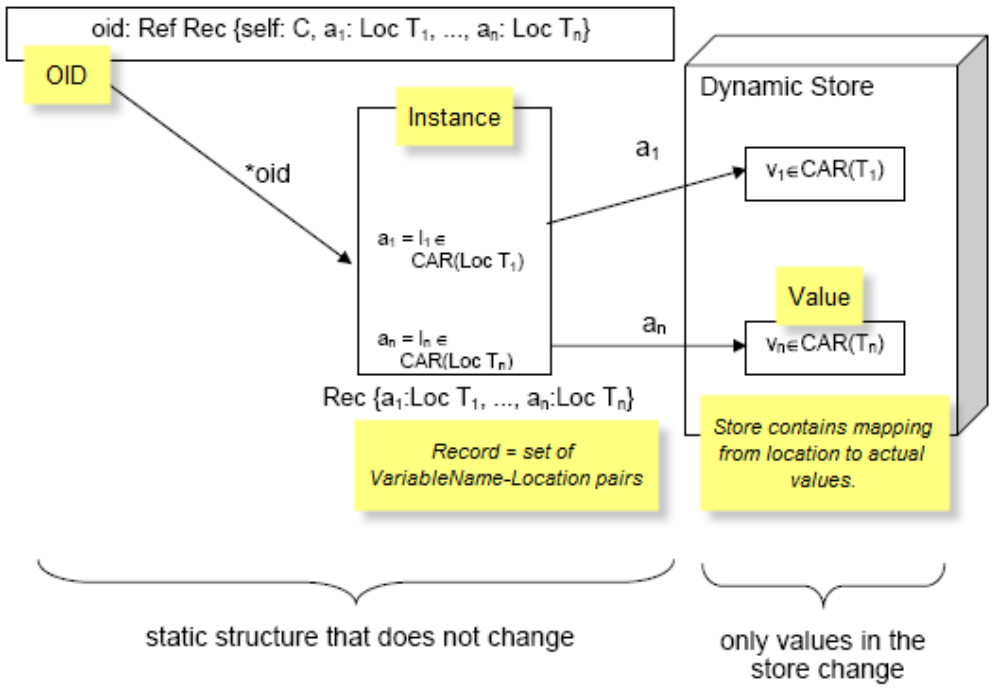


Figure 3.5: Structure of an object, as defined in the System Model documentation [13]. An object is a reference to a record. The record contains a set of *VariableName-Location* pairs. The locations refer to *Value*s, contained in the 'dynamic store'. In other words, the value of an object's attribute depends on the current state of *sm*

Figure 3.5 emphasizes the fact that the structure of the System Model universe is static. There is a two-stage dereferencing from an object identifier to its mutable attribute values. First, the object identifier (*oid*) is dereferenced to get the instance representing the object ($*oid$). Then, each attribute of the object (a_1 through a_n) is dereferenced to get the actual *Value* assigned to that attribute.

As an example of how a UML class and object would be represented by System Model concepts, consider the class diagram in Figure 3.6. It shows the *Car* classifier

and an instance of that classifier. The USE diagram in Figure 3.7 shows a partial System Model universe showing the structure of the *Car* classifier and instance. In general, elements above the dotted line represent the definition of *Car*; elements below represent the definition of *Car1*. The definition of *Car* includes the *ClassName* (reference to a record) and the *ObjectType* (record structure). In this case, the structure of the class is one *TypedVariable*, which is a pair composed of a *VariableName* (called *year*) and a *LocType* (for locations of type *Int*). In other words, the class has one attribute, of type *Int*, called *year*. On the object side, the new car object is represented by its identifier (*OID*), which dereferences to the actual object structure (*Instance*). That structure contains one *RecordElement* pair, made up of a *VariableName* (again, *year*), and a *Location* value.

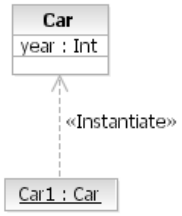


Figure 3.6: Example class diagram of *Car* class, with an instantiated *Car* object

3.2.2.4 Attribute and Projection Functions

The System Model documentation includes rough definitions of *attr* and *proj* functions for *RecordType* and *RecordValue*. The *attr* function is used to retrieve the set of attributes (variable names) used in the definition of a record type. The *proj* function

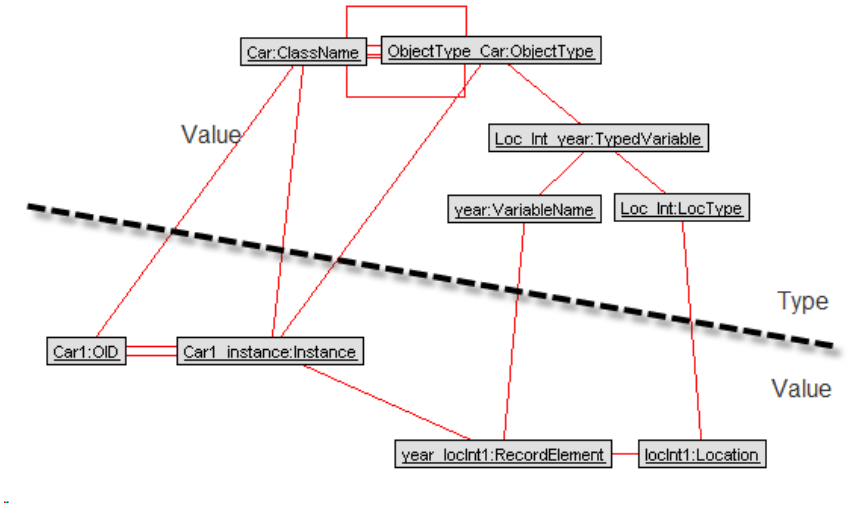


Figure 3.7: Partial System Model universe consistent with the class diagram in Figure 3.6

is used to project a variable name onto a record value and retrieve the value associated with that attribute.

Recall that *ClassName* specializes *ReferenceType*, and references *ObjectType*, which specializes *RecordType*. In other words, an *ObjectType* is simply a set of pairs, where the first element of the pair is a variable name, and the second element is a type name. The *ClassName* is simply a reference to this *ObjectType*.

We define a function $attr : UCLASS \rightarrow \mathcal{P}\, UVAR$ that returns all of the variable names from a class definition, such that

$$attr(class) = \{var \in UVAR \mid \exists t \in UTYPE \wedge (var, t) \in *class\}$$

where $*class$ is the dereference (i.e., object type) of the class name *class*.

Recall also that *OID* specializes *ReferenceValue* and references *Instance*, which specializes *RecordValue*. In other words, an *Instance* is simply a set of pairs, where the first element of the pair is a variable name, and the second element is a location.

The object identifier is simply a reference to this *Instance*.

We define a projection function $proj : UOID \times UVAR \nrightarrow ULOC$ that returns the projection of a variable name onto an object instance, such that

$$proj(oid, var) = \begin{cases} loc \in ULOC & \text{if } (var, loc) \in *oid \\ \text{undefined} & \text{otherwise} \end{cases}$$

where $*oid$ is the dereference (i.e., instance) of the object identifier oid. Note that by definition, variable names are unique in an object definition; therefore *proj* returns at most one location.

3.2.2.5 Inheritance

The System Model provides support for *subclassing*, also called *inheritance*. Subclassing is "not based on a structural definition" [13]; a subclass can have an arbitrarily different structure and two classes with the same attributes are not necessarily in a subclassing relationship. In addition, due to the relational point of view of the System Model, multiple inheritance is implicitly supported [13].

The System Model indicates that a class *C'* inherits from class C with the binary subclass relation

$$sub \subseteq UCLASS \times UCLASS$$

where $C' subC$ or $(C', C) \in sub$ indicates that class C' inherits from class C. This relation is transitive, so $(A, B) \in sub$ and $(B, C) \in sub$ implies that $(A, C) \in sub$. In addition, the relation is reflexive, which means that every class inherits from itself, i.e., $(C, C) \in sub$. Note that naming conflicts are avoided because the System Model requires that "all attribute definitions introduce different names" [59, §2.7].

3.2.2.6 Constraints

Finally, the System Model documentation defines several constraints on classes and objects:

- Object identifiers uniquely point to instances, i.e., every object identifier points to one instance, and every instance is referred to by one object identifier.
- Objects may not share locations.
- Every object knows its identifier and its class; therefore, each object belongs to exactly one class.

3.2.3 Variables

While type and value are two major concepts, from which almost everything that interests us descends, there is a third major concept—that of *VariableName*. The System Model documentation does not distinguish between attribute and variable; both terms are used interchangeably. For continuity purposes, we use the term *VariableName* to refer to the System Model concept. We reserve the term 'attribute' for the attributes of a class, and the term 'variable' for local variables and parameters.

3.3 State

As mentioned earlier, the system model *SYSMOD* is the universe of state machines. The behaviour of a UML model is viewed as a large state machine $sm \in SYSMOD$. The state machine is composed of a set of states and a state transition function, as shown in Figure 3.8.

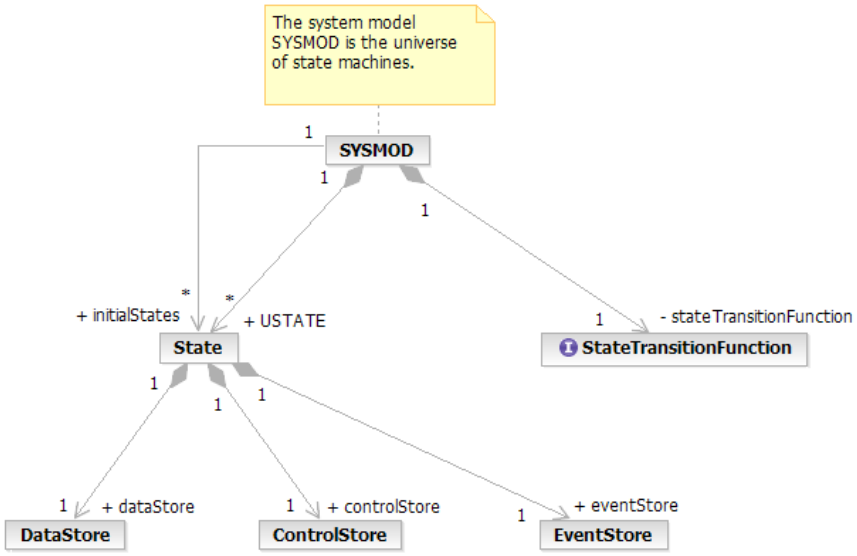


Figure 3.8: Class diagram showing that a state machine is composed of a set of states and a state transition function

The System Model contains a definition of a suitable state transition function, specifically, the "general notion of timed state transition systems" [15]. We, however, have chosen to treat the state transition function element of the System Model as an interface. Although the concept of 'state' is defined in quite some detail in the System Model, the notion of 'state transition system' is defined in a much more abstract manner. This, combined with the fact that our research is focused on UML actions, and activities are a natural fit for the composition of actions into behaviours, has led us to use UML activities as our state transition function. In other words, the UML activity determines how the next state of the state machine will be reached. *USTATE* is the universe of states. As can be seen from Figure 3.8, a state $state \in USTATE$ is defined as a triple $state = (ds, cs, es)$, where:

- $ds : DataStore$ is the data store, i.e., information about the current objects

existing in the UML model under examination.

- *cs* : *ControlStore* is the control store, i.e., information related to the transitioning from state to state.
- *es* : *EventStore* is the event store, i.e., information related to messages, message passing, event buffers, etc.

3.3.1 Data Store

The data store "models the state of a system at a certain point in time" [13]. At any point in time, the data store contains a finite set of object identifiers, i.e., a "snapshot of the data state" [13] of an executing model. Figure 3.9 shows a class diagram of the data store's structure.

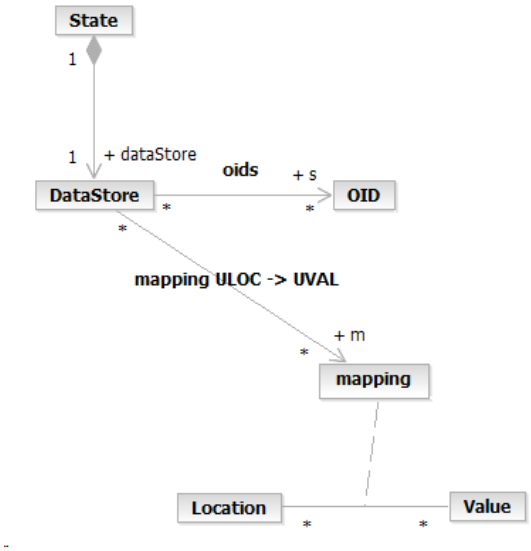


Figure 3.9: Class diagram showing the structure of the data store. The data store contains a set s of object identifiers, and a mapping m from *Location* to *Value*

3.3.1.1 Data Store Defined

The data store is a tuple

$$ds = (s, m)$$

where

- $s \subseteq UOID$ is the set of object identifiers that currently exist in the model's execution.
- $m \in ULOC \rightarrow UVAL$ is the set of mappings of those objects' attributes to values.

3.3.2 Control Store

The control store contains "information needed to determine the state transitions" [14]. As originally defined, the control store consists of:

- A stack of method/operation calls.
- A program counter, to indicate the "progress of the running program" [14]. *UPC* is the universe of program counters.
- Possibly information about threads. *UTHREAD* is the universe of threads. Each thread is associated with a stack, which may contain one or more stack frames. *UFRAME* is the universe of frames.

Figure 3.10 shows a class diagram of how the control store was originally intended to be structured. Note that it was necessary to modify the structure of the control store to accommodate activities. These modifications are discussed in Section 4.1.

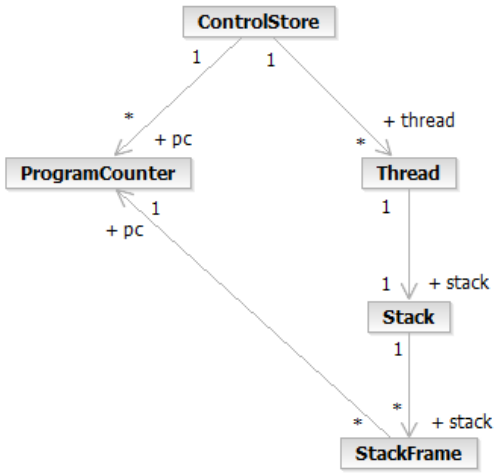


Figure 3.10: Class diagram showing the original structure of the control store

3.3.2.1 Methods and Operations

The System Model tends to focus on the object-oriented paradigm; objects are accessed through their methods/operations [14]. The System Model distinguishes between an object's methods, operations, and method names. Each operation has a name, a signature, and an implementation.

The relationship between these three concepts (and how they relate to the System Model itself) can be seen in the class diagram in Figure 3.11.

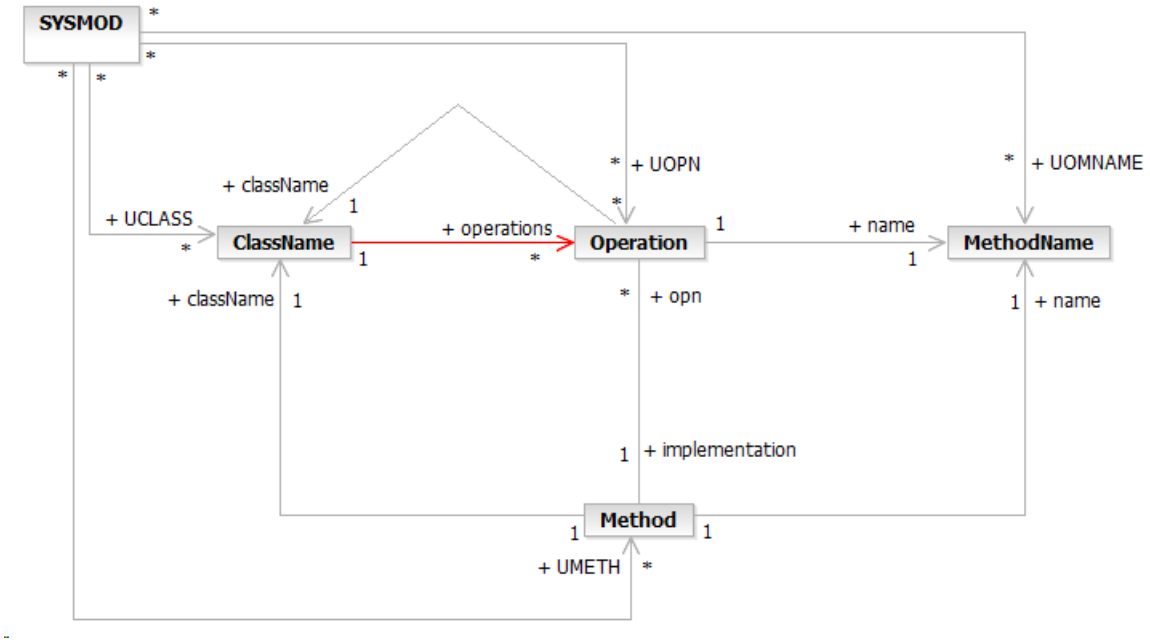


Figure 3.11: Class diagram showing the relationship between *SYSMOD*, *Operation*, *Method* and *MethodName*

Operations The term *Operation* refers only to the signature of an operation. An operation $op \in Operation$ is defined as a 5-tuple

$$op = (name, class, implementation, parameters, return)$$

where

- $name \in UOMNAME$ is the name of the operation (see below).
- $class \in UCLASS$ is the class in which this operation is defined.
- $implementation \in UMETH$ represents the implementation of this operation (see below).
- $parameters \in List(UTYPE)$ is a (possibly empty) list of parameter types.

- $return \in UTYPE$ is a return type.

$UOPN$ denotes the universe of operations.

Methods The term *Method* refers also to the implement (or body) of the operation. Methods have signatures, as well as implementations. A method $method \in Method$ is defined as a 4-tuple

$$method = (name, class, operation, behaviour)$$

where

- $name \in UOMNAME$ is the name of the operation (see below).
- $class \in UCLASS$ is the class in which this method is defined.
- $operation \in UOPN$ is the operation associated with this method (represents the method signature).
- $behaviour \in Behaviour$ is the behaviour representing the actual implementation of this method.

$UMETH$ denotes the universe of method implementations.

MethodName The term *MethodName* refers to simply the name of the operation/method. $UOMNAME$ denotes the universe of method names.

Behaviour We add the abstract concept of behaviour to the System Model universe, to represent that an operation's implementation may take various forms. For instance, an implementation may take the form of a state machine, an activity, a piece of actual program code, etc. Let $Behaviour$ be the set of all behaviours.

3.3.2.2 Control Store Defined

An important rationale in using mathematics to define the System Model was to avoid biases inherent in other formalisms. That said, the use of program counters, stacks and threads imposes its own bias. The underlying assumption is that the executing UML model behaves in accordance with an imperative programming paradigm, e.g., the notion of a program counter pointing to the next instruction, stack frames used to resume computation after a method call, etc.

Even with this slight imperative programming bias, the control store defined in the System Model is actually a very general concept, consisting only of program counters, threads, and stacks. The precise formalization of the control store depends heavily on the type of behaviour to be modelled. For instance, the program counter for a fragment of program code would be a single reference to a line of code. On the other hand, the program counter for a state machine could be the set of currently active states. The program counter for an activity is the set of tokens sitting on enabled nodes. We specialize the control store for use with activities in Chapter 4.

3.3.3 Event Store

The purpose of the event store is to buffer events until they can be processed. The System Model supports events and messages, in a slightly more simplified manner than that described by UML. There is no circular relationship between events and messages; they are considered separate entities. *UEVENT* is the universe of events. *UMESSAGE* is the universe of messages. Two specific types of events are currently described: receive events (receipt of a message) and send events (sending of a message). Likewise, two types of message are described: call messages (invoking a method

call) and return messages (returning from a method call). Other types of events and messages are permitted as required. In fact, we have added a third type of message to represent signals.

The class diagram in Figure 3.12 shows the relationship between the different types of events and messages.

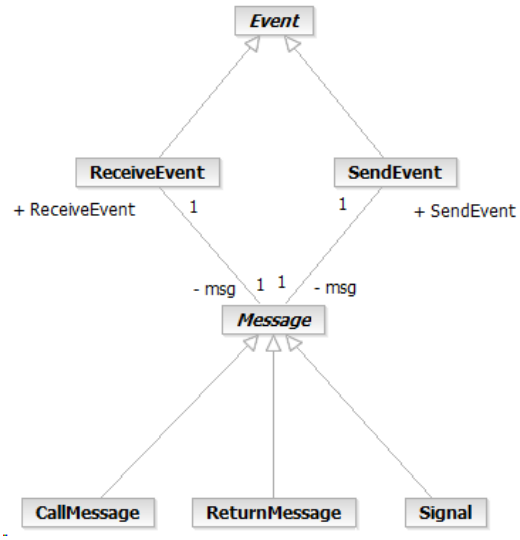


Figure 3.12: Class diagram showing the relationship between different types of *Events* and *Messages* in the System Model. We have added a new type of message to represent signals

3.3.3.1 Messages

UMESSAGE is the universe of messages. The System Model specifically describes two types of messages: call messages and return messages. The System Model permits other types of messages; we have added a third type of message to represent signals.

We define three sets of messages, *Calls*, *Returns* and *Signals* such that

$$Calls \cup Returns \cup Signals \subseteq UMESSAGE$$

and

$$Calls \cap Returns = \{\}$$

$$Calls \cap Signals = \{\}$$

$$Signals \cap Returns = \{\}$$

Calls Call messages are used to invoke method calls. A call message $call \in Calls$ is defined as a 5-tuple

$$call = (receiver, op, params, sender, thread)$$

where

- $receiver \in UOID$ is the called object.
- $op \in UOPN$ is the operation being called.
- $params \in List(UVAL)$ is the list of parameters being passed to the called method.
- $sender \in UOID$ is the sending object.
- $thread \in UTHREAD$ is the thread currently executing the call.

Returns On the other hand, return messages are used to return from method calls. A return message $return \in Returns$ is defined as an 5-tuple

$$return = (receiver, op, returner, thread, result)$$

where

- $receiver \in UOID$ is the object to which the execution is returning, i.e., the original calling object.

- $op \in UOPN$ is the operation that was originally called.
- $returner \in UOID$ is the object that was called upon to perform the operation.
- $thread \in UTHREAD$ is the thread that was executing when the original call was made.
- $result \in UVAL$ is the result of the called operation. For simplicity, we restrict the return to only one value; however, it is possible to return a record (collection).

Signals Finally, signals are used for asynchronous communication between objects. A signal $signal \in Signals$ is defined as an 4-tuple

$$signal = (name, receiver, sender, params)$$

where

- $name \in UNAME$ is the name (e.g., description) of this signal.
- $receiver \subseteq UOID$ are the targets of this signal; multiple targets are used for broadcasting.
- $sender \in UOID$ is the source of this signal.
- $params \in List(UVAL)$ is the list of parameters being passed to the target of this signal, i.e., contents of the signal.

and

- $UNAME$ is the universe of names. We have added this universe to the System Model to represent the names of items, such as signals, triggers, etc. For instance, the AcceptEventAction waits for a signal whose name/type/description

matches a particular trigger. We simply use the universe of names to represent these descriptors, as well as other names[3] used in the System Model, e.g.,

$$UCLASS \subseteq UTYPE \subseteq UNAME$$

$$UVAR \subseteq UNAME$$

$$UOMNAME \subseteq UNAME$$

There are several functions defined on messages:

- $ReceiveEvent : UMESSAGE \rightarrow UEVENT$ is a function that maps a message to its associated receive event.
- $SendEvent : UMESSAGE \rightarrow UEVENT$ is a function that maps a message to its associated send event.

3.3.3.2 Events

$UEVENT$ is the universe of events. The System Model specifically describes two types of events: send and receive events. Other types of events are permitted, but not necessary for our purposes. Note that the transmission of a message includes a send event (for the sender), followed by a receive event (for the receiver). Events may be handled, ignored, or stored until a later time. In addition, events need not be handled in the order they appear.

We define two sets of events, *Sends* and *Receives* such that

$$Sends \cup Receives \subseteq UEVENT$$

[3]Our $UNAME$ is based on the UML concept of NamedElement, i.e., elements that have names.

and

$$Sends \cap Receives = \{\}$$

In addition, $msgOf : UEVENT \rightarrow UMESSAGE$ is a function that maps an event to its associated message, such that

$$msgOf(ReceiveEvent(m)) = m$$

and

$$msgOf(SendEvent(m)) = m$$

where *ReceiveEvent* and *SendEvent* are System Model functions defined on messages (see page 58).

3.3.3.3 Event Buffer

The System Model assumes a rather general structure for an event buffer; at the most basic, it would simply store events in a First-In-First-Out (FIFO) queue without any priority mechanisms [14]. $Buffer(UEVENT)$ is some data structure that receives and returns events. We define the following buffer functions. Note that we make no assumptions about how an event is added to or retrieved from a buffer, i.e., we do not assume a FIFO queue or any other data structure.

- $addEvent : Buffer(UEVENT) \times UEVENT \rightarrow Buffer(UEVENT)$ is a function that adds an event to an existing buffer.

- $removeEvent : Buffer(UEVENT) \times UEVENT \rightarrow Buffer(UEVENT)$ is a function to remove a specific event from an existing buffer.

- $inBuffer : Buffer(UEVENT) \times UEVENT \rightarrow Boolean$ is a function that indicates whether or not a given event is in a buffer, such that

$$inBuffer(buffer, e) = \begin{cases} true & \text{if event } e \text{ is in } buffer \\ false & \text{otherwise} \end{cases}$$

3.3.3.4 Event Store Defined

We define the event store as simply a function that maps an object identifier to its event buffer, i.e.,

$$es : UOID \rightarrow Buffer(UEVENT)$$

where

- $Buffer(UEVENT)$ is a data structure that receives and returns events, as defined on the previous page.

The USE diagram in Figure 3.13 shows a partial System Model universe, showing how a state relates to its three stores. In this case, a car object (named *OIDfor_Car_1*) has been instantiated and is shown as part of the data store. That object's event buffer is shown as part of the event store. In addition, there are two local variables showing as part of the control store.

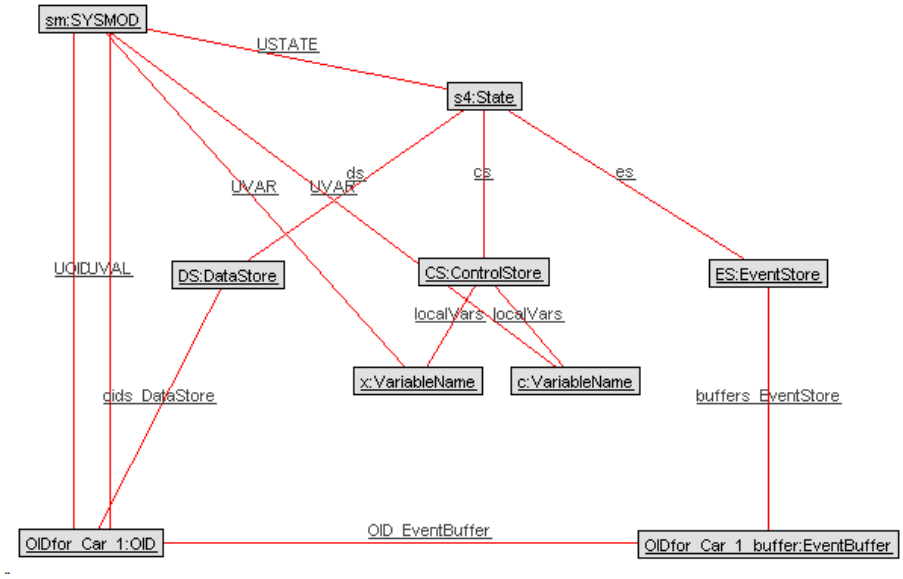


Figure 3.13: Partial System Model universe with state and stores

Chapter 4

Structure of Activities

As discussed in Section 2.2, the System Model represents the structural foundation of UML, i.e., the lowest layer in the three-layer semantics architecture. Actions, which act like executable programming language statements, are then explained in terms of this semantic domain. However, the System Model has been deliberately described in a very general fashion. In fact, as discussed in Chapter 3, it is necessary to specialize the control store, i.e., to tailor it to the type(s) of behaviour being modelled. We are using UML activities to compose individual actions into larger executions. In order to discuss the semantics of actions, we first must discuss the structure of activities, and tailor the System Model's control store appropriately.

In this chapter, we first define a control store suitable for handling the execution of activities. Then, we formally define the structure of activities and discuss the concept of node enablement. Further discussion, including pseudocode, about the semantics of activity execution is presented in Chapter 6.

4.1 Control Store for Activities

4.1.1 Program Counters

When executing a typical imperative program, the control store's program counter would hold the address of the next instruction to be executed. However, activities are by nature concurrent executions. We use the System Model concept of *Program-Counter* as an interface and introduce the concept of $Token \subseteq UPC$ to represent control tokens in the executing set of activities. Thus, *pc* in the control store is now the *set* of tokens sitting on nodes.

A token $t \in Token$ is defined as a 4-tuple

$$t = (id, thread, sittingOn, cameFrom)$$

where

- $id \in Identifier$ is a unique identifier.
- $thread \in SchedulerThread$ is the runtime thread associated with this token.
- $sittingOn \in GraphNode$ is the activity node upon which this token rests.
- $cameFrom \in GraphNode$ is the activity node from which this token was sent.

and *GraphNode* refers to a node in the graph representing the activity.

4.1.2 Threads

The System Model provides support for a centralized, multiple-threaded view of computation. We use the System Model's *Thread* as an interface, and have created the concept of $SchedulerThread \subseteq UTHREAD$ to (loosely) realize that interface. We say

'loosely' because the System Model concept of thread contains additional restrictions, such as the constraint that a thread is only active if its stack is not empty [14]. However, it turns out that many of our threads will never have anything in their stacks, and yet are still considered active. See our discussion on invocation actions in Section 5.4.2.

A scheduler thread $th \in SchedulerThread$ is defined as a pair

$$th = (id, token)$$

where

- $id \in Identifier$ is a unique identifier.
- $token \in Token$ is the token associated with this thread.

Our interpretation uniquely pairs each token (program counter) with a thread. When a new *Token* is created, a new *SchedulerThread* is created; when a token is deleted, so is its associated thread:

$$(\forall t \in Token \mid \exists! th \in SchedulerThread \mid t.thread = th \wedge th.token = t)$$

$$\wedge(\forall th \in SchedulerThread \mid \exists! t \in Token \mid th.token = t \wedge t.thread = th)$$

Figure 4.1 shows how our token and scheduler thread realize the System Model interfaces.

4.1.3 Variables

The System Model supports local variables, e.g., local to a method implementation. They are stored only in stack frames, a practice which is logical, considering an imperative programming paradigm. However, as mentioned earlier, this practice does

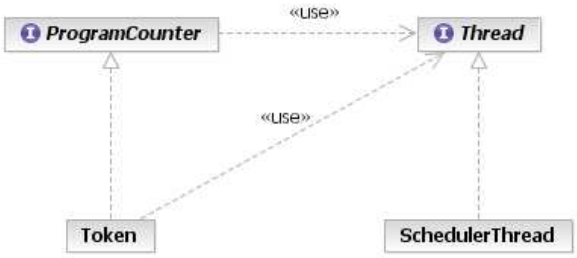


Figure 4.1: Our concept of *Token* and *SchedulerThread* realize the System Model's *ProgramCounter* and *Thread* interfaces, respectively

not suit activities. Specifically, stack frames are only used with a certain kind of invocation action.[1] Therefore, we require some manner in which to store local variables; these would be variables local to a particular activity. We have added local variable storage, similar to the data store, to the control store.

Specifically, we introduce the tuple

$$vars = (varloc, locval)$$

where

- $varloc \in UVAR \rightarrow ULOC$ is a function recording the location associated with each local variable.
- $locval \in ULOC \nrightarrow UVAL$ is a function recording the current value associated with a variable's location, such that

$$locval(loc) = \begin{cases} val \in UVAL & \text{if the location } loc \text{ has value } val \\ \text{undefined} & \text{otherwise} \end{cases}$$

[1] In essence, only one action, the CallOperationAction, uses the stack and therefore stack frames. No other action makes use of the stack. See Section 5.4.2.

The use of an intermediate location between the variable and the attribute intentionally mimics the two-stage dereferencing of object attributes discussed in Chapter 3.

Although the *vars* tuple fulfills a similar function as the (s, m) tuple in the data store, we have chosen to keep it in the control store for two reasons:

- The System Model specifically maintains local variables in the control store.
- We wanted to maintain the separation between static and dynamic data. Local variables are, by definition, dynamic, and relate to the current activities being executed.

Finally, it should be noted that, like the mechanics of the data store, the mechanics of using two mappings to store local variables is completely hidden from the user.

4.1.4 Current Activities

It is possible for an executing activity to call upon other activities. As each new activity is called, it must be added to the 'working set' of current activities. We have added a new term *ads* to the control store to represent the currently executing activities.

4.1.5 Control Store Defined (for Activities)

We define the control store as a 4-tuple

$$cs = (ads, pc, thr, vars)$$

where:

- $ads \subseteq ActivityDiagram$ is the set of activities currently being executed (see Section 4.2).
- $pc \subseteq UPC$ is the set of program counters (i.e., tokens) currently available during the model's execution.
- $thr \subseteq UTHREAD$ is the set of threads (i.e., scheduler threads) currently available during the model's execution.
- $vars \in (UVAR \rightarrow ULOC) \times (ULOC \nrightarrow UVAL)$ is data related to local variables associated with the currently executing activities.

4.2 Defining Activities

We use a subset of UML activities to compose individual actions into larger executions. Specifically, we support control and data flow, tokens, and all control nodes. We do not support: partitions, interruptible regions, exceptions, weights on edges, multiplicities on pins other than 1..*, object nodes that are not pins, or loop and conditional structures.

Activities are directed graphs. Each graph node represents some element of the activity, e.g., an action node represents a specific UML action or a control node represents a fork, etc.

Activities are a kind of behaviour, i.e., we use activities to represent behaviour in the System Model. Let $ActivityDiagram \subseteq Behaviour$ denote the set of activity diagrams.

An activity diagram $ad \in ActivityDiagram$ is represented as a 3-tuple

$$ad = (id, nodes, localVars)$$

where

- $id \in Identifier$ is the unique identifier (e.g., file name) of the activity.
- $nodes \subseteq GraphNode$ is the set of all nodes in the activity.
- $localVars \subseteq UVAR$ is the set of variable names local to this activity.

We define a function $initialNodes : ActivityDiagram \to \mathcal{P}\, GraphNode$ to retrieve the set of initial nodes of the activity, such that

$$initialNodes(ad) = \{n \in ad.nodes \mid n.in = \{\}\}$$

Note that the set of initial nodes consists of initial control nodes (e.g., as in Figure 2.10 on page 23), as well as any other node without incoming edges.

4.2.1 Graph Nodes

There are three types of node in the graph representing an activity:

- **Action nodes** represent actions in an activity. They could be high-level concepts, such as "Pay Bills", or low-level primitives, such as CreateObjectAction.
- **Object nodes** represent data and/or pins in an activity. We use object nodes as explicit pins.
- **Control nodes** represent the control mechanisms in an activity; they coordinate the flows (both data and control) between other nodes [59, §12.3.20].

Figure 4.2 shows the hierarchy of control nodes in UML. We support all control nodes.

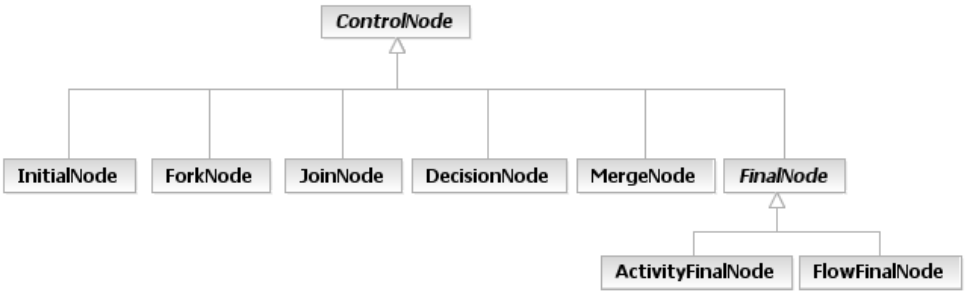


Figure 4.2: Control nodes from UML metamodel

Every node in an activity diagram graph is either an action node, an object node, or a control node, i.e.,

$$GraphNode = ActionNode \cup ObjectNode \cup ControlNode$$

where all three sets are pairwise disjoint:

$$ActionNode \cap ObjectNode = \{\}$$

$$ObjectNode \cap ControlNode = \{\}$$

$$ControlNode \cap ActionNode = \{\}$$

4.2.1.1 Action Nodes

We define an action node $an \in ActionNode$ as an 8-tuple

$$an = (id, in, out, action, annot, waiting, owner, stalled)$$

where

- $id \in Identifier$ is the unique identifier of the node.
- $in \subseteq Identifier$ is the set of identifiers representing source nodes of this node.
- $out \subseteq Identifier$ is the set of identifiers representing target nodes of this node.
- $action \in Action$ is the action associated with this node. See Section 5.1 on page 80.
- $annot \subseteq Annotation$ is the set of annotations associated with the action. Annotations are used to provide information about an action's attributes and associations.
- $waiting \in Boolean$ is a flag indicating whether or not this action is waiting for a result from a synchronously invoked behaviour.
- $owner \in UOID$ is the object that owns the activity containing this node.
- $stalled \in Boolean$ is a flag indicating whether or not this node is stalled (due to token offer semantics).

We define an annotation $annot \in Annotation$ as a pair

$$annot = (attr, value)$$

where

- $attr \in String$ is the name of the attribute or association described by the annotation.
- $value \in String$ is the name of the value to be assigned to the attribute.

For example, the annotations for the ReadIsClassifiedObjectAction shown in Figure 2.8 on page 21 could be {("isDirect", "true"), ("classifier", "Car")}.

4.2.1.2 Control Nodes

We define a control node $cn \in ControlNode$ as a 5-tuple

$$cn = (id, in, out, cnType, stalled)$$

where

- $id \in Identifier$ is the unique identifier of the node.
- $in \subseteq Identifier$ is the set of identifiers representing source nodes of this node.
- $out \subseteq Identifier$ is the set of identifiers representing target nodes of this node.
- $cnType \in \{Initial, Fork, Join, Decision, Merge, ActivityFinal, FlowFinal\}$ is the type of control node.
- $stalled \in Boolean$ is a flag indicating whether or not this node is stalled (due to token offer semantics).

Constraints There are several constraints on control nodes:

- An initial node has no incoming edges [59, §12.3.31]

$$\forall cn \in ControlNode \mid cn.cnType = Initial \Rightarrow \left|cn.in\right| = 0$$

- A final node has no outgoing edges [59, §12.3.28]

$$\forall cn \in ControlNode \mid cn.cnType = ActivityFinal \Rightarrow \left|cn.out\right| = 0$$

$$\forall cn \in ControlNode \mid cn.cnType = FlowFinal \Rightarrow \left|cn.out\right| = 0$$

- A fork node has one incoming edge [59, §12.3.30]

$$\forall cn \in ControlNode \mid cn.cnType = Fork \Rightarrow \left|cn.in\right| = 1$$

- The edges coming into and out of a fork node must be either all object flows or all control flows [59, §12.3.30]

$$\begin{aligned}&\forall cn \in ControlStore \mid cn.cnType = Fork\\ &\Rightarrow \big(isAllControl(getTrueSources(cn)) \Rightarrow isAllControl(getTrueTargets(cn))\\ &\wedge isAllData(getTrueSources(cn)) \Rightarrow isAllData(getTrueTargets(cn))\big)\end{aligned}$$

  and

  - $isAllControl : \mathcal{P}\,GraphNode \rightarrow Boolean$ is a function to check that a set of nodes can carry control flow, such that

$$isAllControl(nodes) = \forall n \in nodes \mid n \notin ObjectNode$$

  - $isAllData : \mathcal{P}\,GraphNode \rightarrow Boolean$ is a function to check that a set of nodes can carry data flow, such that

$$isAllData(nodes) = \forall n \in nodes \mid n \notin ActionNode$$

  - $getTrueSources : GraphNode \rightarrow \mathcal{P}\,GraphNode$ is a function to find the 'true' sources of a graph node (i.e., navigating through control nodes), such that

$$getTrueSources(gn) = \bigcup_{n \in gn.in} getTrueS(getNode(n))$$

  - $getTrueS : GraphNode \rightarrow \mathcal{P}\,GraphNode$ is a helper function to find the 'true' sources of a graph node, such that

$$getTrueS(gn) = \begin{cases} \{gn\} & \text{if } gn \in ActionNode \\ \{gn\} & \text{if } gn \in ObjectNode \\ \{\} & \text{if } gn \in ControlNode \\ & \wedge gn.cnType = Initial \\ getTrueSources(gn) & \text{otherwise} \end{cases}$$

– $getNode : Identifier \nrightarrow GraphNode$ is a function to retrieve a graph node based on an identifier, such that

$$getNode(n) = \begin{cases} gn \in GraphNode & \text{if } gn.id = n \\ \text{undefined} & \text{otherwise} \end{cases}$$

– $getTrueTargets : GraphNode \rightarrow \mathcal{P}\, GraphNode$ is a function to find the 'true' targets of a graph node (i.e., navigating through control nodes), such that

$$getTrueTargets(gn) = \bigcup_{n \in gn.out} getTrueT(getNode(n))$$

– $getTrueT : GraphNode \rightarrow \mathcal{P}\, GraphNode$ is a helper function to find the 'true' targets of a graph node, such that

$$getTrueT(gn) = \begin{cases} \{gn\} & \text{if } gn \in ActionNode \\ \{gn\} & \text{if } gn \in ObjectNode \\ \{\} & \text{if } gn \in ControlNode \\ & \wedge (gn.cnType = ActivityFinal \\ & \quad \vee gn.cnType = FlowFinal) \\ getTrueTargets(gn) & \text{otherwise} \end{cases}$$

- A join node has one outgoing edge [59, §12.3.34]

$$\forall cn \in ControlNode \mid cn.cnType = Join \Rightarrow \left|cn.out\right| = 1$$

- If a join node has an incoming object flow, it must have an outgoing object flow, otherwise, it must have an outgoing control flow [59, §12.3.34]

$$\begin{aligned} &\forall cn \in ControlStore \mid cn.cnType = Join \\ &\Rightarrow \big(\neg isAllControl(getTrueSources(cn)) \Rightarrow isAllData(getTrueTargets(cn)) \\ &\wedge isAllControl(getTrueSources(cn)) \Rightarrow isAllControl(getTrueTargets(cn))\big) \end{aligned}$$

- A decision node has one incoming edge [59, §12.3.22]

$$\forall cn \in ControlNode \mid cn.cnType = Decision \Rightarrow \left|cn.in\right| = 1$$

- The edges coming into and out of a decision node must be either all object flows or all control flows [59, §12.3.22]

$$\begin{aligned}&\forall cn \in ControlStore \mid cn.cnType = Decision\\ &\Rightarrow \big(isAllControl(getTrueSources(cn)) \Rightarrow isAllControl(getTrueTargets(cn))\\ &\wedge isAllData(getTrueSources(cn)) \Rightarrow isAllData(getTrueTargets(cn))\big)\end{aligned}$$

- A merge node has one outgoing edge [59, §12.3.36]

$$\forall cn \in ControlNode \mid cn.cnType = Merge \Rightarrow \left|cn.out\right| = 1$$

- The edges coming into and out of a merge node must be either all object flows or all control flows [59, §12.3.36]

$$\begin{aligned}&\forall cn \in ControlStore \mid cn.cnType = Merge\\ &\Rightarrow \big(isAllControl(getTrueSources(cn)) \Rightarrow isAllControl(getTrueTargets(cn))\\ &\wedge isAllData(getTrueSources(cn)) \Rightarrow isAllData(getTrueTargets(cn))\big)\end{aligned}$$

4.2.1.3 Object Nodes

We define an object node $on \in ObjectNode$ as a 6-tuple

$$on = (id, in, out, argName, value, stalled)$$

where

- $id \in Identifier$ is the unique identifier of the node.
- $in \subseteq Identifier$ is the set of identifiers representing source nodes of this node.
- $out \subseteq Identifier$ is the set of identifiers representing target nodes of this node.
- $argName \in ObjectNode \nrightarrow String$ is a function that retrieves the name of the argument, if any, that the object node represents. This name is used when passing arguments to invoked behaviour and is defined such that, for all $on \in ObjectNode$,

$$argName(on) = \begin{cases} \text{“args”} & \text{if this node represents all arguments} \\ \text{“arg\_i”} & \text{if this node represents the } i^{th} \text{ argument} \\ \text{“returnInformation”} & \text{if this node represents the return information to be passed from an accept action to a reply action} \\ \text{undefined} & \text{if this node does not represent an argument} \end{cases}$$

- $value \in UVAL$ is the value currently held in the object node.
- $stalled \in Boolean$ is a flag indicating whether or not this node is stalled (due to token offer semantics).

4.3 Defining Enablement

A pre-condition for any action node to execute is that it must be ‘enabled’, i.e., ready to fire. Whether or not a node is enabled is not so much about the node itself, but rather how that node fits into the current activity.

A node in an activity is considered ‘enabled’ if the following conditions hold:

- The node is not 'stalled'. A node is stalled if it has finished executing, but has not yet been able to pass its tokens to its targets.
- If the node is waiting, i.e., it needs a call or an event before being able to proceed, then that call/event is available in its owner's event buffer.
- Otherwise, the node has sufficient tokens, on the right incoming edges, i.e., one token coming in on each incoming edge.

Aside: Tokens Recall that $Token \subseteq UPC$. For any state $s \in USTATE$, the set of current program counters is $s.cs.pc$. For convenience sake, we define $Token_s$ as the set of tokens existing in state s and $Token = \bigcup_{s \in USTATE} Token_s$. In general, $Token_s \subseteq s.cs.pc$; however, when dealing exclusively with activities, tokens are the *only* program counters and thus $Token_s = s.cs.pc$. Similarly, $SchedulerThread_s \subseteq s.cs.thr$ in general and $SchedulerThread_s = s.cs.thr$ when dealing exclusively with activities.

Enablement Formally, we define the function

$$enabled : GraphNode \times State \rightarrow Boolean$$

where

$$enabled(gn, s) = \begin{cases} false & \text{if } gn.stalled \\ enabled_{on}(gn, s) & \text{if } \neg gn.stalled \wedge gn \in ObjectNode \\ enabled_{cn}(gn, s) & \text{if } \neg gn.stalled \wedge gn \in ControlNode \\ enabled_{an}(gn, s) & \text{if } \neg gn.stalled \wedge gn \in ActionNode \end{cases}$$

and

- $enabled_{on} : ObjectNode \times State \rightarrow Boolean$ is a function that determines whether or not an object node is enabled, such that

$$enabled_{on}(on, s) = sourceTokenMatch(on, s)$$

An object node is enabled if it has sufficient tokens on its incoming edges.

- $enabled_{cn} : ControlNode \times State \rightarrow Boolean$ is a function that determines whether or not a control node is enabled, such that

$$enabled_{cn}(cn, s) = \begin{cases} true & \text{if } cn.cnType = Initial \\ & \wedge \exists t \in Token_s \mid t.sittingOn = cn \\ true & \text{if } \big(cn.cnType = FlowFinal \\ & \vee cn.cnType = ActivityFinal\big) \\ & \wedge \exists t \in Token_s \mid t.sittingOn = cn \\ false & \text{otherwise} \end{cases}$$

A control node is enabled if it is an initial node, activity final node or flow final node, and it there is at least one token, in the current state, sitting on it.

- $enabled_{an} : ActionNode \times State \rightarrow Boolean$ is a function that determines whether or not an action node is enabled, such that

$$enabled_{an}(an, s) = \begin{cases} true & \text{if } an.waiting \wedge \Big|an.owner.buffer\Big| > 0 \\ & \wedge an.action.name = \text{“CallOperationAction”} \\ & \wedge returnMessageWaiting(an, s) \\ true & \text{if } !an.waiting \\ & \wedge \big(an.action.name = \text{“AcceptCallAction”} \\ & \vee an.action.name = \text{“AcceptEventAction”}\big) \\ & \wedge triggerEventWaiting(an, s) \\ & \wedge sourceTokenMatch(an, s) \\ true & \text{if } !an.waiting \\ & \wedge an.action.name \neq \text{“CallOperationAction”} \\ & \wedge an.action.name \neq \text{“AcceptCallAction”} \\ & \wedge an.action.name \neq \text{“AcceptEventAction”} \\ & \wedge sourceTokenMatch(an, s) \\ false & \text{otherwise} \end{cases}$$

An action node is enabled if it represents a CallOperationAction *waiting for a return message and there is such a message waiting, or it represents an accept action and there is a trigger event waiting, or it represents some other type of action and it has sufficient tokens on its incoming edges.*

- *sourceTokenMatch* : *GraphNode* × *State* → *Boolean* is a function that determines whether or not there are sufficient tokens on the node's incoming edges, such that

$$sourceTokenMatch(gn, s) = \begin{cases} true & \text{if } \forall n \in gn.in \mid \exists t \in Token_s \\ & \mid t.sittingOn = getNode(gn) \\ & \wedge t.cameFrom = n \\ false & \text{otherwise} \end{cases}$$

  The sources and tokens on a graph node match if there is at least one token currently on this node that came in from every incoming edge.

- *returnMessageWaiting* : *ActionNode* × *State* → *Boolean* is a function that determines whether or not there is an appropriate return message waiting for this action, such that

$$\begin{aligned} &returnMessageWaiting(an, (ds, cs, es)) = \\ &\Big|\{e \in es(an.action.owner) \mid \\ &e \in Receives \wedge msgOf(e) \in Returns \\ &\wedge msgOf(e).op = an.action.attrs(\text{“Operation”}) \\ &\wedge msgOf(e).thread \in threads(an)\}\Big| > 0 \end{aligned}$$

  If, in the action node's owner's event buffer, there is at least one receive event with a return message matching the operation called by this action node and matching a thread on this action node, then an appropriate return message is waiting. Specifying the exact thread is important because it is possible for one object to make repeated calls to a particular operation in a target. In that case, there may be several return messages in the caller's buffer, all with the same operation name. By specifying the thread, we ensure that the right return message is received (each call and subsequent return is associated with a specific thread from the calling object).

- *getNode* : *Identifier* → *GraphNode* is a function to retrieve a graph based on an identifier (see page 73).

- *triggerEventWaiting* : *ActionNode* × *State* → *Boolean* is a function that determines whether or not there is an appropriate event waiting for this action, such

that

$$triggerEventWaiting(an, (ds, cs, es)) = \Big|\{e \in es(an.action.owner) \mid e \in Receives \wedge msgOf(e) \in Calls \wedge msgOf(e).op.name = an.action.inPins(\text{“trigger”})\}\Big| > 0$$

If, in the action node's owner's event buffer, there is at least one receive event with a call message matching the action's trigger, then an appropriate trigger event is waiting.

- $action \in Action$ is the action associated with this node. See Section 5.1 on the next page for a discussion of *owner* and *attrs*.
- $msgOf : UEVENT \rightarrow UMESSAGE$ is a function that maps an event to its associated message (see page 59).
- $threads : GraphNode \rightarrow \mathcal{P}\,UTHREAD$ is a function that returns the threads associated with a particular node in the activity (see page 134).

Chapter 5

Actions

In this chapter, we formally define the concept of action. We use several activity examples to introduce individual actions and then formally map these actions to the System Model, thereby defining each action's behaviour. Finally, we provide a summary of how actions affect the underlying System Mode state.

5.1 Defining Actions

We define an action $a \in \mathit{Action}$ as a 4-tuple

$$a = (\mathit{name}, \mathit{inPins}, \mathit{outPins}, \mathit{attrs})$$

where

- $\mathit{name} \in \mathit{String}$ is the name of the action, e.g., "CreateObjectAction".
- $\mathit{inPins} \in \mathit{String} \nrightarrow \mathit{UVAL}$ is a function recording the current values on the input

pins, such that

$$inPins(s) = \begin{cases} v \in UVAL & \text{if input pin with name } s \text{ contains value } v \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $outPins \in String \nrightarrow UVAL$ is a function recording the current values on the output pins, such that

$$outPins(s) = \begin{cases} v \in UVAL & \text{if output pin with name } s \text{ contains value } v \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $attrs \in String \nrightarrow (UVAL \cup UVAR \cup UTYPE)$ is a function recording the values, variables or types of the action's attributes and associations,[1] such that

$$attrs(s) = \begin{cases} v \in UVAL & \text{if attribute/association } s \text{ contains value } v \\ v \in UVAR & \text{if attribute/association } s \text{ contains variable name } v \\ v \in UTYPE & \text{if attribute/association } s \text{ contains type name } v \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that the *name* and *attrs* of an action are static and will not change as the activity is executed. On the other hand, *inPins* and *outPins* are dynamic, and their values may change as the activity is executed.

Information about an action's attributes, associations, and pins can be found in the UML metamodel. For example, Figure 2.8 on page 21 shows the metamodel of the ReadIsClassifiedObjectAction. If we were to use this action in an activity, it would be defined as:

$$a = \big(\text{“ReadIsClassifiedObjectAction”}, \{(\text{“object”}, w)\},$$
$$\{(\text{“result”}, x)\}, \{(\text{“classifier”}, y), (\text{“isDirect”}, z)\}\big)$$

[1] Attribute and association names are unique; the same name will never be used for both an attribute and an association.

where the strings "object", "result", "classifier" refer to the names of the various pins, attributes, and associations, $w, x, z \in UVAL$ and $y \in UTYPE$.

5.2 Semantics of Actions

Using the System Model as our basic semantic domain, we formally define the semantics for the behaviour of individual UML actions by clearly identifying how the underlying state $s = (ds, cs, es)$ changes as an action node representing each particular action executes.

These mappings of action behaviour to the System Model will be presented in the following manner:

- We present an example activity containing a few actions. The purpose of these examples is to demonstrate how actions are used; we do not provide an example-driven definition of the actions.
- We discuss in plain language the purpose of each action.
- We provide the formal description of those actions included in the diagram.

The details of how to use our interpreter to execute these activities are discussed in Chapter 7.

5.2.1 Scope

There are 45 UML actions (including the abstract Action class), 36 of which are concrete. We provide formal definitions for the behaviour of 24 of these concrete actions, covering 30 actions in total (including abstract actions). In addition to some

miscellaneous actions, such as TestIdentityAction and ReadIsClassifiedObjectAction, we define all structural feature actions and all variable actions. In addition, we formally define all invocation actions, which are by far the most complicated actions.

In UML, links are instances of associations [59, §7.3.3]. Although the System Model supports associations in general, it clearly does not permit the instantiation of associations, i.e., "associations are somehow contained within the [data] store...they are somehow part of objects and locations and association relations do not extend the store" [13, page 22]. Due to this limitation, we decided early in our research that we would use attributes to indicate associations between objects, thereby negating the requirement to formally define any of the nine link or association actions.

The purpose of the ReclassifyObjectAction is to dynamically change the classifiers of an object. The System Model currently supports reclassification, but only in terms of casting (i.e., the new classifier must be related to the old classifier). On the one hand, it would not be technically difficult to reclassify an object (i.e., remove the object identifier from one carrier set and place it in another). On the other hand, defining the exact semantics of reclassification is an ongoing research area [26]. For simplicity, we decided to not attempt the formal definition of this action without further research.

Finally, for simplicity, we do not define the ReduceAction, UnmarshallAction, RaiseExceptionAction, or OpaqueAction.

5.2.2 Example 1: Create and Read Object Classifier

The activity shown in Figure 5.1 creates an object of type *Car*, using the CreateObjectAction. It then checks that the classifier of the newly created object is in fact *Car*,

using the ReadIsClassifiedObjectAction. The value of object node labelled #5 will be *true* if the newly created object is a car object.

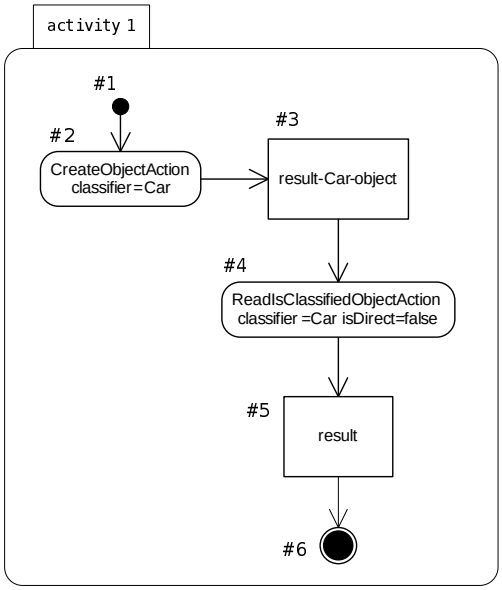


Figure 5.1: Example activity to execute CreateObjectAction and ReadIsClassifiedObjectAction

See Section B.1 for the ADLF representation of this activity and the input file(s) required for execution. Sample output from executing this activity is shown in Figure 5.2.

```
Printing the order in which nodes were visited - (4)
[1, 2, 4, 6]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #3[result-Car-object] = OIDfor_Car_1
.  #5[result] = vTrue
.....................................................................

Running time = 541 ms = 0 sec = 0 min
```

Figure 5.2: Partial output of a sample execution of activity in Figure 5.1

5.2.2.1 CreateObjectAction

The CreateObjectAction is described as "an action that creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime" [59, §11.3.16]. The metamodel for this action is shown in Figure A.10 on page 278. There are no specific input pins for this action. The classifier is statically specified by an association between the CreateObjectAction and a Classifier object. In addition, the action has no other effect than to create the new object with the given classifier. Specifically, no behaviours are executed, nothing is triggered, the new object has no structural feature values and participates in no links.

In our example in Figure 5.1, the CreateObjectAction is shown at node #2, with its output pin as the object node #3. The static information about the classifier is provided as an annotation in the action node, i.e., "classifier=Car".

Let *coa* be an action node representing an instance of CreateObjectAction. The effect of executing *coa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *coa* is enabled.[2]

Post-conditions The execution of *coa* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= addobj(ds, oid, \{\}) \\ cs' &= updateOutput(cs, coa, \text{“result”}, oid) \\ es' &= addBuf(ds, es, oid) \end{aligned}$$

where

- $addobj : DataStore \times UOID \times (ULOC \rightarrow UVAL) \rightarrow DataStore$ is a System Model function to add a new object to the data store [13, p.20], such that

  $$addobj((s, m), oid, f) = (s \cup \{oid\}, m \oplus f)$$

  and

  - $oid \in UOID$ is the object identifier of the newly created object

    $$oid = newoid(ds, className)$$

  - $newoid : DataStore \times UCLASS \rightarrow UOID$ is a function that returns an object identifier that is not yet used in the data store, that is

    $$newoid(ds, c) \in CAR(c) \wedge newoid(ds, c) \notin ds.s$$

    The new object identifier is in the carrier set of the classifier and did not exist in the previous state's data store.

  - $className \in UCLASS$ is the classifier to be instantiated

    $$className = coa.action.attrs(\text{“classifier”})$$

  - $f \subseteq (ULOC \rightarrow UVAL)$ is the set of mappings from the new object's attributes to values.[3]

[2]The precise definition of *enabled* is contained in Section 4.3. An action node is considered enabled if it is not stalled (due to the nature of token offers and passing), it does not represent a call action waiting for a result, it does not represent an accept action waiting for an event, and it has sufficient tokens on its incoming edges.

[3]The System Model *addobj* function permits the creation of a new object *and* the assignment of values to its attributes. However, UML does not support the initialization of a new object's attributes during the execution of the CreateObjectAction; hence we set $f = \{\}$.

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \to ControlStore$ is a function that sets the value of an output pin in the control store, such that

$$\begin{aligned} updateOutput(cs, an, s, v) =& (updateDiagrams(cs.ads, an, s, v), \\ & cs.pc, cs.thr, cs.vars) \end{aligned}$$

  where

  - $updateDiagrams : \mathcal{P}\,ActivityDiagram \times ActionNode \times String \times UVAL \to \mathcal{P}\,ActivityDiagram$ is a function that sets the value of an output pin in the set of activity diagrams, such that

$$updateDiagrams(ads, an, s, v) = \{updateDiagram(ad, an, s, v) \mid ad \in ads\}$$

  - $updateDiagram : ActivityDiagram \times ActionNode \times String \times UVAL \to ActivityDiagram$ is a function that sets the value of an output pin in one activity diagram from the control store, such that

$$updateDiagram(ad, an, s, v) = \begin{cases} ad & \text{if } an \notin ad.nodes \\ (ad.id, ad.nodes \setminus an & \\ \cup updateNode(an, s, v), & \\ ad.localVars) & \text{otherwise} \end{cases}$$

    The new diagram is like the old diagram, except that the action node is updated.

  - $updateNode : ActionNode \times String \times UVAL \to ActionNode$ is a function that sets the value of an output pin of an action node, such that

$$\begin{aligned} updateNode(an, s, v) =& (an.id, an.in, an.out, \\ & updateAction(an.action, s, v), \\ & an.annot, an.waiting, an.owner, an.stalled) \end{aligned}$$

    The new action node is like the old node, except that its action is updated.

  - $updateAction : Action \times String \times UVAL \to Action$ is a function that updates the value of an output pin on an action, such that

$$\begin{aligned} updateAction(a, s, v) = &(a.name, a.inPins, \\ &a.outPins \oplus [s = v], \\ &a.attrs) \end{aligned}$$

The new action is like the old action, except that its output pins are updated. The new output pins are all the old output pins that are not named by the string, plus a new pin with the string and value argument.

- $addBuf : DataStore \times EventStore \times UOID \rightarrow EventStore$ is a function that adds a mapping from the newly created object to its buffer, such that

$$addBuf(ds, es, o) = es \oplus [o = newBuf(ds, es)]$$

The new event buffer (which is a map from UOID to Buffer(UEVENT)) is the same as the old buffer, except that a mapping from the oid to a new buffer is added.

and

  - $newBuf : DataStore \times EventStore \rightarrow Buffer(UEVENT)$ is a function that retrieves an event buffer not yet used in the event store, such that

$$\begin{aligned} &newBuf(ds, es) \in Buffer(UEVENT) \\ \wedge &\neg \exists o \in ds.s \mid es(o) = newBuf(ds, es) \end{aligned}$$

The new buffer is not used for any other object in the current data store.

5.2.2.2 ReadIsClassifiedObjectAction

The ReadIsClassifiedObjectAction is described as "an action that determines whether a runtime object is classified by a given classifier" [59, §11.3.32].

The metamodel for this action is shown in Figure A.13 on page 279. The action has one input pin, object, that holds the object identifier whose classification is to be tested. There is also an association, classifier of type Classifier, that is the classifier to test the incoming object against. In addition, the action has an attribute, isDirect,

that indicates whether or not the test should test the incoming object against directly against the specified classifier. Otherwise, the object's classifier is tested against the given classifier and its subclasses.[4] Finally, the action has one output pin, result, to hold the result of the test.

In our example in Figure 5.1, the input object to be tested is held in the object node #3. The ReadIsClassifiedObjectAction is shown as node #4, and the result as the object node #5. There are two annotations in the action node, one for the classifier and one for the value of the isDirect attribute. Although there is no default value for the classifier, there is one for the isDirect attribute; if the latter annotation was missing, the interpreter would use the default value of *false*.

Let *ricoa* be an action node representing an instance of ReadIsClassifiedObjectAction. The effect of executing *ricoa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *ricoa* is enabled.

Post-conditions The execution of *ricoa* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= updateOutput(cs, ricoa, \text{“result”}, inClass(oid, c, isDirect)) \\ es' &= es \end{aligned}$$

where

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).

[4] For example, if *Car* extends *Vehicle*, a ReadIsClassifiedObjectAction with *classifier* = *Vehicle* and *isDirect* = *true* would check the target object against *Vehicle* only. If *isDirect* = *false*, the action would check the object against both *Vehicle* and *Car*.

- $inClass : UOID \times UCLASS \times Boolean \rightarrow Boolean$ is a function that indicates whether or not an object is classified by a particular classifier, such that

$$inClass(oid, c, isDirect) = \begin{cases} oid \in CAR(c) & \text{if } isDirect = true \\ (classOf(oid), c) \in sub & \text{otherwise} \end{cases}$$

  and

  - $classOf : UOID \rightarrow UCLASS$ is a System Model function to retrieve the classifier of an object identifier [14, p.5], such that

$$oid \in CAR(classOf(oid))$$

  - $sub \subseteq UCLASS \times UCLASS$ is the subclassing (inheritance) relation, discussed in Section 3.2.2.5 on page 46.

- $oid \in UOID$ is the object to be tested

$$oid = ricoa.action.inPins(\text{“object”})$$

- $c \in UCLASS$ is the classifier to be tested against

$$c = ricoa.action.attrs(\text{“classifier”})$$

- $isDirect \in Boolean$ indicates whether or not the test must be made against the classifier directly, or if its subclasses are included

$$isDirect = ricoa.action.attrs(\text{“isDirect”})$$

5.2.3 Example 2: Specify Value and Test Identity

The activity shown in Figure 5.3 uses the ValueSpecification action to retrieve existing objects from the System Model universe, and then uses the TestIdentityAction to compare the objects. The value of the object node #8 will be *true* if both retrieved objects are identical.

See Section B.2 for the ADLF representation of this activity, the input file(s)

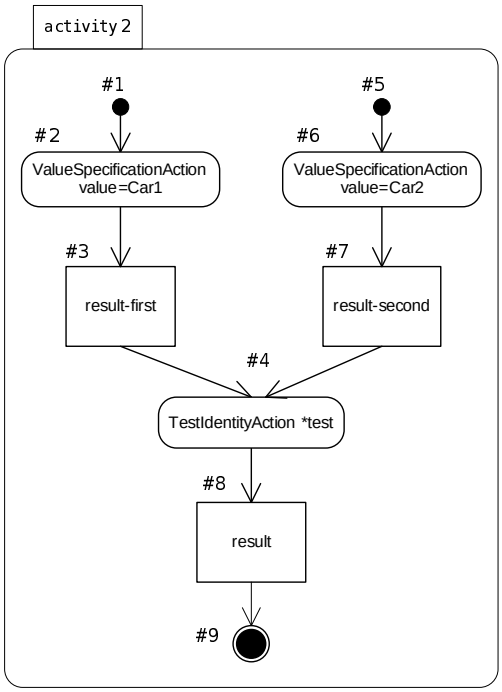


Figure 5.3: Example activity to execute ValueSpecificationAction and TestIdentityAction

required for execution, as well as sample output from the interpreter.

5.2.3.1 ValueSpecificationAction

The ValueSpecificationAction is described as "an action that evaluates a value specification" [59, §11.3.51]. A ValueSpecification is "an abstract metaclass used to identify a value or values in a model. It may reference an instance or it may be an expression denoting an instance or instances when evaluated" [59, §7.3.54].

Figure 5.4 shows the UML metamodel hierarchy stemming from the abstract class ValueSpecification. As evident from the hierarchy, there are many kinds of value specification supported by UML. We restrict our work to the following three:

- The object identifier *OID* represents the UML InstanceValue.

- The type name $Int \in UTYPE$ represents the UML LiteralInteger.
- The type name $Boolean \in UTYPE$ represents the UML LiteralBoolean.

Note that other type names could be defined to represents the other UML literal specifications. For instance, we could define a type name $String \in UTYPE$ and populate its carrier set with all strings in the universe.

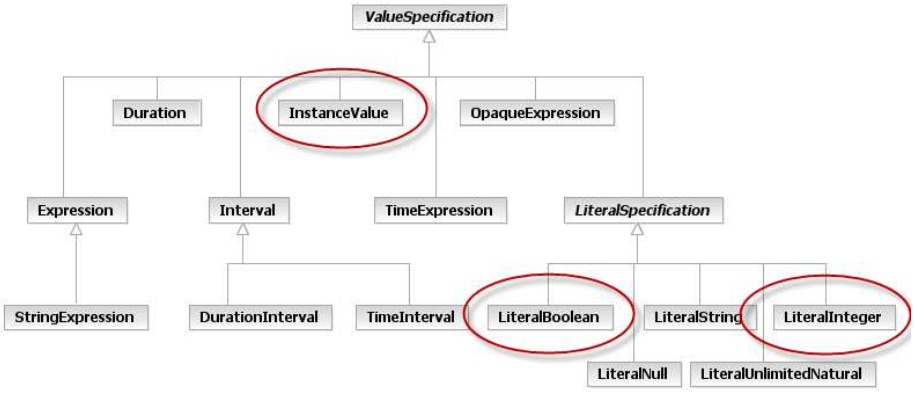


Figure 5.4: UML metamodel hierarchy of ValueSpecification. We restrict our work to the InstanceValue (*OID*), LiteralInteger (type name *Int*) and LiteralBoolean (type name *Boolean*) value specifications

The metamodel for the ValueSpecificationAction is shown in Figure A.24 on page 284. The action has an association, value of type ValueSpecification, that indicates the value specification to be evaluated (or, in our case, the value or object identifier to be retrieved). The action also has an output pin, result, to hold the retrieved value.

In our example in Figure 5.3, there are two ValueSpecificationActions, annotated with "Car1" and "Car2". Before executing this activity, the universe has been populated with two instances of "Car". Therefore, these actions will simply retrieve the

two instances from the universe, and place them in their respective output pins, i.e., the object nodes #3 and #7.

Let *vsa* be an action node representing an instance of ValueSpecificationAction. The effect of executing *vsa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *vsa* is enabled.

Post-conditions The execution of *vsa* results in a post-state $st' = (ds', cs', es')$, such that

$$
\begin{aligned}
ds' &= ds \\
cs' &= updateOutput(cs, vsa, \text{“result”}, vsa.action.attrs(\text{“value”})) \\
es' &= es
\end{aligned}
$$

where

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).

Notation For convenience sake, we often use an object node with no sources as shorthand for a ValueSpecificationAction, where the object node refers to the result of that action. For example, consider the activities in Figure 5.5. The solitary object node in Figure 5.5(b) is equivalent to using an explicit ValueSpecificationAction with its result object node, as shown in Figure 5.5(a).

5.2.3.2 TestIdentityAction

The TestIdentityAction is described as “an action that tests if two values are identical objects” [59, §11.3.48].

(a) ValueSpecificationAction (b) Shorthand

Figure 5.5: An object node with no source is shorthand for using a ValueSpecificationAction. The two examples shown are equivalent

The metamodel for this action is shown in Figure A.23 on page 284. The action has two input pins, first and second, to hold the two values[5] to be compared. It also has an output pin, result, to hold the result of the test.

For our example in Figure 5.3, the TestIdentityAction in node #4 will receive as input two objects, i.e., *Car1* and *Car2*, and compare them. They are not the same object and the result placed in the result pin, i.e., node #8, will be *false*, as can be seen in the output in Figure B.14 on page 291.

Let *tia* be an action node representing an instance of TestIdentityAction. The effect of executing *tia* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *tia* is enabled.

[5]Remember that object identifiers are kinds of values, i.e., $UOID \subseteq UVAL$.

Post-conditions The execution of *tia* results in a post-state $st' = (ds', cs', es')$, such that

$$ds' = ds$$
$$cs' = updateOutput(cs, tia, \text{“result”}, equals(val1, val2))$$
$$es' = es$$

where

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).
- $equals : UVAL \times UVAL \rightarrow Boolean$ is a function that tests the equality of two values, such that

$$equals(v1, v2) = \begin{cases} true & \text{if } v1 = v2 \\ false & \text{otherwise} \end{cases}$$

  Remember that $UOID \subseteq UVAL$ so this action can also be used to test equality between object identifiers.

- $val1 \in UVAL$ is the first value to be tested for equality

$$val1 = tia.action.inPins(\text{“first”})$$

- $val2 \in UVAL$ is the second value to be tested for equality

$$val2 = tia.action.inPins(\text{“second”})$$

5.2.4 Example 3: Read Extent and Destroy Object

The activity shown in Figure 5.6 reads the extent of the classifier *Car*; the result of this action is a collection of all instances of that type existing in the current execution. Then, one of the instances is deleted, and the extent of *Car* is re-read.

See Section B.3 for the ADLF representation of this activity, the input file(s) required for execution, as well as sample output from the interpreter.

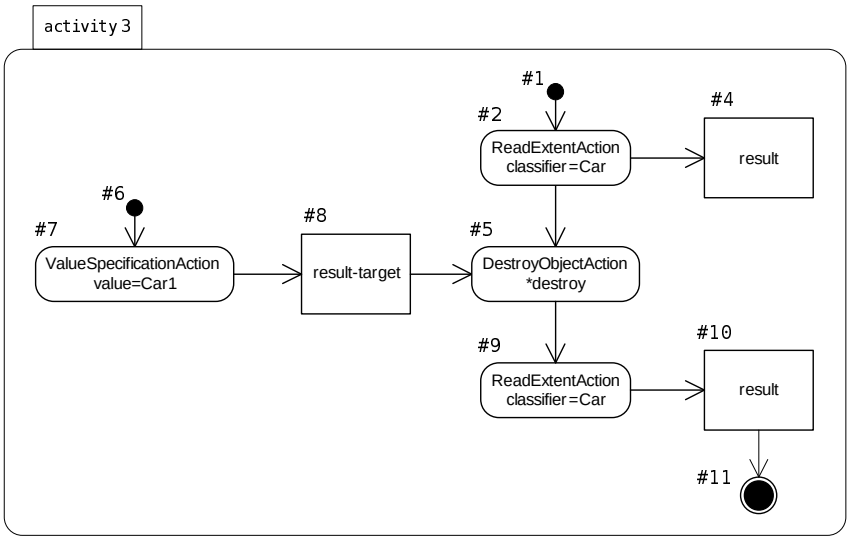


Figure 5.6: Example activity to execute ReadExtentAction and DestroyObjectAction

5.2.4.1 ReadExtentAction

The ReadExtentAction is described as "an action that retrieves the current instances of a classifier" [59, §11.3.31].

The metamodel for this action is shown in Figure A.12 on page 279. The action has no input pins. It has one association, classifier of type Classifier, that is the classifier (i.e., *ClassName*) to retrieve the instances of. Finally, the action has one output pin, result, to hold the result. We have created a collection type for the System Model; the result will be an instance of this collection type, holding all of the instances of the specified class name.

For our example in Figure 5.6, the ReadExtentAction appears twice, at nodes #2 and #9. Each of these nodes references a different instance of ReadExtentAction. Both nodes have been annotated with "classifier=Car", meaning that they are retrieving instances of the *Car* class, before and after a DestroyObjectAction is executed. We

expect that the result of the second ReadExtentAction will be missing the destroyed object; this is confirmed by the output in Figure B.14 on page 291.

Let *rea* be an action node representing an instance of ReadExtentAction. The effect of executing *rea* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *rea* is enabled.

Post-conditions The execution of *rea* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= updateOutput(cs, rea, \text{“result”}, objects) \\ es' &= es \end{aligned}$$

where

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).
- $objects \in UVAL$ is a collection value that holds all object identifiers classified by the given classifier

  $$objects = \{o \in ds.s \mid o \in CAR(rea.action.attrs(\text{“classifier”}))\}$$

  The collection holds the set of all object identifiers in the data store that are in the carrier set of the classifier.

5.2.4.2 DestroyObjectAction

The DestroyObjectAction is described as “an action that destroys objects” [59, §11.3.18]. The metamodel for this action is shown in Figure A.11 on page 278. The action has one input pin, target, that is the object to be destroyed. In addition, the action has

two boolean attributes. The first, isDestroyLinks, indicates whether or not to also destroy any links (i.e., instances of associations) in which the object participates. We do not support links or associations; hence this attribute is not supported. The second attribute, isDestroyOwnedObjects, indicates whether or not objects owned (through composite aggregation) by the target object are to be destroyed. Again, we do not support associations, so this attribute is also not supported. The DestroyObjectAction has no output pins.

When an object is destroyed "the classifiers of the objects are removed as its classifiers, and the object is destroyed" [59, §11.3.18]. According to the UML specification, there is no other effect. More specifically, references to the destroyed object are not changed. We do destroy the object's event buffer.

For our example in Figure 5.6, the DestroyObjectAction is shown at node #5; its target is the object in node #8. We can see that the target object is destroyed by the sample output in Figure B.14 on page 291—the output of the first ReadExtentAction contains the target object while the output of the second ReadExtentAction does not.

Let *doa* be an action node representing an instance of DestroyObjectAction. The effect of executing *doa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *doa* is enabled.

Post-conditions The execution of *doa* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= delObj(ds, oid) \\ cs' &= cs \\ es' &= delBuf(es, oid) \end{aligned}$$

where

- $oid \in UOID$ is the object identifier of the object to be deleted

$$oid = doa.action.inPins(\text{“target”})$$

- $delObj : DataStore \times UOID \to DataStore$ is a function to delete an object from the data store, such that

$$delObj((ds.s, ds.m), oid) = (s \setminus oid, delMaps(m, oid))$$

and

  - $delMaps : (ULOC \to UVAL) \times UOID \to (ULOC \to UVAL)$ is a function to remove the mappings associated with the deleted object's attributes, such that

$$delMaps(m, oid) = \{(loc, val) \in m \mid loc \notin getLocs(oid)\}$$

The new set of mappings is like the old set, but we remove any individual mapping whose domain is in the set getLocs(oid).

  - $getLocs : UOID \to \mathcal{P}\,ULOC$ is a function that returns the set of locations used to store an object's attributes, such that

$$getLocs(oid) = \bigcup_{v \in attr(classOf(oid))} proj(oid, v)$$

  - $attr : UCLASS \to \mathcal{P}\,UVAR$ is a function that returns the attributes of an object (see page 45).
  - $classOf : UOID \to UCLASS$ is a function to retrieve the classifier of an object identifier (see page 90).
  - $proj : UOID \times UVAR \to ULOC$ is a function that projects an attribute onto an object, returning the location of that attribute's value (see page 46).

- $delBuf : (UOID \to Buffer(UEVENT)) \times UOID \to (UOID \to Buffer(UEVENT))$ is a function to remove the mapping from an object to its buffer, such that

$$delBuf(es, oid) = \{(o, b) \in es \mid o \neq oid\}$$

The new event buffer (which is a map from UOID to Buffer(UEVENT)) is the same as the old buffer, but any mapping from the oid is removed.

5.2.5 Example 4: Structural Feature Actions

The activity shown in Figure 5.7 performs several actions on a structural feature. A StructuralFeature is a "typed feature of a classifier that specifies the structure of instances of the classifier" [59, §7.3.49]. We consider structural features to be attributes, or fields, of classes. To set an object's attribute, we use the AddStructuralFeatureValueAction; to read it, we use the ReadStructuralFeatureAction, etc.

In our example, the *year* structural feature, or attribute, of *Car1* is modified to the value *v2005* by the AddStructuralFeatureValueAction. The attribute is read by the ReadStructuralFeatureAction. Then, the attribute value is completely cleared with the ClearStructuralFeatureAction. Finally, the attribute is read again. At this point, the attribute has no value, resulting in a null result, as can be seen by the partial output in Figure B.19 on page 294.

See Section B.4 for the ADLF representation of this activity, the input file(s) required for execution, as well as sample output from the interpreter.

Aside: Single vs. Multiple Values The UML specification is quite clear that a structural feature can hold multiple values, which may be ordered or unordered. On the other hand, the System Model is defined such that a maximum of one value may be assigned to an object's attribute. This restriction is evident by the definition of the projection function, as well as the retrieval function defined on the data store.

To maintain consistency with the System Model, we restrict structural features to a maximum of one value. This, in turn, restricts the execution of the structural feature actions. For instance, adding a value to a structural feature will automatically overwrite any previous value, reading a structural feature will result in at most one

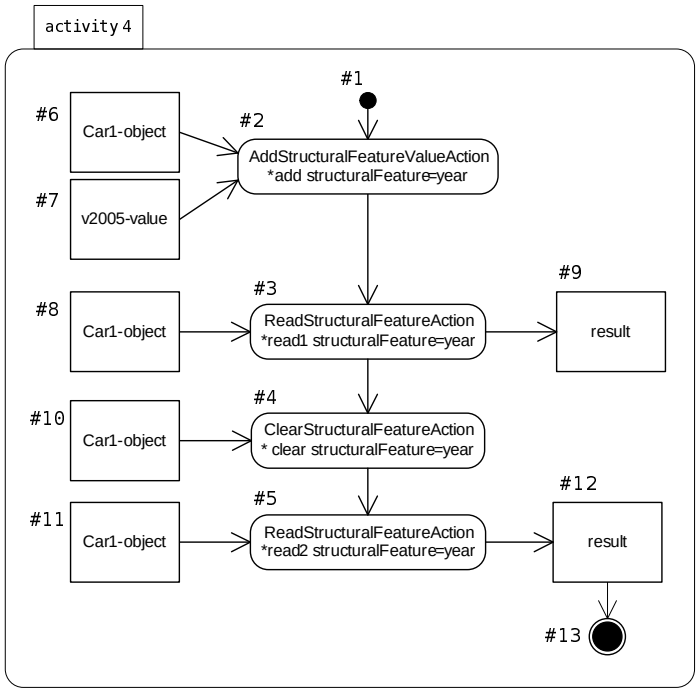


Figure 5.7: Example activity to execute AddStructuralFeatureValueAction, ReadStructuralFeature-Action and ClearStructuralFeatureAction

value, etc.

5.2.5.1 AddStructuralFeatureValueAction

The AddStructuralFeatureValueAction is described as "a write structural feature action for adding values to a structural feature" [59, §11.3.5]. We make use of this action to assign a value to an attribute of an object. The metamodel for this action is shown in Figure A.3 on page 275. The action inherits two input pins, object and value. The object pin contains the object whose attribute is to be set and the value pin contains the value which will be assigned to that attribute. The action also inherits an

association, structuralFeature of type StructuralFeature, that indicates which attribute of the incoming object is to be modified.

As discussed above, a structural feature can typically contain multiple values, and be ordered or unordered. To support this ability, the action has another input pin, insertAt, to indicate where in an ordered collection of values the new value should be added. Similarly, the action has an attribute, isReplaceAll, that specifies whether or not the existing values of the structural feature should be removed before adding the new value. Because we are restricted to storing single values in structural features, we do not make use of the insertAt pin or the isReplaceAll attribute. The new value always overwrites any previously existing value.

For our example in Figure 5.7, the AddStructuralFeatureValueAction is shown at node #2. The structural feature (or attribute) to be modified is indicated with the action's annotation "structuralFeature=year". The object to be modified is in the object node #6 and the value to be added is in the object node #7. The result of this action is equivalent to the pseudocode statement `Car1.year = v2005`.

Let *asfva* be an action node representing an instance of AddStructuralFeatureValueAction. The effect of executing *asfva* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *asfva* is *enabled*.

Post-conditions The execution of *asfva* results in a post-state $st' = (ds', cs', es')$, such that

$$ds' = setval(ds, oid, at, v)$$
$$cs' = cs$$
$$es' = es$$

where

- $setval : DataStore \times UOID \times UVAR \times UVAL \rightarrow DataStore$ is a System Model function to modify the value of an attribute mapping in the data store [13, p.20], such that

$$setval((s, m), oid, at, v) = (s, m \oplus [proj(oid, at) = v])$$

- $oid \in UOID$ is the object identifier whose attribute will be modified

$$oid = asfva.action.inPins(\text{“object”})$$

- $at \in UVAR$ is the attribute to be modified

$$at = asfva.action.attrs(\text{“structuralFeature”})$$

- $v \in UVAL$ is the new value of the attribute

$$v = asfva.action.inPins(\text{“value”})$$

- $proj : UOID \times UVAR \rightarrow ULOC$ is a function that projects an attribute onto an object, returning the location of that attribute's value (see page 46).

5.2.5.2 ReadStructuralFeatureAction

The ReadStructuralFeatureAction is described as “a structural feature action that retrieves the values of a structural feature” [59, §11.3.37]. We use this action to read the single value assigned to an attribute of an object. The metamodel for this action is shown in Figure A.15 on page 280. The action inherits one input pin, object, that contains the object whose attribute is to be retrieved. The action also inherits an association, structuralFeature of type StructuralFeature, that indicates which attribute of the incoming object is to be read. Finally, the action has one output pin, result, to hold the value read from the structural feature.

For our example in Figure 5.7, the ReadStructuralFeatureAction is used twice, in nodes #3 and #5. These nodes represent two separate instances of the action. The first read is performed after the *Car1.year* attribute has been set. The second read is performed after the attribute has been cleared. For each action, the structural feature (or attribute) to be read is indicated with the action's annotation "structuralFeature=year".

Let *rsfa* be an action node representing an instance of ReadStructuralFeatureAction. The effect of executing *rsfa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *rsfa* is enabled.

Post-conditions The execution of *rsfa* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= updateOutput(cs, rsfa, \text{“result”}, val(ds, oid, at)) \\ es' &= es \end{aligned}$$

where

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).
- $val : DataStore \times UOID \times UVAR \rightarrow UVAL$ is a System Model function to retrieve the value of a given object and attribute [13, p.20], such that

$$val((s, m), oid, at) = m(proj(oid, at))$$

- $oid \in UOID$ is the object whose attribute is to be read

$$oid = rsfa.action.inPins(\text{“object”})$$

- $at \in UVAR$ is the attribute to be read

$$at = rsfa.action.attrs(\text{“structuralFeature”})$$

- $proj : UOID \times UVAR \rightarrow ULOC$ is a function that projects an attribute onto an object, returning the location of that attribute's value (see page 46).

5.2.5.3 ClearStructuralFeatureAction

The ClearStructuralFeatureAction is described as “a structural feature action that removes all values of a structural feature” [59, §11.3.12]. We use this action to remove the value assigned to an object's attribute.

The metamodel for this action is shown in Figure A.8 on page 277. The action inherits one input pin, object, that identifies the object whose structural feature will be cleared. It also inherits an association, structuralFeature of type StructuralFeature, that indicates which attribute of the incoming object is to be modified. The action has no output pins.

For our example in Figure 5.7, the ClearStructuralFeatureAction is shown at node #4. The structural feature (or attribute) to be modified is indicated with the action's annotation “structuralFeature=year”. The object to be modified is in the object node #10.

Let *csfa* be an action node representing an instance of ClearStructuralFeatureAction. The effect of executing *csfa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *csfa* is enabled.

Post-conditions The execution of *csfa* results in a post-state $st' = (ds', cs', es')$, such that

$$ds' = (ds.s, clearMap(ds.m, oid, at))$$
$$cs' = cs$$
$$es' = es$$

where

- $clearMap : (ULOC \rightarrow UVAL) \times UOID \times UVAR \rightarrow (ULOC \rightarrow UVAL)$ is a function to clear all mappings of an attribute of an object, such that

  $$clearMap(m, oid, at) = \{(loc, val) \in m \mid loc \neq proj(oid, at)\}$$

  The new mappings are the same as the old, less any that map from the incoming object's attribute.

- $oid \in UOID$ is the object whose attribute will be cleared

  $$oid = csfa.action.inPins(\text{“object”})$$

- $at \in UVAR$ is the attribute that will be cleared

  $$at = csfa.action.attrs(\text{“structuralFeature”})$$

- $proj : UOID \times UVAR \rightarrow ULOC$ is a function that projects an attribute onto an object, returning the location of that attribute's value (see page 46).

5.2.5.4 RemoveStructuralFeatureValueAction

The RemoveStructuralFeatureValueAction is described as “a write structural feature action that removes values from structural features” [59, §11.3.41]. We use this action to remove a specific value from an object's attribute.[6]

The metamodel for this action is shown in Figure A.17 on page 281. The action inherits an input pin, object, that specifies which object whose structural feature will

[6]This action differs slightly from the previously defined ClearStructuralFeatureAction in that this action removes a specific value from the structural feature. The ClearStructuralFeatureAction removes any/all value(s) from the structural feature.

be modified. It inherits an association, structuralFeature of type StructuralFeature, that indicates which attribute of the incoming object is to be modified. It also inherits an input pin, value, that identifies which value is to be removed from the structural feature.

As discussed above, a structural feature can contain multiple values, ordered or unordered. To support this ability, the action has another input pin, removeAt, to indicate the position of the value to be removed from an ordered collection. Finally, the action has a boolean attribute isRemoveDuplicates, used to indicate whether or not all instances of the indicated value are to be removed. Because we are restricted to storing single values in structural features, we do not make use of the removeAt pin or the isRemoveDuplicates attribute. We simply remove any instance of the specified value found in the structural feature.

Let *rsfva* be an action node representing an instance of RemoveStructuralFeatureValueAction. The effect of executing *rsfva* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *rsfva* is enabled.

Post-conditions The execution of *rsfva* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= (ds.s, removeMap(ds.m, oid, at, v)) \\ cs' &= cs \\ es' &= es \end{aligned}$$

where

- $removeMap : (ULOC \rightarrow UVAL) \times UOID \times UVAR \times UVAL \rightarrow (ULOC \rightarrow$

UVAL) is a function to remove any occurrence of a specific value from an attribute of an object, such that

$$removeMap(m, oid, at, v) = \{(loc, val) \in m \mid loc \neq proj(oid, at) \vee val \neq v\}$$

The new set of mappings is the same as the old set, less any that have the same location as the incoming object's attribute and the same value.

- $oid \in UOID$ is the object whose attribute is to be modified

$$oid = rsfva.action.inPins(\text{“object”})$$

- $at \in UVAR$ is the attribute to be modified

$$at = rsfva.action.attrs(\text{“structuralFeature”})$$

- $v \in UVAL$ is the value to be removed from the attribute

$$v = rsfva.action.inPins(\text{“value”})$$

- $proj : UOID \times UVAR \rightarrow ULOC$ is a function that projects an attribute onto an object, returning the location of that attribute's value (see page 46).

5.2.6 Example 5: Variable Actions

The activity shown in Figure 5.8 performs several actions on a (local) variable. Variables are used to pass “data between actions indirectly” [59, §12.3.52]. A variable is local to a particular structured activity group, enabling actions in that group to share information. For simplicity, we do not consider structured activity nodes (e.g., ConditionalNode, LoopNode, etc.), interruptible activity regions, or partitions. We therefore define the scope of a variable to be the activity in which it is present.

In our example, the variable x is set to the value *v2008* by the AddVariableValueAction. The variable is read by the ReadVariableValue. Then the variable is completely cleared with the ClearVariableAction. Finally, the variable is read again. At this point,

the variable has no value, resulting in a null result, as can be seen by the partial output in Figure B.23 on page 296.

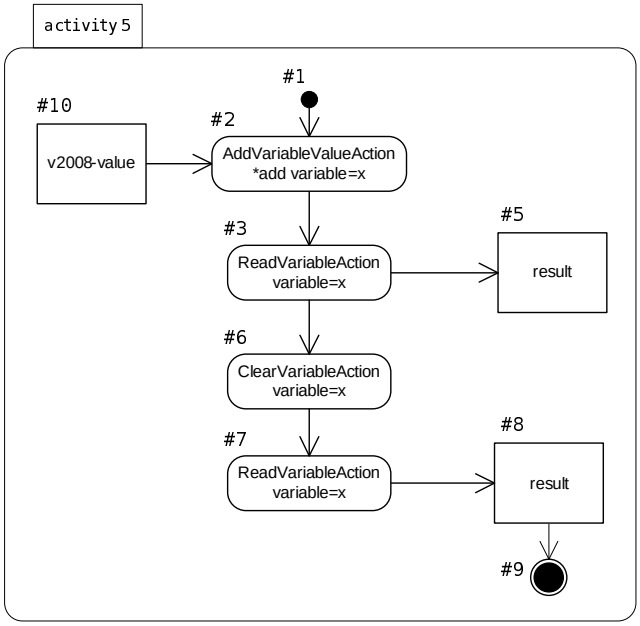


Figure 5.8: Example activity to execute AddVariableValueAction, ReadVariableAction and ClearVariableAction

See Section B.5 for the ADLF representation of this activity, the input file(s) required for execution, as well as sample output from the interpreter.

Aside: Single vs. Multiple Values The variable actions are similar to the structural feature actions in that there is an action for writing to a variable, an action for reading from a variable, and two actions for removing from or clearing a variable. The UML specification is quite clear that, like structural features, variables can hold multiple values, which may be ordered or unordered.

As discussed in Section 4.1.3, we added the concept of local variables to the System

Model, in a manner similar to how object attributes were stored. That is to say that, for consistency, we assume that a local variable may store at most one value, just as an attribute may have at most one value. This assumption restricts the execution of variable actions. For instance, adding a value to a variable will automatically overwrite any previous value, reading a variable will result in at most one value, etc.

5.2.6.1 AddVariableValueAction

The AddVariableValueAction is described as "a write variable action for adding values to a variable" [59, §11.3.6]. This action is used to add a value to a variable local to a particular activity. The metamodel for this action is shown in Figure A.4 on page 275. The action inherits a pin, value, that contains the value to assign to the specified variable. The variable itself is specified by the variable association that this action has with Variable.

As discussed above, a variable can typically contain multiple values, and be ordered or unordered. To support this ability, the action has another input pin, insertAt, to indicate where in an ordered collection of values the new value should be added. Similarly, the action has an attribute, isReplaceAll, that specifies whether or not the existing values of the variable should be removed before adding the new value. Because we are restricted to storing single values in local variables, we do not make use of the insertAt pin or the isReplaceAll attribute. The new value always overwrites any previously existing value.

For our example in Figure 5.8, the AddVariableValueAction is shown at node #2. The variable to be modified is indicated with the action's annotation "variable=x".

The value to be added to this variable is in the object node `#10`. The result of this action is equivalent to the pseudocode statement `x = v2008`.

Let *avva* be an action node representing an instance of AddVariableValueAction. The effect of executing *avva* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *avva* is enabled.

Post-conditions The execution of *avva* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' =& ds \\ cs' =& (cs.ads, cs.pc, cs.thr, \\ & (cs.vars.varloc, setVarVal(cs.vars.locval, cs.vars.varloc(var), val))) \\ es' =& es \end{aligned}$$

where

- $setVarVal : (ULOC \nrightarrow UVAL) \times ULOC \times UVAL \rightarrow (ULOC \nrightarrow UVAL)$ is a function to set the value of a local variable, such that

$$setVarVal(lv, loc, val) = lv \oplus [loc = val]$$

- $var \in UVAR$ is the local variable to be modified

$$var = avva.action.attrs(\text{“variable”})$$

- $val \in UVAL$ is the value to assign to the local variable

$$val = avva.action.inPins(\text{“value”})$$

5.2.6.2 ReadVariableAction

The ReadVariableAction is described as “a variable action that retrieves the values of a variable” [59, §11.3.38]. We use this action to read the single value assigned to a

local variable of an activity. The metamodel for this action is shown in Figure A.16 on page 280. The action inherits no input pins. It inherits an association, variable of type Variable, which indicates which variable is to be read. Finally, the action has one output pin, result, to hold the value of the variable.

For our example in Figure 5.8, the ReadVariableAction is used twice, in nodes #3 and #7. The nodes represent two separate instances of the action. The first read is performed after the variable x has been set. The second read is performed after the variable has been cleared. For each action, the variable to be read is indicated with the action's annotation "variable=x".

Let *rva* be an action node representing an instance of ReadVariableAction. The effect of executing *rva* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *rva* is enabled.

Post-conditions The execution of *rva* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= updateOutput(cs, rva, \text{“result”}, getVarVal(cs, var)) \\ es' &= es \end{aligned}$$

where

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).
- $getVarVal : ControlStore \times UVAR \nrightarrow UVAL$ is a function to retrieve the current value of a local variable, such that

$$getVarVal(cs, var) = \begin{cases} cs.vars. & \text{if } cs.vars.locval(cs.vars. \\ \quad locval(cs.vars.varloc(var)) & \quad varloc(var)) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $var \in UVAR$ is the local variable to be read

$$var = rva.action.attrs(\text{“variable”})$$

- The functions *varloc* and *locval* are defined on page 65.

5.2.6.3 ClearVariableAction

The ClearVariableAction is described as “a variable action that removes all values of a variable” [59, §11.3.13]. We use this action to remove the value assigned to a local variable.

The metamodel for this action is shown in Figure A.9 on page 278. The action has no input pins. It inherits an association, variable of type Variable, which indicates which variable is to be modified. The action has no output pins.

For our example in Figure 5.8, the ClearVariableAction is shown at node #6. The variable to be modified is indicated with the action’s annotation “variable=x”.

Let *cva* be an action node representing an instance of ClearVariableAction. The effect of executing *cva* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *cva* is enabled.

Post-conditions The execution of *cva* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' =& ds \\ cs' =& (cs.ads, cs.pc, cs.thr, \\ & (cs.vars.varloc, clearVar(cs.vars.locval, var, cs.vars.varloc))) \\ es' =& es \end{aligned}$$

where

- $clearVar : (ULOC \rightharpoonup UVAL) \times UVAR \times (UVAR \rightarrow ULOC) \rightarrow (ULOC \rightharpoonup UVAL)$ is a function to clear the value of a local variable, such that

$$clearVar(lv, var, vl) = \{(l, v) \in lv \mid l \neq vl(var)\}$$

  The new location-value mappings are the same as the old, less any that map from a particular location. That location is associated with the given variable in the variable-location mappings.

- $var \in UVAR$ is the local variable to be cleared

$$var = cva.action.attrs(\text{“variable”})$$

- The functions *varloc* and *locval* are defined on page 65.

5.2.6.4 RemoveVariableValueAction

The RemoveVariableValueAction is described as “a write variable action that removes values from variables” [59, §11.3.42]. We use this action to remove a specific value from a local variable.[7]

The metamodel for this action is shown in Figure A.18 on page 281. The action has no input pins. It inherits an association, variable of type Variable, which indicates

[7] This action differs slightly from the previously defined ClearVariableAction in that this action removes a specific value from the variable. The ClearVariableAction removes any/all value(s) from the variable.

which variable to be modified. It also inherits an input pin, value, which identifies which value is to be removed from the local variable.

As discussed above, a local variable can contain multiple values, ordered or unordered. To support this ability, the action has another input pin, removeAt, to indicate the position of the value to be removed from an ordered collection. Finally, the action has a boolean attribute isRemoveDuplicates, used to indicate whether or not all instances of the indicated value are to be removed. Because we are restricted to storing single values in local variables, we do not make use of the removeAt pin or the isRemoveDuplicates attribute. We simply remove any instance of the specified value found in the local variable.

Let *rvva* be an action node representing an instance of RemoveVariableValueAction. The effect of executing *rvva* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *rvva* is enabled.

Post-conditions The execution of *rvva* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' =& ds \\ cs' =& (cs.ads, cs.pc, cs.thr, \\ & \quad (cs.vars.varloc, removeVarVal(cs.vars.locval, cs.vars.varloc(var), val))) \\ es' =& es \end{aligned}$$

where

- $removeVarVal : (ULOC \nrightarrow UVAL) \times ULOC \times UVAL \rightarrow (ULOC \nrightarrow UVAL)$ is a function to remove any instance of a specific value from a local variable, such that
$$removeVarVal(lv, loc, val) = \{(l, v) \in lv \mid l \neq loc \vee v \neq val\}$$

The new set of mappings is the same as the old set, less any that have the same location as the incoming local variable and the same value.

- $var \in UVAR$ is the local variable to be modified

$$var = rvva.action.attrs(\text{“variable”})$$

- $val \in UVAL$ is the value to be removed from the local variable

$$val = rvva.action.inPins(\text{“value”})$$

5.2.7 Example 6: Start Classifier Behaviour and Read Self

The activities shown in Figure 5.9 demonstrate two actions. The StartClassifierBehaviorAction is used to initiate the behaviour associated with *Car1*, an instance of the *Car* class. The ReadSelfAction is executed once that classifier behaviour is started, and simply retrieves the instance that owns the activity. As can be seen from the partial output in Figure B.29 on page 299, the instance is *Car1*, as expected.

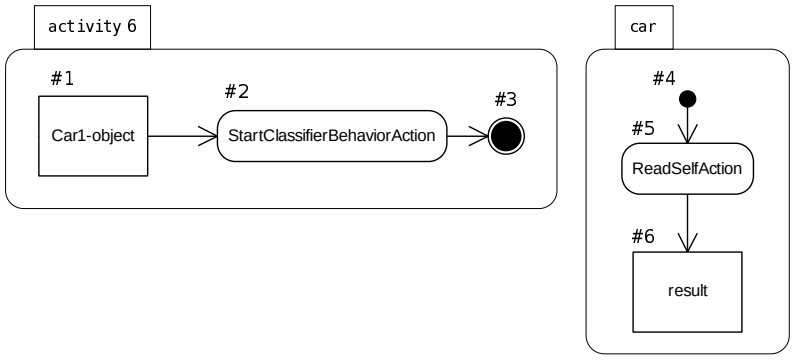


Figure 5.9: Example activity to execute StartClassifierBehaviorAction and ReadSelfAction

See Section **??** for the ADLF representation of this activity, the input file(s) required for the execution, as well as sample output from the interpreter.

5.2.7.1 StartClassifierBehaviorAction

The StartClassifierBehaviorAction is described as "an action that starts the classifier behaviour of the input" [59, §11.3.46]. This action "provides a way to indicate when the classifier behavior of a newly created object should begin to execute" [59, 11.1]. A *classifier behaviour* is a "definition of behavior...it describes the sequence of state changes an instance of a classifier may undergo in the course of its lifetime" [59, 13.3.2].

For simplicity, our interpretation of classifier behaviour differs slightly from that described by the UML specification:

- The definition of classifier behaviour above implies the use of state machines to define it. This may be an over-specification in the UML specification [59]; we use an activity to define the behaviour.

- Typically, classifier behaviour is associated with active objects, and the behaviour is executed as soon as an active object is created. Likewise, as soon as the behaviour is complete, the object is terminated [59, 13.3.8]. The scope of our research does not include the distinction between active and passive objects. We permit the association of classifier behaviour with any class; this behaviour may or may not be instantiated. Instantiation must be accomplished with a StartClassifierBehaviorAction. When the classifier behaviour has completed execution, the object does not terminate. In this way, we could use classifier behaviour to contain instructions that would typically be included in a class constructor. Alternatively, we could create an activity that simply waits for triggers, responding as required. This latter type of activity would be appropriate for an object waiting for method calls.

Interestingly enough, this action is not considered an *invocation* action. This topic is discussed further in Section 5.4.2.

The metamodel for this action is shown in Figure A.22 on page 283. The action has one input pin, object, which holds the object whose behaviour is to be started. The action has no other attributes, associations, or output pins.

There are two activities in our example in Figure 5.9. Only the 'activity6' activity is considered when the interpreter starts execution. As the StartClassifierBehaviorAction in node #2 is executed, the second activity is incorporated into the control store and becomes available for execution.

Let *scba* be an action node representing an instance of StartClassifierBehaviorAction. The effect of executing *scba* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *scba* is enabled.

Post-conditions The execution of *scba* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= addAD(cs, newad, ds) \\ es' &= es \end{aligned}$$

where

- $newad \in ActivityDiagram$ is the activity representing the classifier behaviour to be started

  $$newad = classBeh(classOf(scba.action.inPins(\text{“object”})))$$

  and

  - $classBeh : ClassName \nrightarrow ActivityDiagram$ is a function that retrieves the

activity representing a classifier behaviour, such that

$$classBeh(c) = \begin{cases} ad \in ActivityDiagram & \text{if activity } ad \text{ has been} \\ & \text{associated with classifier } c \\ \text{undefined} & \text{otherwise} \end{cases}$$

– $classOf : UOID \rightarrow UCLASS$ is a function that retrieves the classifier of an object identifier (see page 90).

- $addAD : ControlStore \times ActivityDiagram \times DataStore \rightarrow ControlStore$ is a function that adds a new activity to the set of activities currently executing, such that

$$\begin{aligned} addAD((ads, pc, thr, vars), newad, ds) = &(ads \cup newad, \\ &addPC(pc, newad), \\ &addThread(thr, newad), \\ &addVar(vars, newad, ds)) \end{aligned}$$

The control store is modified in that the new activity is added to the current set of activities and the program counters, threads, and local variables are all updated.

and

– $addPC : \mathcal{P}\,UPC \times ActivityDiagram \rightarrow \mathcal{P}\,UPC$ is a function that adds new program counters (tokens) to the control store, based on the initial nodes of the new activity

$$addPC(pc, ad) = pc \cup T$$

where

$$T \subseteq Token$$

and

$$\forall n \in initialNodes(ad) \mid \exists! t \in T \mid t.sittingOn = n$$

The new set of program counters is like the old set, except that there is now exactly one new token for each initial node in the new activity.

– $addThread : \mathcal{P}\,UTHREAD \times ActivityDiagram \rightarrow \mathcal{P}\,UTHREAD$ is a function that adds new threads to the control store, based on the initial nodes of the new activity, such that

$$addThread(thr, ad) = thr \cup ST$$

where

$$ST \subseteq SchedulerThread$$

and

$$\forall n \in initialNodes(ad) \mid \exists! st \in ST \mid st.token.sittingOn = n$$

The new set of threads is like the old set, except that there is now exactly one new thread for each initial node in the new activity.

– $addVar : ((UVAR \rightarrow ULOC) \times (ULOC \nrightarrow UVAL)) \times ActivityDiagram \times DataStore \rightarrow ((UVAR \rightarrow ULOC) \times (ULOC \nrightarrow UVAL))$ is a function that adds new local variables to the control store, based on the new activity, such that, assuming $ad.localVars = \{v_1, ..., v_n\}$,

$$addVar((varloc, locval), ad, ds) = (varloc \oplus [v_1 = l_1][v_2 = l_2]...[v_n = l_n], locval)$$

and

$$\forall 1 \leq i \leq n.(l_i \notin dom(locval) \wedge l_i \notin dom(ds.m))$$

The new local variable mappings are like the old mappings, except that there is now a new mapping for each local variable defined in the new activity. That mapping is from a local variable to a new location, previously unused in either the local variable mappings or the data store mappings.

– $initialNodes : ActivityDiagram \rightarrow \mathcal{P}\,GraphNode$ is a function that retrieves the initial nodes of an activity (see page 68).

5.2.7.2 ReadSelfAction

Every action is considered to be part of some behaviour (e.g., defined by an activity), and every behaviour executes in the context of some object. The ReadSelfAction is described as "an action that retrieves the host object of an action" [59, 11.3.36]. This action is used to simply retrieve the instance that 'owns' the activity that the action is in.

The metamodel for this action is shown in Figure A.14 on page 279. The action has no input pins, attributes or other associations. It has one output pin, result,

which will hold the object owning the action.

There are two activities in our example in Figure 5.9. One of them, called 'car', is the activity describing the classifier behaviour of the *Car1* object. When the ReadSelfAction is executed, if returns the instance that owns the activity in which it appears, i.e., *Car1*.

Let *rsa* be an action node representing an instance of ReadSelfAction. The effect of executing *rsa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *rsa* is enabled.

Post-conditions The execution of *rsa* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= updateOutput(cs, rsa, \text{“result”}, rsa.action.owner) \\ es' &= es \end{aligned}$$

where

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).
- $rsa.action.owner \in UOID$ is the object that owns the activity containing the node (see page 70).

Notation For convenience sake, we often use an object node labelled "this" as shorthand for a ReadSelfAction, where the object node refers to the result of that action. For example, consider the activities in Figure 5.10. The solitary object node in Figure 5.10(b) is equivalent to using an explicit ReadSelfAction with its result object node, as shown in Figure 5.10(a).

(a) ReadSelfAction (b) Shorthand

Figure 5.10: An object node labeled "this" is shorthand for using a ReadSelfAction. The two examples shown are equivalent

5.3 Invocation Actions

In general, the majority of UML actions are straightforward, e.g., the CreateObjectAction creates a new object identifier, the ReadVariableAction reads a variable, etc. Invocation actions, on the other hand, can be quite complicated. There are so many actions devoted to invoking behaviour in UML that it can be rather difficult to determine exactly which action to use when modelling an activity.

Table 5.1 lists UML's invocation actions.[8] The actions are categorized according to two dimensions: whether the invocation can be considered direct or indirect, and whether the invoked behaviour can be executed synchronously or asynchronously. We consider an invocation to be direct when the target behaviour is accessed without any intermediaries, e.g., there is no message or signal to be accepted before the invoked behaviour can execute. Indirect execution relies on a call message, or a transmitted signal or object, in order to request the execution of a behaviour. In other words,

[8]Note that the StartClassifierBehaviorAction is not included in this section; it does not specialize the abstract InvocationAction class, and is therefore not considered a true invocation action, even though its effect is to "start the classifier behavior of the input" [59, §11.3.46].

there is the possibility that a behaviour invoked indirectly will not actually execute. On the other hand, behaviour invoked directly should execute, barring unforeseen circumstances. With respect to the second dimension, a behaviour that is invoked synchronously must execute and terminate before the calling behaviour[9] can complete its own execution. Note that the calling action is stalled; however, other parts of the calling activity could possibly execute can complete its own execution. When a caller invokes a behaviour asynchronously, however, the calling behaviour can continue to execute concurrently with the invoked behaviour.

Table 5.1: Invocation actions classified along two dimensions: synchronous vs. asynchronous and direct vs. indirect

Invocation	**Asynchronous**	**Synchronous**
Direct	CallBehaviorAction isSynchronous = false (Section 5.3.1)	CallBehaviorAction isSynchronous = true (Section 5.3.1)
Indirect	CallOperationAction isSynchronous = false (Section 5.3.2) SendSignalAction (Section 5.3.3) SendObjectAction (Section 5.3.3) BroadcastSignalAction (Section 5.3.3)	CallOperationAction isSynchronous = true (Section 5.3.2)

[9]The calling behaviour is an activity, where different actions may potentially execute concurrently. In the case of a synchronous call action, that particular calling action is stalled until the called behaviour finishes execution. However, other actions in the calling behaviour may continue to execute.

5.3.1 Example 7: Call Behaviour

The activities in Figure 5.11 demonstrate the CallBehaviorAction, which is used to invoke behaviour directly.

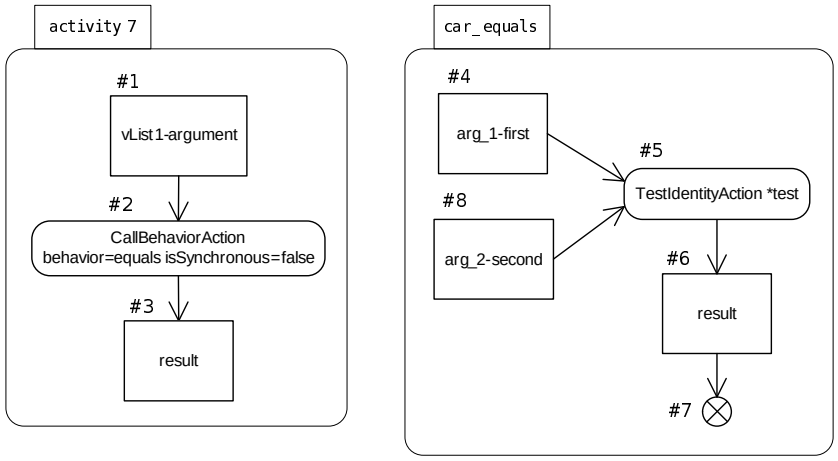


Figure 5.11: Example activities to execute CallBehaviorAction asynchronously

5.3.1.1 CallBehaviorAction

The CallBehaviorAction is described as "a call action that invokes a behavior directly" [59, §11.3.9]. For our purposes, that means that no event or message is required in order to request the invocation of behaviour; instead, a behaviour is directly invoked.

The metamodel for this action is shown in Figure A.6 on page 276. The action may have multiple argument input pins. We have amalgamated these input pins into one argument pin, which contains a (possibly empty) collection. The action also has an association, behavior of type Behavior, which represents the invoked behaviour. In addition, there is a Boolean isSynchronous attribute, which indicates how the call

should be performed. By default, all calls are synchronous; adding the annotation "isSynchronous=false" ensures that the call in our example will be performed asynchronously. Finally, the action may have multiple output pins to contain the result(s) of the operation. For simplicity, we assume a single result output pin, which could contain a collection. Note that when a CallBehaviorAction is executed asynchronously, its result will be undefined, because the "action completes immediately without a result, if the call is asynchronous" [59, §11.3.9].

There are two activities in our example. Only the 'activity7' activity is considered when the interpreter starts execution. As the CallBehaviorAction in node #2 is executed, the second activity is incorporated into the control store and becomes available for execution.

Let *cba* be an action node representing an instance of CallBehaviorAction. The effect of executing *cba* in state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions. Note that for synchronous calls, *cba* is executed twice: once in a non-waiting state, which causes the direct invocation of the called behaviour; and again in a waiting state, which causes any result to be output.

Pre-conditions The action node *cba* is enabled.

Post-conditions The execution of *cba* results in a post-state $st' = (ds', cs', es')$, such that

$$ds' = ds$$

$$cs' = \begin{cases} setWaitingCS(updateOutput(cs, cba, \text{“result”}, \\ \quad getResult(newad)), cba, false) & \text{if } waiting \\ asyncCBA(cs, newad, cba) & \text{if } \neg waiting \wedge \neg isSync \\ syncCBA(cs, newad, cba) & \text{otherwise} \end{cases}$$

$$es' = es$$

where

- $waiting \in Boolean$ indicates whether or not this invocation action has already invoked the behaviour and is now waiting for the invoked behaviour to finish execution
$$waiting = cba.action.waiting$$

- $isSync \in Boolean$ indicates whether or not this invocation is to be made synchronously
$$isSync = cba.action.attrs(\text{“isSynchronous”})$$

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).

- $getResult : ActivityDiagram \nrightarrow UVAL$ is a function that takes as input an activity (e.g., the invoked behaviour) and returns a result value. That value will be the value of the last object node labelled “result” before the invoked activity's flow final node, such that

$$getResult(ad) = \begin{cases} v \in UVAL & \text{if } v \text{ is the value of the “result” object node} \\ & \text{before } ad\text{'s activity final node} \\ \text{undefined} & \text{otherwise} \end{cases}$$

  We insist that any activity representing invoked behaviour must have a final node, which effectively represents a return statement. If there is no “result” node, then the result is undefined.

- $newad \in ActivityDiagram$ is the activity representing the behaviour to be executed
$$newad = cba.action.attrs(\text{“behavior”})$$

- $setWaitingCS : ControlStore \times ActionNode \times Boolean \rightarrow ControlStore$ is a function that updates the control store by setting the waiting status of one specific action node, such that

$$\begin{aligned} setWaitingCS(cs, an, b) = &(setWaitingAds(cs.ads, an, b), \\ &cs.pc, cs.thr, cs.localVars) \end{aligned}$$

  and

- $setWaitingAds : \mathcal{P}\,ActivityDiagram \times ActionNode \times Boolean \rightarrow \mathcal{P}\,ActivityDiagram$ is a function that updates the *waiting* status of one specific action node among all executing activities, such that

$$setWaitingAds(ads, an, b) = \{setWaitingAd(ad, an, b) \mid ad \in ads\}$$

- $setWaitingAd : ActivityDiagram \times ActionNode \times Boolean \rightarrow ActivityDiagram$ is a function that updates the *waiting* status of one specific action node in a given activity, such that

$$setWaitingAd(ad, an, b) = \begin{cases} ad & \text{if } an \notin ad.nodes \\ (ad.id, & \text{otherwise} \\ ad.nodes \setminus an & \\ \cup setWaitingNode(an, b), & \\ ad.localVars) & \end{cases}$$

  The new diagram is like the old diagram, except that the action node is updated.

- $setWaitingNode : ActionNode \times Boolean \rightarrow ActionNode$ is a function to update the *waiting* status of one specific node, such that

$$\begin{aligned} setWaitingNode(an, b) = &(an.id, an.in, an.out, \\ &an.annot, b, an.owner, an.stalled) \end{aligned}$$

  The new action node is like the old node, except that its waiting term has been updated.

- $asyncCBA : ControlStore \times ActivityDiagram \times ActionNode \rightarrow ControlStore$ is a function that updates the control store by invoking a behaviour asynchronously, such that

$$asyncCBA(cs, newad, an) = addAD(cs, assignArgs(newad, an))$$

  The asynchronous CallBehaviorAction *is similar to the* StartClassifierBehaviorAction *in that it can bring a new activity into the set of activities being executed. In addition, this action assigns argument values to nodes in the new activity. Finally, because this invocation is asynchronous, there is no requirement for the action to wait for a response.*

  and

- $addAD : ControlStore \times ActivityDiagram \rightarrow ControlStore$ is a function that adds a new activity to the set of activities currently execution (see page 119).
- $assignArgs : ActivityDiagram \times ActionNode \rightarrow ActivityDiagram$ is a function that assigns argument values (from an action node's input pin) to specific nodes in the target activity, such that

$$\begin{aligned} &assignArgs(ad, an) = \\ &\quad (ad.id, \\ &\quad \{assignArg(gn, an.action.inPins(\text{“argument”})) \mid gn \in ad.nodes\}, \\ &\quad ad.localVars) \end{aligned}$$

  The new activity is just like the old activity, except that argument values have been assigned.
- $assignArg : GraphNode \times List(UVAL) \rightarrow GraphNode$ is a function to assign an argument (if required) to individual graph nodes, such that

$$assignArg(gn, (v_1, ..., v_n)) = \begin{cases} assign(gn, v_i) & \text{if } gn \in ObjectNode \\ & \wedge argName(gn) = \text{“arg”}\&i \\ assign(gn, (v_1, ..., v_n)) & \text{if } gn \in ObjectNode \\ & \wedge argName(gn) = \text{“args”} \\ gn & \text{otherwise} \end{cases}$$

  The new node is like the old node, except that argument nodes have been updated.
- $assign : ObjectNode \times UVAL \rightarrow ObjectNode$ is a function that assigns a specific value to an object node, such that

$$assign(on, val) = (on.id, on.in, on.out, argName(on), val, on.stalled)$$

  The new object node is the same as the old object node, except that a specific value has been assigned.
- $argName : ObjectNode \nrightarrow String$ is a function that retrieves the name of the argument, if any, that the object node represents (see page 75).

- $syncCBA : ControlStore \times ActivityDiagram \times ActionNode \rightarrow ControlStore$ is a function that updates the control store by invoking a behaviour synchronously, such that

$$syncCBA(cs, newad, an) = setWaitingCS(asyncCBA(cs, newad, an), an, true)$$

The synchronous CallBehaviorAction is similar to the asynchronous version, except that the current call action's status must be set to 'waiting'.

5.3.2 Example 8: Call Operation

The CallOperationAction is used to invoke behaviour indirectly, for instance, sending a call message to an object, requesting the invocation of one of that object's methods. Like all indirect invocation actions, the CallOperationAction must be paired with an accept action on the receiving end; otherwise, the behaviour would never be invoked.

5.3.2.1 Aside: Accept Actions

The AcceptEventAction waits for the "occurrence of an event" [59, §11.3.2]. It is meant to handle "asynchronous messages, including asynchronous calls" [59, §11.3.2]; it cannot be used with synchronous calls. The AcceptCallAction specializes this action and represents the "receipt of a synchronous call request" [59, §11.3.1]. Based on these descriptions, one might infer that the AcceptCallAction should be used with synchronous invocations, and the AcceptEventAction for asynchronous invocations with the CallOperationAction.

Technically, we could use the AcceptEventAction for certain asynchronous CallOperationAction invocations. However, for several reasons, we have chosen to use the AcceptCallAction for all calls:

- According to the UML specification, the AcceptEventAction cannot be used to pass arguments into the called behaviour. Specifically, "there are no output pins if the trigger events...are only CallEvents when this action is an instance of AcceptEventAction" [59, §11.3.2]. This means that an asynchronous call to an

operation requiring arguments could never be performed with an AcceptEventAction.

- When a call is performed, an invocation is made to *another* object's behaviour. When the target behaviour is defined (e.g., when the activity is modelled), the user does not necessarily know whether or not that behaviour will be invoked synchronously or asynchronously. Section 8.5 shows two activities representing invoked behaviour; one of which can only be invoked asynchronously. By using only AcceptCallActions, we can create one implementation of a behaviour, which can be used regardless of the type of call.

- There is nothing in the description of the AcceptCallAction that forbids using it to receive asynchronous calls. The AcceptCallAction must be paired with a ReplyAction in order to return a result to the caller. However, if receiving an asynchronous call, "execution of the subsequent reply action will complete immediately with no effect" [59, §11.3.1]. This implies that the invoked behaviour can be executed, but there will be no result returned to the caller.

For these reasons, we always pair a CallOperationAction with an AcceptCallAction, whether or not the call is synchronous. The AcceptEventAction is used to accept asynchronous signals, e.g., from the SendSignalAction.

The activities in Figure 5.12 demonstrate the execution of a synchronous CallOperationAction. The 'getYear' activity represents the called behaviour; note the required AcceptCallAction and ReplyAction. This pair of activities is equivalent to the

pseudocode statement `Car1.getYear(voidList)`, where `voidList` is an empty list, and the result of the call will be captured by the 'result' node `#3`.

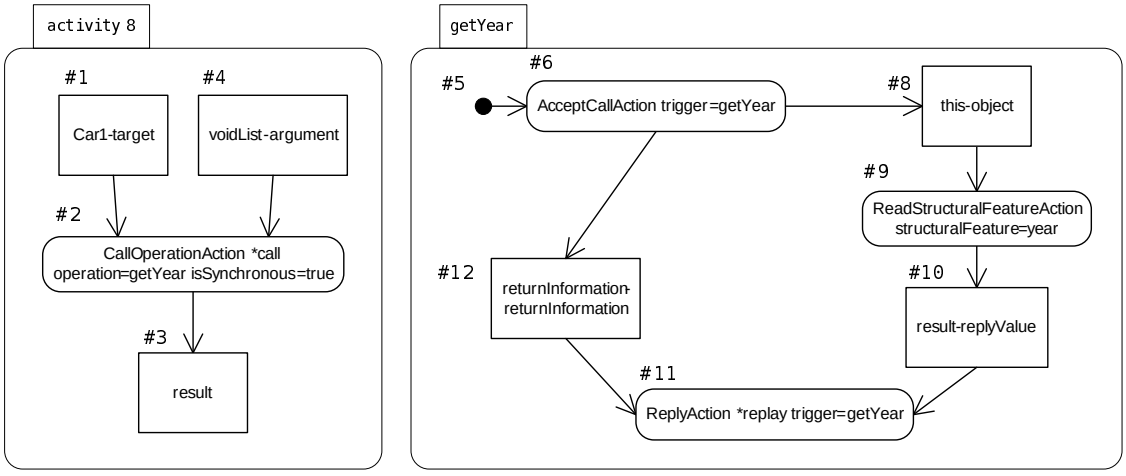


Figure 5.12: Example activity to execute synchronous CallOperationAction, AcceptCallAction and ReplyAction

5.3.2.2 CallOperationAction

The CallOperationAction is described as "an action that transmits an operation call request to the target object, where it may cause the invocation of associated behavior" [59, §11.3.10]. For our purposes, a call operation action causes a call message to be created and sent to the target object.

The metamodel for this action is shown in Figure A.7 on page 277. The action may have multiple argument input pins. For simplicity, we have amalgamated these input pins into one argument pin, which contains a (possibly empty) collection. In addition to the incoming arguments, the action also has a target input pin, which is the target object for the invocation. The action also has an association, operation

of type Operation, which represents the invoked behaviour. In addition, there is a Boolean isSynchronous attribute, which indicates how the call should be performed. By default, all calls are synchronous; adding the annotation "isSynchronous=false" would ensure that the call would be performed asynchronously. Finally, the action may have multiple output pins to contain the result(s) of the operation. Again, for simplicity, we assume a single result output pin, which could contain a collection. Note that when a CallOperationAction is executed asynchronously, its result will be undefined, because the "any return or out values from the invoked operation are not passed back" [59, §11.3.10].

Aside: Synchronous vs. Asynchronous Calls If a CallOperationAction is performed synchronously, as in our example in Figure 5.12, the action's execution is considered complete when a reply transmission is received [59, §11.3.10]. If the action is performed asynchronously, the action is considered complete when "the invocation of the operation is established" [59, §11.3.10]. Any return values from an asynchronous call are not returned to the caller. We consider an asynchronous CallOperationAction to be complete when the call message has been sent to the target's event buffer.

There are two activities in our example in Figure 5.12. Only the 'activity8' activity is considered when the interpreter starts execution. As the CallOperationAction in node #2 is executed, the second activity is incorporated into the control store and becomes available for execution.

It is important to note that the CallOperationAction must be paired by an accept action in the target. Otherwise, a call could be made, but it would never be received

and would therefore never trigger the execution of the called behaviour. Similarly, the AcceptCallAction must be paired with a ReplyAction in order to return a result to a caller.

Let *coa* be an action node representing an instance of CallOperationAction. The effect of executing *coa* in state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions. Note that for synchronous calls, *coa* is executed twice: once in a non-waiting state, which causes the call to be sent; and again in a waiting state, which causes any result to be output.

Pre-conditions The action node *coa* is enabled.

Post-conditions The execution of *coa* results in a post-state $st' = (ds', cs', es')$, such that

$$ds' = ds$$

$$cs' = \begin{cases} setWaitingCS(updateOutput(cs, coa, \text{“result”}, & \text{if } waiting \\ \quad getBufRes(es(coa.action.owner), coa)), coa, false) & \\ asyncCOA(cs, newad) & \text{if } \neg waiting \wedge \neg isSync \\ syncCOA(cs, newad, coa) & \text{otherwise} \end{cases}$$

$$es' = \begin{cases} es & \text{if } waiting \\ addReceive(addSend(es, coa.action.owner, cm), & \text{otherwise} \\ \quad coa.action.inPins(\text{“target”}), cm) & \end{cases}$$

where

- $waiting \in Boolean$ indicates whether or not this invocation action has already invoked the behaviour and is now waiting for the invoked behaviour to finish execution

$$waiting = coa.action.waiting$$

- $isSync \in Boolean$ indicates whether or not this invocation is to be made synchronously

$$isSync = coa.action.attrs(\text{“isSynchronous”})$$

- $updateOutput : ControlStore \times ActionNode \times String \times UVAL \rightarrow ControlStore$ is a function that sets the value of an output pin in the control store (see page 87).
- $getBufRes : Buffer(UEVENT) \times ActionNode \nrightarrow UVAL$ is a function that retrieves the return value of a call (if such a return has been made) from an object's event buffer, such that

$$getBufRes(buffer, an) = \begin{cases} msgOf(e).result & \text{if } \exists e \in UEVENT. \\ & (inBuffer(buffer, e) \\ & \wedge msgOf(e) \in Returns \\ & \wedge msgOf(e).op.name = \\ & an.action.attrs(\text{“operation”}) \\ & \wedge msgOf(e).thread \in threads(an)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

  The return value is contained in a return message, associated with an event in the action node's owner's buffer. The specific message has an operation name that match the action node's operation, and they both refer to the same thread.

  and

  - $inBuffer : Buffer(UEVENT) \times UEVENT \rightarrow Boolean$ is a function that indicates whether or not a given event is in a buffer (see page 60).
  - $threads : GraphNode \rightarrow \mathcal{P}\, UTHREAD$ is a function that returns the threads associated with a particular node in the activity, such that

$$threads(gn) = \{th \in UTHREAD \mid th.token.sittingOn = gn\}$$

- $setWaitingCS : ControlStore \times ActionNode \times Boolean \rightarrow ControlStore$ is a function to update the control store by setting the waiting status of one specific action node (see page 126).
- $asyncCOA : ControlStore \times ActivityDiagram \rightarrow ControlStore$ is a function that updates the control store by invoking a behaviour asynchronously, such that

$$asyncCOA(cs, newad) = addAD(cs, newad)$$

  The asynchronous CallOperationAction *is similar to the* StartClassifierBehaviorAction *and asynchronous* CallBehaviorAction *in that it brings a new activity into*

the set of activities currently being executed. Unlike the CallBehaviorAction *however, this action does not assign values to arguments in the new activity; this is the role of the matching accept action. Finally, because this invocation is asynchronous, there is no requirement for the action to wait for a response.*

- $newad \in ActivityDiagram$ is the activity representing the behaviour to be executed

$$newad = coa.action.attrs(\text{“operation”}).implementation$$

An operation's implementation is defined by a separate activity.

- $syncCOA : ControlStore \times ActivityDiagram \times ActionNode \rightarrow ControlStore$ is a function that updates the control store by invoking a behaviour synchronously, such that

$$syncCOA(cs, newad, an) = setWaitingCS(asyncCOA(cs, newad), an, true)$$

The synchronous CallOperationAction *is similar to the asynchronous version, except that the current call action's status must be set to 'waiting' through the* $setWaiting()$ *function.*

and

- $addSend : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a send event (associated with a particular message) to an object's event buffer, such that

$$addSend(es, oid, m) = es \oplus [oid = addEvent(es(oid), SendEvent(m))]$$

The new event store is like the old event store, except that we update the oid's buffer.

and

  - $addEvent : Buffer(UEVENT) \times UEVENT \rightarrow Buffer(UEVENT)$ is a function that adds an event to an existing buffer (see page 59).
  - $SendEvent : UMESSAGE \rightarrow UEVENT$ is a function that maps a message to its associated send event (see page 58).

- $cm \in Calls$ is the call message associated with a specific call operation action, such that

$$\begin{aligned} & cm.receiver = coa.action.inPins(\text{“target”}) \\ & \wedge cm.op = coa.action.attrs(\text{“operation”}) \\ & \wedge cm.params = coa.action.inPins(\text{“argument”}) \\ & \wedge cm.sender = coa.action.owner \\ & \wedge cm.thread \in \{th \in UTHREAD \mid th.token.sittingOn = coa\} \end{aligned}$$

- $addReceive : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a receive event (associated with a particular message) to an object's event buffer, such that

$$addReceive(es, oid, m) = es \oplus [oid = addEvent(es(oid), ReceiveEvent(m))]$$

The new event store is like the old event store, except that we update the oid's buffer.

and

  - $ReceiveEvent : UMESSAGE \rightarrow UEVENT$ is a function that maps a message to its associated receive event (see page 58).

Note that we are assuming lossless communication in that both the *SendEvent* and *ReceiveEvent* are placed in the appropriate buffers. There is also an assumption that the *ReceiveEvent* occurs after the *SendEvent*.

5.3.2.3 AcceptCallAction

The AcceptCallAction is described as “an accept event action representing the receipt of a synchronous call request” [59, §11.3.1].

The metamodel for this action is shown in Figure A.1 on page 274. The action has no input pins. It inherits one association, trigger of type Trigger, which indicates the type of events accepted by the action.[10]

[10] As discussed on page 57, we associate the UML concept of Trigger with the universe of names,

In addition, the action inherits a Boolean attribute, isUnmarshall, which indicates whether or not there is a single output pin for the event. We do not make use of this attribute, assuming instead that there are zero or more output pins for this action that would hold "received event objects or their attributes" [59, §11.3.2]. The output pins are used to pass arguments from the caller to the invoked behaviour. In our example in Figure 5.12, there are no arguments to be passed to the 'getYear' behaviour. There is one other output pin, returnInformation, which holds "sufficient information to perform a subsequent reply and return control to the caller" [59, §11.3.1].[11] In other words, the return information must include any results to be returned to the caller, as well as enough information to send a return message to the caller.

When the AcceptCallAction in node #6 in our example executes, it populates its output pins. Specifically, it reads any argument information from the call message and populates the argument pins (none in this example) so that the invoked behaviour has sufficient information. The AcceptCallAction also populates the returnInformation pin, which is used as input for the ReplyAction.

Aside: Return Information As mentioned above, the AcceptCallAction must be paired with a ReplyAction in order to return a result to the caller. They are connected by a returnInformation pin. This pin contains "a value containing sufficient information to perform a subsequent reply and return control to the caller" [59, §11.3.1].

UNAME, i.e., the trigger is the name of an operation that an AcceptCallAction is waiting for. In our example in Figure 5.12, the accept action is waiting for a call to the 'getYear' operation. When an event bearing a call message for that operation is found in the target object's buffer, the AcceptCallAction will be executed.

[11] Note that there is one more output pin associated with the AcceptCallAction in our example. In this case, however, the object node labeled 'this' is simply a shorthand for a ReadSelfAction as discussed on page 121.

We assume the existence of a set of values, $RI \subseteq UVAL$, that represent the return information values to be passed between the AcceptCallAction and its matching ReplyAction. A return information value $ri \in RI$ is defined as a 4-tuple

$$ri = (receiver, sender, thread, isSync)$$

where

- $receiver \in UOID$ is the target of the original call.
- $sender \in UOID$ is the sender of the original call.
- $thread \in UTHREAD$ is the thread in which the original call was created.
- $isSync \in Boolean$ indicates whether or not the original call was synchronous.

Let aca be an action node representing an instance of AcceptCallAction. The effect of executing aca in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node aca is enabled.

Post-conditions The execution of aca results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= (acaDiags(cs.ads, aca, msgOf(e)), cs.pc, cs.thr, cs.vars) \\ es' &= removeEvent(es(aca.action.owner), e) \end{aligned}$$

where

- $acaDiags : \mathcal{P}\,ActivityDiagram \times ActionNode \times Calls \to \mathcal{P}\,ActivityDiagram$ is a function that uses information from an accept call action to set argument values in the set of all current activities, such that

$$acaDiags(ads, an, m) = \{acaDiag(ad, an, m) \mid ad \in ads\}$$

and

  - $acaDiag : ActivityDiagram \times ActionNode \times Calls \to ActivityDiagram$ is a function that uses information from an accept call action to set argument values in an activity, such that

$$acaDiag(ad, an, m) = \begin{cases} ad & \text{if } an \notin ad.nodes \\ (ad.id, & \text{otherwise} \\ \quad ad.nodes \setminus an.out & \\ \quad \cup acaNodes(an.out, m), & \\ \quad ad.localVars) & \end{cases}$$

    The new activity is like the old activity, except that the action node's output pins are updated.

  - $acaNodes : \mathcal{P}\,GraphNode \times Calls \to \mathcal{P}\,GraphNode$ is a function that uses information from an accept call action to set argument values of an action node's targets, such that

$$acaNodes(gns, m) = \{acaNode(gn, m.params) \mid gn \in gns\}$$

    The new set of nodes is like the old set, except that their values have been updated.

  - $acaNode : GraphNode \times List(UVAL) \to GraphNode$ is a function that uses information from an accept call action to set an argument value in one target node, such that

$$acaNode(gn, (v_1, ..., v_n)) = \begin{cases} assign(gn, v_i) & \text{if } gn \in ObjectNode \\ & \quad \wedge argName(gn) = \text{“arg”}\&i \\ assign(gn, & \text{if } gn \in ObjectNode \\ \quad (v_1, ..., v_n)) & \quad \wedge argName(gn) = \text{“args”} \\ assign(gn, ri) & \text{if } gn \in ObjectNode \\ & \quad \wedge argName(gn) = \\ & \text{“returnInformation”} \\ gn & \text{otherwise} \end{cases}$$

The new node is like the old node, except that its value has been updated.

- $assign : ObjectNode \times UVAL \rightarrow ObjectNode$ is a function that assigns a specific value to an object node (see page 128).
- $argName : ObjectNode \nrightarrow String$ is a function that retrieves the name of the argument, if any, that the object node represents (see page 75).
- $ri \in UVAL$ is a value representing the return information required by the ReplyAction matched with this AcceptCallAction

  $$ri = (msgOf(e).sender, msgOf(e).op, msgOf(e).receiver, msgOf(e).thread)$$

  Note that the return information value contains most of the information in the call message to which this AcceptCallAction is responding.

- $msgOf : UEVENT \rightarrow UMESSAGE$ is a function that maps an event to its associated message (see page 59).
- $e \in UEVENT$ is a receive event in this action node's owner's buffer, whose call message operation matches the trigger that this action node is waiting for

  $$e = getTriggerE(es, aca)$$

  and

  - $getTriggerE : EventStore \times ActionNode \nrightarrow UEVENT$ is a function that retrieves a receive event from an action node's owner's buffer, such that

$$getTriggerE(es, an) = \begin{cases} ev \in & \text{if } ev \in Receives \\ \quad es(an.action.owner) & \quad \wedge msgOf(ev) \in Calls \\ & \quad \wedge msgOf(ev).op.name = \\ & \quad an.action.inPins(\text{“trigger”}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

The trigger event is associated with a call message whose operation matches the action node's trigger.

- *removeEvent* : $Buffer(UEVENT) \times UEVENT \rightarrow Buffer(UEVENT)$ is a function to remove a specific event from an existing buffer (see page 59).

5.3.2.4 ReplyAction

The ReplyAction is described as an action that “returns the values to the caller of the previous call, completing execution of the call” [59, §11.3.43].

The metamodel for this action is shown in Figure A.19 on page 282. The action may have multiple input pins, containing the values to be returned to the caller. For simplicity, we assume a single replyValue input pin, which may contain a collection value. The action has another input pin, returnInformation, which contains the return information produced by the previous AcceptCallAction. The action also has an association, trigger of type Trigger,[12] that indicates the name of the operation whose call is being replied to. This action has no output pins.

For our example in Figure 5.12, the ReplyAction at node #11 receives any reply values from the object node #10. The return information, created by the AcceptCallAction, is in the object node #12, and the ReplyAction uses this information to create a reply message to be sent to the calling object. As discussed earlier, a reply

[12] As discussed on page 57, we associate the UML concept of Trigger with the universe of names, *UNAME*.

message is only sent if the invocation of this behaviour was synchronous; otherwise, the ReplyAction executes with no effect.

Let *ra* be an action node representing an instance of ReplyAction. The effect of executing *ra* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *ra* is enabled.

Post-conditions The execution of *ra* results in a post-state $st' = (ds', cs', es')$, such that

$$ds' = ds$$
$$cs' = cs$$

$$es' = \begin{cases} addReceive(addSend(es, ra.action.owner, rm), \\ \quad caller, rm) & \text{if } isSync \\ es & \text{otherwise} \end{cases}$$

where

- $addSend : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a send event (associated with a particular message) to an object's event buffer (see page 135).
- $rm \in Returns$ is the return message to be sent to the caller of this behaviour, such that

$$\begin{aligned} & rm.receiver = ri.sender \\ & \wedge rm.op.name = ra.action.attrs(\text{“replyToCall”}) \\ & \wedge rm.returner = ra.action.owner \\ & \wedge rm.thread = ri.thread \\ & \wedge rm.result = ra.inPins(\text{“replyValue”}) \end{aligned}$$

and

– $ri \in RI$ is the return information value passed to this ReplyAction by the matching AcceptCallAction

$$ri = ra.action.inPins(\text{“returnInformation”})$$

- $addReceive : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a receive event (associated with a particular message) to an object's event buffer (see page 136).
- $caller \in UOID$ is the original caller of this behaviour

$$caller = ra.action.inPins(\text{“returnInformation”}).sender$$

5.3.3 Example 9: Send Signal

The activities in Figure 5.13 demonstrate the execution of a SendSignalAction and AcceptEventAction. After the main 'activity9' activity starts, a StartClassifierBehaviorAction is executed to initiate the behaviour of the *Car1* object. Then, an 'initialize' signal is sent to the *Car1* object using the SendSignalAction. Then, the main diagram finishes execution. The second diagram represents the *Car* classifier behaviour; when started, it simply waits for an 'initialize' event to occur. When the signal is received, this activity simply clears the *year* structuralFeature of the owning object. The sending of signals is an asynchronous invocation of behaviour; there will be no value returned to the calling object.

5.3.3.1 SendSignalAction

The SendSignalAction is described as "an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause...the execution of an activity" [59, §11.3.45]. The sending of a signal is an asynchronous invocation of behaviour and the sending object continues execution. In addition, any reply

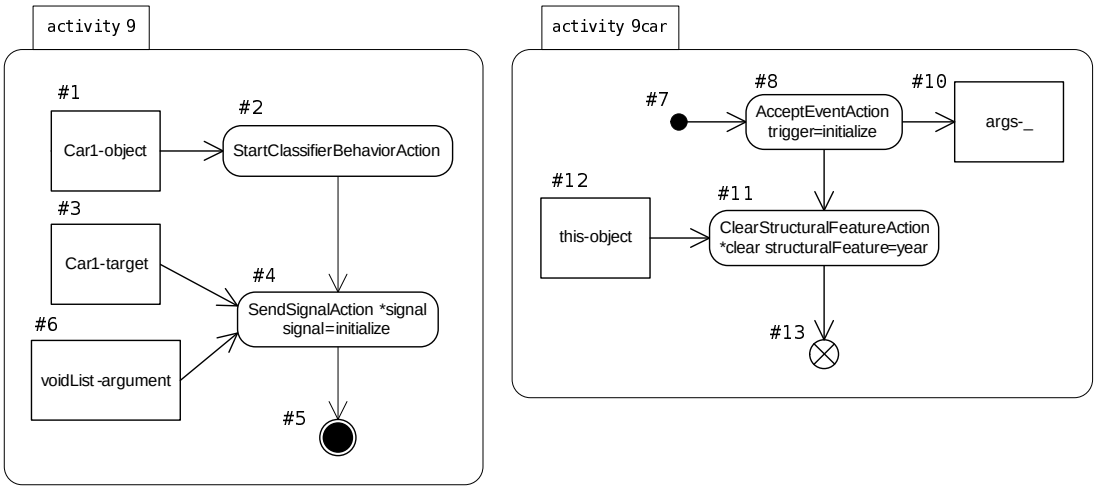


Figure 5.13: Example activities to execute SendSignalAction and AcceptEventAction

messages are ignored.

The metamodel for this action is shown in Figure A.21 on page 283. The action has zero or more argument input pins, which are used to create the signal instance. For simplicity, we use one argument pin, which holds a (possibly empty) collection value. The action has one other input pin, target, which is the target object to which the signal will be transmitted. The action also has an association, signal of type Signal,[13] which is the name (or type) of signal to be transmitted. The action has no output pins.

For our example in Figure 5.13, the SendSignalAction at node #4 will use the inputs from the argument object node #6 to create a signal instance to transmit to the target object indicated by the target object node #3. Note that the UML specification suggests the use of a special symbol to represent a SendSignalAction in an activity.

[13] A SendSignalAction must be matched with a corresponding AcceptEventAction, which is waiting for the signal. The accept action has a Trigger, which must match the signal being sent. As discussed on page 57, we associate the UML concepts of Signal and Trigger with the universe of names, *UNAME*.

We do not use this 'convex pentagon' symbol, using instead the standard rounded rectangle shape.

Note that the SendSignalAction is similar to an asynchronous CallOperationAction in that both create some type of message and transmit it to a target object. Also, like the CallOperationAction, a SendSignalAction must be matched with an accept action, specifically, an AcceptEventAction. Unlike the CallOperationAction however, no additional activities are added to the set of currently executing activities.

Let *ssa* be an action node representing an instance of SendSignalAction. The effect of executing *ssa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *ssa* is enabled.

Post-conditions The execution of *ssa* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= cs \\ es' &= addReceive(addSend(es, ssa.action.owner, signal), \\ &\qquad ssa.action.inPins(\text{“target”}), signal) \end{aligned}$$

where

- $addSend : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a send event (associated with a particular message) to an object's event buffer (see page 135).

- $signal \in Signals$ is the signal to be sent, such that

$$signal.name = ssa.action.attrs(\text{“signal”})$$
$$\wedge ssa.action.inPins(\text{“target”}) \in signal.receiver$$
$$\wedge signal.sender = ssa.action.owner$$
$$\wedge signal.params = ssa.action.inPins(\text{“argument”})$$

- $addReceive : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a receive event (associated with a particular message) to an object's event buffer (see page 136).

5.3.3.2 AcceptEventAction

The AcceptEventAction is described as “an action that waits for the occurrence of an event meeting specified condition” [59, §11.3.2].

The metamodel for this action is shown in Figure A.2 on page 274. The action has no input pins. It has one association, trigger of type Trigger, which indicates the type of events accepted by the action.[14] The action has a Boolean attribute, isUnmarshall, which indicates whether or not there is a single output pin for the event. We do not make use of this attribute, assuming instead that there are zero or one output pins for this action. The output pin, result holds “received event objects or their attributes” [59, §11.3.2]. The result output pin is used to pass arguments[15] from the caller to the invoked behaviour.

The AcceptEventAction is very similar to the AcceptCallAction except that:

- The AcceptEventAction is used only with asynchronous messages. As discussed

[14] See our discussion of signals and triggers on page 144.

[15] As discussed in Section 5.3.2.1, according to the UML specification, the AcceptEventAction cannot be used to pass arguments when receiving from a call CallOperationAction. However, it can be used for this purpose when receiving a signal from a SendSignalAction.

on page 129, we do not use the AcceptEventAction to receive asynchronous call messages. The AcceptCallAction should be used instead.

- The AcceptEventAction does assign values to its output pins, based on the arguments passed in; however, there is no requirement to provide 'return information', as performed by the AcceptCallAction.

Let *aea* be an action node representing an instance of AcceptEventAction. The effect of executing *aea* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *aea* is enabled.

Post-conditions The execution of *aea* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= (aeaDiags(cs.ads, aea, msgOf(e)), cs.pc, cs.thr, cs.vars) \\ es' &= removeEvent(es(aea.action.owner), e) \end{aligned}$$

where

- $aeaDiags : \mathcal{P}\,ActivityDiagram \times ActionNode \times Signals \rightarrow \mathcal{P}\,ActivityDiagram$ is a function that uses information from an accept event action to set argument values in the set of all current activities, such that

$$aeaDiags(ads, an, m) = \{aeaDiag(ad, an, m) \mid ad \in ads\}$$

  and

  - $aeaDiag : ActivityDiagram \times ActionNode \times Signals \rightarrow ActivityDiagram$ is

a function that uses information from an accept event action to set argument values in an activity, such that

$$aeaDiag(ad, an, m) = \begin{cases} ad & \text{if } an \notin ad.nodes \\ (ad.id, & \text{otherwise} \\ \quad ad.nodes \setminus an.out & \\ \quad \cup aeaNodes(an.out, m), & \\ \quad ad.localVars) & \end{cases}$$

The new diagram is like the old diagram, except that the action node's output pins are updated.

- $aeaNodes : \mathcal{P}\,GraphNode \times Signals \rightarrow \mathcal{P}\,GraphNode$ is a function that uses information from an accept event action to set argument values of an action node's targets, such that

$$aeaNodes(gns, m) = \{aeaNode(gn, m.arguments) \mid gn \in gns\}$$

The new set of nodes is like the old set, except that their values have been updated.

- $aeaNode : GraphNode \times List(UVAL) \rightarrow GraphNode$ is a function that uses information from an accept event action to set a value in one target node, such that

$$aeaNode(gn, (v_1, ..., v_n)) = \begin{cases} assign(gn, v_i) & \text{if } gn \in ObjectNode \\ & \quad \wedge argName(gn) = \text{“arg\_”}\&i \\ assign(gn, (v_1, ..., v_n)) & \text{if } gn \in ObjectNode \\ & \quad \wedge argName(gn) = \text{“args”} \\ gn & \text{otherwise} \end{cases}$$

The new node is like the old node, except that its value has been updated.

- $assign : ObjectNode \times UVAL \rightarrow ObjectNode$ is a function that assigns a specific value to an object node (see page 128).
- $argName : ObjectNode \nrightarrow String$ is a function that retrieves the name of the argument, if any, that this object node represents (see page 75).

- $msgOf : UEVENT \times UMESSAGE$ is a function that maps an event to its associated message (see page 59). In this case, the event is mapped to its associated signal.

- $e \in UEVENT$ is a receive event in this action node's owner's buffer, whose associated signal matches the trigger that this action node is waiting for

$$e = getSignalE(es, aea)$$

and

  - $getSignalE : EventStore \times ActionNode \nrightarrow UEVENT$ is a function that retrieves a receive event from an action node's owner's buffer; that event is associated with a signal which matches the action node's trigger, such that

$$getSignalE(es, an) = \begin{cases} ev \in & \text{if } ev \in Receives \\ \quad es(an.action.owner) & \quad \wedge msgOf(e) \in Signals \\ & \quad \wedge msgOf(ev).name = \\ & \quad an.action.inPins(\text{“trigger”}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

    As discussed on page 129, we use the AcceptEventAction to receive asynchronous signals, but not asynchronous calls. All calls are to be received with the AcceptCallAction. This restriction is reflected by our emphasis that the event e must be associated with a signal, and not a call.

- $removeEvent : Buffer(UEVENT) \times UEVENT \rightarrow Buffer(UEVENT)$ is a function to remove a specific event from an existing buffer (see page 59).

5.3.3.3 SendObjectAction

The SendObjectAction is described as "an action that transmits an object to the target object, where it may invoke behavior such as...the execution of an activity" [59, §11.3.44]. Like the SendSignalAction, this action is an asynchronous invocation of behaviour and the sending object continues execution. In addition, any reply messages are ignored.

The metamodel for this action is shown in Figure A.20 on page 282. The action has two input pins, target and request. The target input pin is the target object to which the object will be transmitted. The request input pin is the object that is to be transmitted to the target.

The SendObjectAction is similar to the SendSignalAction, except that instead of a signal created from arguments, a single object is to be sent to the target. To maintain consistency (and to avoid the need for a type of message that represents the sending of an object), we treat the two actions alike. The sole exception is that we use the incoming request object of the SendObjectAction as the contents of a signal to be sent to the target. Like the SendSignalAction, we pair the SendObjectAction with an AcceptEventAction.

Let *soa* be an action node representing an instance of SendObjectAction. The effect of executing *soa* in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node *soa* is enabled.

Post-conditions The execution of *soa* results in a post-state $st' = (ds', cs', es')$, such that

$$\begin{aligned} ds' &= ds \\ cs' &= cs \\ es' &= addReceive(addSend(es, soa.action.owner, objectSignal), \\ &\quad soa.action.inPins(\text{“target”}), objectSignal) \end{aligned}$$

where

- $addSend : EventStore \times UOID \times UMESSAGE \rightarrow$ is a function to add a send event (associated with a particular message) to an object's event buffer (see page 135).

- $objectSignal \in Signals$ is the signal to be sent, such that

$$\begin{aligned} & objectSignal.name = classOf(object) \\ & \wedge soa.action.inPins(\text{“target”}) \in objectSignal.receiver \\ & \wedge objectSignal.sender = soa.action.owner \\ & \wedge objectSignal.params = (object) \end{aligned}$$

  and

  - $classOf : UOID \rightarrow UCLASS$ is a function to retrieve the classifier of an object identifier (see page 90).
  - $object \in UOID$ is the object identifier to be sent in the signal

$$object = soa.action.inPins(\text{“request”})$$

- $addReceive : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a receive event (associated with a particular message) to an object's event buffer (see page 136).

The UML specification states “the object on the input pin may be copied during transmission, so identity might not be preserved” [59, §11.3.44]. For simplicity, we support a ‘pass by reference’ concept in that a reference to the object is transmitted, not the object itself (which would be represented in the system Model universe as an *Instance*). The UML specification does not explicitly state that a copy of the object must be transmitted, merely that identity may not be preserved. We choose to interpret this as a semantic variation point and treat the transmission of an object (or rather, its reference) in the same manner in which we treat the passage of an object through a fork node (see Section 2.4.1.2). In other words, references to objects are transmitted, rather than the objects themselves.

5.3.3.4 BroadcastSignalAction

The BroadcastSignalAction is described as "an action that transmits a signal instance to all the potential target objects in the system, which may cause...the execution of associated activities of a target object" [59, §11.3.7]. Like the SendSignalAction, this action is an asynchronous invocation of behaviour and the sending object continues execution. In addition, any reply messages are ignored.

The metamodel for this action is shown in Figure A.5 on page 276. The action inherits zero or more argument input pins, which are used to create the signal instance. For simplicity, we use one argument pin, which holds a (possibly empty) collection value. The action also has an association, signal of type Signal,[16] which is the type of signal to be transmitted. The action has no output pins.

Unlike the SendSignalAction, this action has no specific target input pin. The signal is to be transmitted "concurrently to each of the implicit broadcast target objects in the system" [59, §11.3.7]. However, the definition of these targets is left as a semantic variation point; it may be limited to some subset of all objects, or not. For simplicity, we have chosen to broadcast the signal to all objects in the data store, i.e., all currently existing objects, regardless of type.

Let bsa be an action node representing an instance of BroadcastSignalAction. The effect of executing bsa in a state $st = (ds, cs, es)$ is defined via the following pre- and post-conditions:

Pre-conditions The action node bsa is enabled.

[16] As with the SendSignalAction, this action must be matched with a corresponding AcceptEventAction, which is waiting for the signal. The accept action has a Trigger, which must match the signal being sent. As discussed on page 57, we associate the UML concepts of Signal and Trigger with the universe of names, *UNAME*.

Post-conditions The execution of *bsa* results in a post-state $st' = (ds', cs', es')$, such that

$$
\begin{aligned}
ds' &= ds \\
cs' &= cs \\
es' &= broadcastReceive(addSend(es, bsa.action.owner, signal), \\
&\qquad ds.s, signal)
\end{aligned}
$$

where

- $addSend : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a send event (associated with a particular message) to an object's event buffer (see page 135).
- $signal \in Signals$ is the signal to be broadcast, such that

$$
\begin{aligned}
&signal.name = bsa.action.attrs(\text{“signal”}) \\
&\wedge signal.receiver = ds.s \\
&\wedge signal.sender = bsa.action.owner \\
&\wedge signal.params = bsa.action.inPins(\text{“argument”})
\end{aligned}
$$

  The signal for a broadcast is similar to a standard signal, except that instead of one receiver, there are many. Every object in the current data store is a target of this broadcast signal.

- $broadcastReceive : EventStore \times \mathcal{P}\, UOID \times Signals \rightarrow EventStore$ is a function to broadcast a signal to a set of objects, such that

$$
\begin{aligned}
&broadcastReceive(es, \{o_1, o_2, ..., o_n\}, s) = \\
&\quad addReceive((...addReceive(addReceive(es, o_1, s), o_2, s)...), o_n, s)
\end{aligned}
$$

  and

  - $addReceive : EventStore \times UOID \times UMESSAGE \rightarrow EventStore$ is a function to add a receive event (associated with a particular message) to an object's event buffer (see page 136).

5.4 Pragmatics of Invocation Actions

It can be difficult to determine exactly when to use each kind of invocation action. It is possible to invoke behaviour synchronously/asynchronously and directly/indirectly. Invocation may include the passing of arguments to the target behaviour and may or may not result in a return value. To simplify the decision making process, we include the summary Table 5.2. In addition to the standard UML invocation actions, we include the StartClassifierBehaviorAction because its effect is to "start the classifier behavior of the input" [59, §11.3.46].

Each row represents an invocation action; the CallBehaviorAction and CallOperationAction appear twice, as they may be executed either synchronously or asynchronously. The '(A)Sync' column indicates whether the invocation is executed synchronously or asynchronously. The '(In)Direct' column indicates whether the behaviour is invoked directly or indirectly. The 'Args' column indicates whether arguments can be passed to the invoked behaviour. The 'Accept' column shows which accept action, if any, is used to receive the invocation. The 'Result' column indicates whether a result can be returned from the invoked behaviour. All columns described thus far contain information readily available in the UML specification [59]. The last column, 'Behaviour', pertains more to how we use activities as our chosen behaviour. This column shows whether a new activity is incorporated into the set of executing activities. For instance, the purpose of a StartClassifierBehaviorAction is to start a classifier behaviour; we take this to mean that a new activity, representing that behaviour, is incorporated into the set of executing activities. On the other hand, the send signal/object and broadcast signal actions simply put out signals.

Table 5.2: Summary of invocation actions with details about whether each invocation is synchronous/asynchronous, direct/indirect, passes arguments, received by an accept action, provides a result, or causes new behaviour to be added to the control store

Action	(A)Sync	(In)Direct	Args	Accept	Result	Behaviour
CallBehavior	sync	direct	yes	n/a	yes	yes
CallBehavior	async	direct	yes	n/a	no	yes
CallOperation	sync	indirect	yes	AcceptCall	yes	yes/no[a]
CallOperation	async	indirect	yes no	AcceptCall AcceptEvent[b]	no	yes/no[a]
SendSignal	async	indirect	yes	AcceptEvent	no	no
SendObject	async	indirect	yes[c]	AcceptEvent	no	no
BroadcastSignal	async	indirect	yes	AcceptEvent	no	no
StartClassifierBehavior[d]	async	direct	no	n/a	no	yes

[a] The CallOperationAction can introduce new behaviour (e.g., behaviour associated with a particular operation), or call on behaviour that already exists (e.g., making a call to a classifier behaviour that is already executing).

[b] If the AcceptEventAction is used to accept a CallOperationAction, no arguments can be passed to the invoked behaviour. If the AcceptCallAction is used instead, arguments can be passed.

[c] The SendObjectAction sends an object as its argument.

[d] The StartClassifierBehaviorAction is not technically an invocation action (it does not inherit from the abstract InvocationAction. However, it does cause a behaviour to execute.

Although all information necessary to differentiate the various invocation actions is presented in Table 5.2, it can be useful to discuss actions in terms of the code statements that they can represent. Table 5.3 shows eight statements representing common ways of invoking behaviour. It covers only calls, but shows different statements depending on whether the invocation is direct or indirect, whether there are arguments to pass, and whether a return result is expected. Each statement is annotated with the appropriate invocation action for representing that particular behaviour.

Table 5.3: Different code statements to invoke behaviour, depending on whether or not the invocation is direct/indirect, expects a return value, or passes arguments to the target. Each statement is matched with an appropriate UML invocation action

	Return[a]		**No Return**[b]	
	Direct	**Indirect**	**Direct**	**Indirect**
No Args	`x = get();`[c]	`x = y.get();`[d]	`reset();`[e]	`y.reset();`[f]
Args[g]	`x = add(1,2);`[c]	`x = y.add(1,2);`[d]	`set(3);`[h]	`y.set(3);`[i]

[a] The desire for a result from the invocation implies that the call will be synchronous; otherwise, any return would be lost.

[b] If a result is not required, the invocation could be synchronous, or asynchronous.

[c] CallBehaviorAction with isSynchronous=true. No accept action is required.

[d] CallOperationAction with isSynchronous=true. This call must be matched with an AcceptCallAction.

[e] CallBehaviorAction. This invocation could be used synchronously or asynchronously. No accept action is required.

[f] CallOperationAction. This invocation could be used synchronously or asynchronously. If asynchronous, it could be matched by an AcceptEventAction because there are no parameters being passed. If synchronous, it must be matched by an AcceptCallAction.

[g] The passing of arguments only affects the CallOperationAction, which can be received by either an AcceptCallAction (for synchronous or asynchronous) or an AcceptEventAction (asynchronous only). However, if the AcceptEventAction is used, no arguments can be passed to the target behaviour.

[h] CallBehaviorAction. This invocation could be used synchronously or asynchronously. No accept action is required.

[i] CallOperationAction. This invocation could be used synchronously or asynchronously. Either way, it must be matched by an AcceptCallAction because arguments are being passed.

5.4.1 Streamlining Invocation Actions

UML supports many different ways of invoking behaviour. Table 5.4 shows the various invocation actions again, this time including the StartClassifierBehaviorAction. As can be seen by this version of the table, there are multiple ways of invoking asynchronous behaviour, either directly or indirectly.

Table 5.4: Invocation actions (including the StartClassifierBehaviorAction) classified along two dimensions: synchronous vs. asynchronous and direct vs. indirect

Invocation	Asynchronous	Synchronous
Direct	CallBehaviorAction isSynchronous = false (Section 5.3.1) StartClassifierBehaviorAction	CallBehaviorAction isSynchronous = true (Section 5.3.1)
Indirect	CallOperationAction isSynchronous = false (Section 5.3.2) SendSignalAction (Section 5.3.3) SendObjectAction (Section 5.3.3) BroadcastSignalAction (Section 5.3.3)	CallOperationAction isSynchronous = true (Section 5.3.2)

UML, by definition, is a general modelling language, and thus needs to support various methods of invocation. For instance, the CallOperationAction can be used in an object-oriented model. Specifically, "objects respond to messages that are generated by objects executing communication actions" [59, §6.3.3]. On the other hand, UML also supports a "purely 'procedural' or 'process' model", i.e., "behaviors invoking each other and passing information to each other through arguments to parameters of the invoked behavior" [59, §6.3.3] through the use of the CallBehaviorAction.

That said, if it were ever desirable to streamline the UML specification, it would be possible to remove, or ignore, certain invocation actions as 'syntactic sugar'. For instance:

- The StartClassifierBehavior is used to start the classifier behavior associated with

some object. It would be possible to mimic this functionality with an asynchronous CallBehaviorAction to some 'default' operation. The initial definition of behaviour would be slightly different, but the execution would result in the same effect.

- The SendObjectAction is simply a special case of the SendSignalAction. In fact, if the input to a SendSignalAction is an object, the SendObjectAction is to be used [59, §11.3.45]. It would be possible to streamline the invocation actions by removing the SendObjectAction and just using the SendSignalAction.

- Likewise, the BroadcastSignalAction is simply an iteration of the SendSignalAction, i.e., the transmission of a signal to all potential target objects [59, §11.3.7]. Although using this action is (visually) more efficient than using several SendSignalActions, it could be removed as syntactic sugar.

5.4.2 Invocation Actions and Stacks

There are five different invocation actions, not counting the StartClassifierBehaviorAction. Only one of them, the CallOperationAction, relies on sending a call to a target object and potentially receiving a response. This is the only action where using a stack frame, as described in Chapter 3, makes any sense. The rest of the invocation actions do not use calls (with subsequent returns). Therefore, the majority of UML's invocation actions do not map particularly well to the concept of stacks and stack frames as described by the System Model.

As discussed in Section 4.1, the System Model states that a thread is considered active if its stack is not empty. However, the majority of the invocation actions will

never have anything in their thread's stacks. It is for this reason that we ignore the System Model definition of active threads.

5.5 Summary of Changes to State

Table 5.5 lists the actions that we have formally defined and indicates (with Δ) those parts of the state that change as each action is executed. We have also indicated exactly which parts of the control store are affected. Those parts of the state that are merely read during an action's execution are marked with ✓.

Table 5.5: Summary of state changes caused by the execution of individual UML actions. Entries marked with ✓indicate that information is read only. Entries marked with Δ indicate that information is written

Action	*ds*	*cs*				*es*
		ads set	pins in activity	*pc/th* sets	*vars* maps	
AcceptCallAction			Δ			Δ
AcceptEventAction			Δ			Δ
AddStructuralFeatureValueAction	Δ					
AddVariableValueAction					Δ	
BroadcastSignalAction					✓	Δ
CallBehaviorAction		Δ	Δ[a]	Δ	Δ	
CallOperationAction		Δ	Δ[a]	Δ	Δ	Δ
ClearStructuralFeatureAction	Δ					
ClearVariableAction					Δ	
CreateObjectAction	Δ		Δ			Δ[b]
DestroyObjectAction	Δ					Δ[c]
ReadExtentAction	✓		Δ			
ReadIsClassifiedObjectAction			Δ			
ReadSelfAction		✓	Δ			
ReadStructuralFeatureAction	✓		Δ			
ReadVariableAction			Δ		✓	
RemoveStructuralFeatureValueAction	Δ					
RemoveVariableValueAction					Δ	
ReplyAction						Δ
SendObjectAction					✓	Δ
SendSignalAction					✓	Δ
StartClassifierBehaviorAction		Δ		Δ	Δ	
TestIdentityAction			Δ			
ValueSpecificationAction			Δ			

[a] A result is only promulgated to the call's output pin when the call is performed synchronously. This actually results in two visits to the action: one to invoke the behaviour and one to handle the result.

[b] When a new object is created, a new mapping from it to a new event buffer is added to the event store.

[c] When an object is destroyed, the mapping from it to its event buffer is removed from the event store.

This summary table brings to light several observations about actions:

State As each action executes, it affects the underlying System Model state, i.e., the data store, control store and event store. Although it is possible for an action to modify the control store's program counters and threads, this only occurs when a new activity is being brought into the set of currently executing activities. In other words, individual actions do not affect the program counter of the control store. Advancing the program counter (by passing tokens through the activity) is handled at the activity level, as discussed in Chapter 6.

Computation UML actions are considered the "fundamental unit of behavior specification" [59, §11.1]. For the most part, these actions can be considered primitive; they are defined to perform computation or access memory, but not both [59, §11.1]. Of the actions that we have formally defined, only the TestIdentityAction and ValueSpecificationAction actually perform what could be considered a 'computation'. The TestIdentityAction compares two incoming values to determine if they are equal. We use the ValueSpecificationAction to read a value from the System Model universe, but it is designed to be able to evaluate a value specification. Other UML actions that could be considered computations are: UnmarshallAction (reduce an object to its component parts), ReduceAction (perform pair-wise combination of arguments in order to reach one result), and perhaps the OpaqueAction (perform some opaque behaviour, which could consist of anything).

Output All 'read' actions, and several others, affect the pins in a particular activity, i.e., results are being written to pins. All 'read' actions, as well as the call actions, have result pins. In addition, the accept actions write to pins in that they can fill their output pins with arguments passed in from the invocation actions. The

CreateObjectAction places the newly created object identifier on its result pin. The TestIdentityAction and ValueSpecificationAction also put their results on output pins.

Cohesion In general, most actions are 'cohesive' in that they only modify one part of the state. For instance, structural feature actions only modify the data store, variable actions only modify the *vars* part of the control store, etc. The more parts of the state that are modified by an action, the more complex the action is, and the easier it is to misinterpret its behavioural semantics. As can be seen from Table 5.5, the most complex, least-cohesive actions are the invocation actions CallOperationaction (modifies five elements) and CallBehaviorAction (modifies four elements). The complexity of these actions is supported by the sheer length of their formalizations; each runs to several pages.

Control Store As discussed in the previous section, several invocation actions affect the set of currently executing activities, such as the CallBehaviorAction, CallOperationAction and StartClassifierBehaviorAction. By adding to the set of activities, these actions also affect the set of current tokens (program counters), the set of current threads, and the current variable maps.

Context Some actions are not 'self-contained' and require information about their context in order to execute. For instance, the ReadSelfAction returns the object identifier of the object that owns the activity in which the action is found. To do this, the action needs to navigate to its surrounding activity diagram, and thence to the owning object of the diagram. Similarly, the SendSignalAction, SendObjectAction and BroadcastSignalAction all use the owner object of their containing activity to

create the signals to be sent.

Multi-Step Finally, all but two UML actions are considered 'single-step'. We refer to a 'single-step' action as one whose execution causes changes to the underlying state, and then the action need not be revisited. In contrast, a 'multi-step' action must be visited twice in order to fully complete its execution. For instance, when either the CallOperationAction or CallBehaviorAction is used synchronously, the execution of the calling action is not considered complete until the called behaviour has completed its execution. It is, however, not the case that all invocation actions are multi-step. Either of the call actions can be executed asynchronously, which means that the invoked behaviour is added to the current set of activities, the program counters, threads and variables are updated, and then the call is considered complete. Similarly, the send/broadcast actions never wait for a reply.

Chapter 6

Execution of Activities

Prior to UML 2, activities were described as a special form of state machine, albeit with a slightly different notation. This similarity changed with UML 2, when activities were integrated with actions and made into true flow diagrams, with a Petri-like *token semantics*.

An activity is a "graph of nodes and flows" [67]. The nodes are action nodes, object nodes and control nodes. The flows can be either control or data flows. In addition to the UML specification [59] and the UML Reference Manual [67], Conrad Bock has authored a series of detailed articles [2, 3, 4, 5, 6, 7] explaining the execution of actions and activities. These references discuss the execution of activities in great detail.

In this chapter, we discuss our interpretation of the execution semantics of activities, especially the token semantics. In other words, we describe how our interpreter (introduced in Chapter 7) executes activities. It should be noted that the token semantics described in the UML specification is "not intended to dictate the way

activities are implemented, despite the use of the term 'execution' " [59, §12.3.4]. Instead, it describes the desired results of execution, e.g., the "sequence and conditions for behaviors to start and stop" [59, §12.3.4]. Our interpretation of activity execution is consistent with the end result described by the specification. In particular, we have created an execution algorithm that implements the token offer and passing semantics described in the specification.

6.1 Token Offers

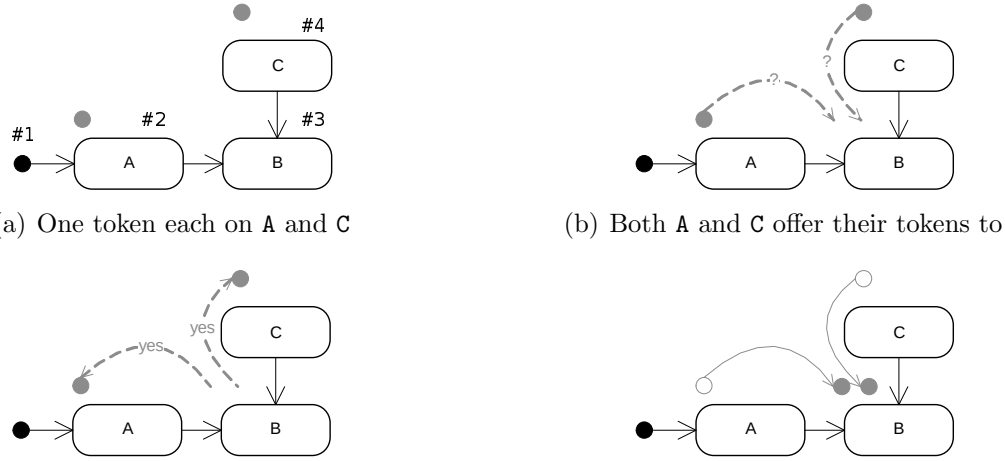
Although UML 2 activities have been "redesigned to use a Petri-like semantics" [59, §12.3.4], they are distinguished in two ways from Petri nets [11]:

- In an activity, tokens move concurrently, and asynchronously. In a Petri net, tokens move all at the same time, i.e., in lock-step.
- In an activity, the decision of which token to move, i.e., which transition to fire, can involved multiple unconnected edges and nodes.

Theoretically, tokens move through an activity in accordance with a "traverse-to-completion" [5] semantics. In essence, a token only moves to a target if it can be used there immediately; the purpose of this strategy is to help avoid deadlock. The mechanism for ensuring that tokens only move when they can be used is through the use of token offers.

Consider, for instance, the activity sequence in Figure 6.1. This sequence shows how typical token offer and passing occurs. Figure 6.1(a): Both `A` and `C` have finished executing. The grey circles represent tokens. Figure 6.1(b): Both actions offer a control token to their target, i.e., `B`. Figure 6.1(c): Action `B` can only execute if it

receives tokens from all of its sources. In this case, because both `A` and `C` are offering their tokens *simultaneously*, `B` can accept both offers. Figure 6.1(d): Because their offers have been accepted, both `A` and `C` can pass their tokens to `B`.



(a) One token each on `A` and `C`

(b) Both `A` and `C` offer their tokens to `B`

(c) `B` accepts the offers from `A` and `C`

(d) Tokens from `A` and `C` move to `B`

Figure 6.1: Activity sequence showing token offers and token passing

The simultaneous offering of tokens is important in this case. If `A` had finished executing, but not `C`, `B` would not be able to accept `A`'s offer. This is because `B` needs tokens on both of its incoming edges. In that case, `A` would be *stalled*, a state where an action has finished execution but control cannot be passed because the action's target cannot accept it.

The need to deal with this rather delicate token offer and passing mechanism complicates the implementation of our interpreter. By definition, our implementation is restricted to sequential execution of activities, even though activities are by nature concurrent. We simulate concurrency through interleaving; however, this complicates the nature of the token offers and token passing. For instance, we need to maintain

the offer from `A` to `B`, even though `B` cannot yet accept it. Other aspects of activities further complicate this semantics, e.g., competition for tokens leaving object nodes, the necessity for offers/tokens to pass through multiple control nodes, etc.

In the rest of this chapter, we focus on key parts of our interpreter's execution algorithm, using example activities and pseudocode for illustration. Note that the pseudocode presented here provides a very high-level view of the algorithm; much detail has been omitted for brevity.

6.2 Initialization

Before an activity can begin execution, it must be initialized. This simply means that every initial node of the activity is populated with a token.[1]

```
FOR every initial graph node in the activity
  assign a new token/thread pair to that node
ENDFOR
```

Consider the activity in Figure 6.2. Tokens (grey circles) have been placed on the initial nodes, in this case, nodes `#1` and `#4`. Remember that an initial node is any node in the activity diagram graph that has no incoming edges.

6.3 Execution Algorithm

As an activity is executed, the underlying System Model state is changed; a *macro step* is the full transition from state to state. Inside this macro step are *micro steps*,

[1]Recall from Chapter 4 that tokens (program counters) and threads are paired. Each token corresponds to a specific thread and vice versa.

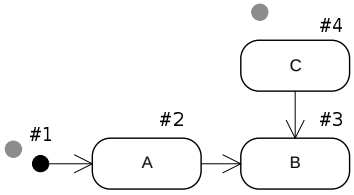


Figure 6.2: Tokens are placed on the initial nodes of the activity

specifically:

1. **Choose Next Token** The scheduler examines all tokens in the current state. Of those tokens that are resting on nodes that are currently enabled,[2] one token is chosen non-deterministically.

2. **Execute node/action** The node that the chosen token is resting on is executed. The execution of the node itself takes place at the node level of detail (see Figure 6.3). Furthermore, if an action node is being executed, then its specific action is executed at the action level of detail. Assuming that the node can complete execution immediately,[3] token offers can be created for the node's immediate targets.

3. **Offer and Pass Tokens** Now that the executed node (usually an action) has modified the underlying state, all that remains is to make the offers and pass the tokens. The scheduler deals with all offers/tokens in one batch.

[2]An action node is enabled if it is not stalled (due to token offers and passing), it does not represent a call action waiting for a result, it does not represent an accept action waiting for an event, and it has sufficient tokens on its incoming edges. See Section 4.3 for the formal definition of 'enabled'.

[3]In almost all situations, a node can complete execution (and thus offer tokens to its targets) in one macro step. However, when either the CallOperationAction or the CallBehaviorAction are executed synchronously, the action calls the target behaviour and then must wait for that behaviour to complete. Only then can the calling action finish its execution and offer tokens.

The activity diagram in Figure 6.3 summarizes the execution of a macro step. We distinguish between three levels of detail for execution. Tasks at the activity level require access to the entire activity. Tasks at the node level are restricted to actual node instances, e.g., action node, control node, etc. Finally, tasks at the action level are linked to implementations of the individual UML actions. Changes made to the System Model's data store and event store are typically made at the action level, while changes to the control store are typically made at the activity level.

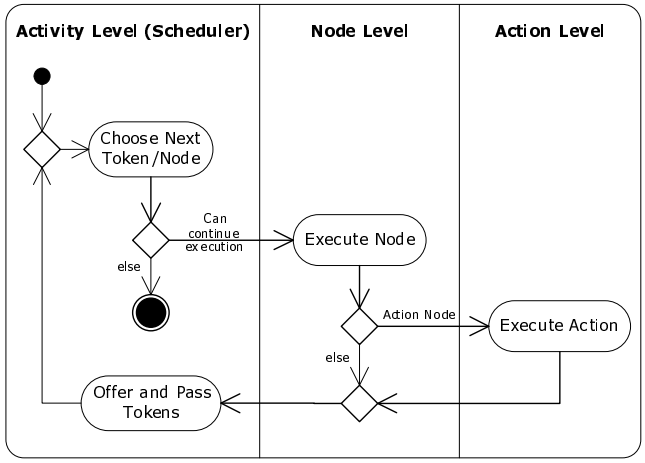


Figure 6.3: Activity showing tasks required during a *macro* step from state to state. Partitions are used to separate tasks by level of detail required; tasks on the left require access to the entire activity while those on the right are related to specific UML actions

6.3.1 Macro Step

The interpreter repeatedly performs macro steps until no more tokens can be dealt with. 'Good' tokens are those sitting on enabled nodes, i.e., nodes that can execute. When there are no more 'good' tokens left, the macro step is complete.

```
WHILE there are tokens
  //Start Macro Step

  //Choose next token
  DO
    pick a random token //Micro Step 1
  WHILE the token's node is NOT enabled

  IF we still don't have a token on an enabled node
    error: there are still tokens but we can not work with any of them
    BREAK //leave while loop; nothing more can be done
  ENDIF

  //Get here, and we have a token on an enabled node
  execute the token's node //Micro Step 2
  offer and pass tokens //Micro Step 3

END WHILE
//No more tokens to work with.  Execution complete.
```

6.3.1.1 Micro Step 1: Choose Next Token

The purpose of this micro step is to find a token that is sitting on an enabled node, e.g., a node that can be executed. Tokens are maintained in a pool. Tokens are retrieved non-deterministically from the pool until a 'good' token is found. Then, the node on which that token sits can be executed.

6.3.1.2 Micro Step 2: Execute Node/Action

For the purposes of describing the execution of an activity, it is important to distinguish between action nodes, object nodes and control nodes.

Control Nodes In general, control nodes are not 'executed'; the main effect of these nodes is in how tokens are offered and passed.

Object Nodes Object nodes are not typically executed either, since they represent pins and are just used to pass values between action nodes. Again, these nodes affect how tokens are offered and passed in the activity. However, it is possible that an activity contains object nodes that have no sources or no targets. In those cases, they can be 'executed'. Each object node has the name of a value associated with it; executing an object node with no sources or no targets simply means retrieving that specific value from the System Model universe.

Action Nodes As can be seen by the swim lanes in the activity in Figure 6.3, the actual execution of an action node takes place at two levels–the action level and the node level. The two levels are necessary because sometimes an action needs more information than it has immediately accessible. Consider the ReadSelfAction, as discussed in the summary Table 5.2. This action needs information available at the node level, specifically, the identity of the object owning the activity that the node is in.

The execution of the following actions takes place at both the node and action level:

- AcceptCallAction
- AcceptEventAction
- BroadcastSignalAction
- CallBehaviorAction
- CallOperationAction
- ReadSelfAction
- ReplyAction
- SendObjectAction

- SendSignalAction
- StartClassifierBehaviorAction

We do not discuss the execution of action nodes any further, as their executions have been formally defined in Chapter 5.

6.3.1.3 Micro Step 3: Offer and Pass Tokens

This step is comprised of two phases. First, the node that just finished executing offers tokens to its targets. A node's targets are those nodes that it is directly connected to via a flow in the graph representing the activity. Offers must often be propagated through the graph until they can be accepted or rejected.

The propagation of offers is best explained by example. Consider the activity in Figure 6.4. If `A` finished execution, it would make offers to both `B` and `C`. If `B` finished execution, it would make an offer to the fork node. That offer would be propagated to both `D` and `E`, since the fork cannot accept or reject an offer. If `D` finished execution, it would make an offer to the join node. This offer would eventually be propagated to `F`, but not until the join also received an offer from `E`. If `C` finished execution, it would make an offer to `O`. Because in this case, `O` is acting as a pin between two actions, the offer would be propagated to `F`. If `O` had no targets, it would be able to accept the offer. Finally, if `F` finished execution, it would make an offer to the final node, which would accept the offer.

All outstanding offers are maintained in a collection. Then, to handle the fact that some offers cannot be immediately accepted and must be propagated, and the fact some nodes can only accept one offer if other related offers are outstanding, a token-passing algorithm is executed. This algorithm makes use of fixed-point iteration,

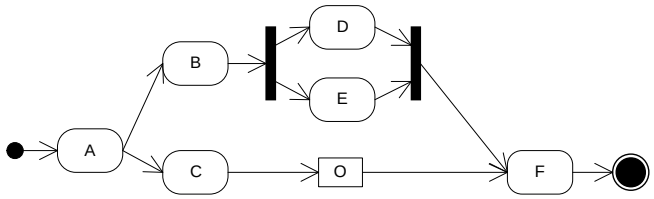


Figure 6.4: Sample activity to demonstrate propagation of offers

allowing for offers to propagate until they can be ultimately accepted/rejected and for multiple offers to be accepted 'simultaneously'.

Offering Tokens

Initial Nodes, Object Nodes, Fork Nodes Most nodes, such as initial nodes, object nodes and fork nodes, offer tokens to all of their targets.

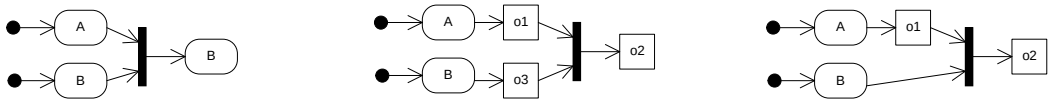
```
FOR every target
  create a new offer for that target
  put that offer into the offers collection
ENDFOR
```

Action Nodes The action node offers tokens on its outgoing edges. However, offers can only be offered if the action has finished execution—a call action waiting for a response cannot make offers.

```
IF NOT waiting for a response
  FOR every target
    create a new offer for that target
    put that offer into the offers collection
  ENDFOR
ENDIF
```

Join Nodes The join node is the most complicated control node with respect to making offers. The join node may have multiple incoming tokens. If there is incoming data, then that data must be offered to the join's target. On the other hand, if there are only control tokens coming into the join node, only one control token is offered to the target.

Consider the three activities in Figure 6.5. In Figure 6.5(a), only control tokens are being offered to the join; it would offer one control token to its target. In Figure 6.5(b), only data is being offered to the join; it would offer both data tokens to its target, in the order in which they were received at the join. In Figure 6.5(c), both control and data is being offered to the join; it would offer only the data to its target.



(a) Only control (b) Only data (c) Control and data

Figure 6.5: Offering control, data and both to a join node

```
IF join has no incoming data
  create a new (control) offer for the target
  put that offer into the offers collection
ELSE
  //At least one data token coming in
  FOR every incoming data token
    create a new (data) offer for the target
    put that offer into the offers collection
  ENDFOR
ENDIF
```

Decision Nodes Decision nodes are never 'executed' per se; they are encountered (and processed) during the passing of tokens, i.e., after an offer has been made

and accepted. When an offer comes through from another node, each outgoing branch of the decision node is examined. For each branch whose guard evaluates to *true*, the incoming offer is propagated to the target. If more than one target accepts the offer, one target is chosen non-deterministically to win the token.

Merge Nodes Merge nodes are also never 'executed'; they are encountered (and processed) during the passing tokens, i.e., after an offer has been made and accepted. When a merge node is encountered in the process of passing a token, the token is simply passed to the merge node's target.

Final Nodes Final nodes make no offers. When a final node is encountered, the incoming token is destroyed.

Passing Tokens Once the newly-executed node has made its offers, the next phase is to handle all offers in bulk and perform the token-passing algorithm. This algorithm is based on fixed-point iteration. We handle each offer in the offers collection, adding new ones as they arise and removing ones for tokens that can be passed. Eventually, the offers collection stabilizes in that no more offers can be added or removed from the collection. At that point, we are done with this macro step's token passing phase.

```
DO
  FOR every offer in offers collection
    IF the offer can be accepted
      pass the token
      remove the offer from the collection
    ELSE IF the offer must be propagated
      propagate the offer //i.e., create new offer from target
    ELSE
      leave the offer in the collection
    ENDIF
```

```
  ENDFOR
WHILE the offers collection has changed
```

Initial Nodes, Action Nodes, Join Nodes Again, different kinds of nodes handle accepted offers and pass tokens in different ways. Most nodes, such as initial nodes, action nodes and join nodes, simply make their offers and, when those offers are accepted, send tokens to those targets accepting the offers. On the other hand, there will be competition if more than one of an object's node's targets accepts an offer. Similarly, a fork node only passes tokens if at least one target accepts an offer.

By default, tokens advance in a sequential fashion, from source to target nodes, as shown in Figures 6.6 and 6.7.

(a) Before (b) After

Figure 6.6: Showing how a control token moves during basic sequential advance from one source to one target

(a) Before (b) After

Figure 6.7: Showing how a control token moves during basic sequential advance from one source to multiple targets

The generic offer handling algorithm is as follows:

```
FOR each offer from this node
  IF target of this offer accepts
    pass token to target
    delete offer
  ENDIF
ENDFOR
```

ObjectNode The object node sends offers to its targets, but only one target can actually receive the token. If more than one target accepts the offer, then competition occurs. One target is chosen non-deterministically to receive the token. The remaining offers are destroyed.

```
//Put all accepted offers into a collection

FOR each offer from this node
  IF target of this offer accepts
    put that offer into collection
  ENDIF
ENDFOR

IF collection size == 0
  //No targets accepting
ELSE IF collection size == 1
  //One accepting target - no competition
  get offer from collection
  pass token to target
  delete offer
ELSE
  //Competition
  pick a random offer from collection
  pass token to target
  FOR each offer in collection
    delete offer
  ENDFOR
ENDIF
```

ForkNode If at least one outgoing edge of a fork accepts an offer, then the token is duplicated and sent along every edge that accepts it. If the target refuses the token, the edge can technically hold it. Edges that do not want the token (e.g., edges with failing guards) do not get tokens. For our purposes, we assume that the guard on every edge from a fork is true, so all edges accept the tokens.

```
allaccepting = FALSE
FOR each offer from this node
  IF target of this offer accepts
    allaccepting = TRUE
  ENDIF
ENDFOR

IF allaccepting
  //At least one target accepts the offer
  FOR each offer from this node
    IF target of this offer accepts
      pass token to target
      delete offer
    ENDIF
    //else offer stays alive
  ENDFOR
ENDIF
```

Accepting Offers Different types of nodes respond to offers in different ways. For instance, a final node always accepts an offer. Offers are never made to initial nodes. The remaining control nodes accept an offer if one (or more) of their targets accepts the offer. Similarly, object nodes do not accept offers in and of themselves, unless they have no targets. Instead, an object node, like a control node, passes the offer to its targets. In fact, the only node that can truly accept or reject offers is an action node, since control is passed from action node to action node.

6.4 Data Flow

Thus far, we have focused mainly on the concept of control flow, even when dealing with object nodes. We treat all tokens in an activity as control tokens. Data is passed by moving a control token *through* an object node.

For instance, consider the activity in Figure 6.8(a). There are two action nodes and a control token being passed from one to the other. On the other hand, Figure 6.8(b) shows two action nodes connected by an object node. The object node represents a pin, i.e., the output pin for `A` and input pin for `B`. In other words, there is data flow between the two actions. In terms of our interpreter, we still see the token as a control token, but when it passes through the object node, the data essentially piggy-backs on the token, making it, in essence, a data token.

(a) Control flow (b) Control flow with data passing

Figure 6.8: Difference between control flow and control flow with data passing

Chapter 7

Interpreter

In this chapter, we describe an interpreter for UML 2 actions and activities. As described in Chapter 6, we have created a execution algorithm that was designed to conform to the UML 2 specification [59] in the sense that an activity's execution is consistent with its description of expected runtime behaviour. The interpreter offers an array of analysis capabilities, ranging from random execution to reachability properties and assertion and deadlock checking. Finally, the interpreter is based on the System Model and shows how, for instance, an execution affects the structural foundation.

In 2005, the Object Management Group (OMG) issued a Request For Proposal for a foundational subset of the UML 2 metamodel that would provide a shared foundation for higher-level UML modelling concepts such as activities, state machines, and interactions and thus define "a basic virtual machine for the Unified Modeling Language...enabling compliant models to be transformed into various executable forms for verification, integration, and deployment" [58]. Using the System Model,

we have created a partial implementation of such a virtual machine. While our implementation currently only supports actions and activities, the architecture and the underlying semantic domain facilitate the extension to other behavioural formalisms, such as state machines and interactions and thus the development of a comprehensive UML virtual machine.

The interpreter implements the majority of the action behaviours formally defined in Chapter 5 and the execution algorithm described in Chapter 6. The differences between our interpreter and the execution behaviour described thus far are:

- The SendObjectAction and BroadcastSignalAction are not implemented by the interpreter. All other actions formally defined in Chapter 5 are implemented.
- We restrict structural features and variables to storing one value. Thus, the RemoveStructuralFeatureValueAction and ClearStructuralFeatureAction behave identically, as do the RemoveVariableValueAction and ClearVariableAction. Similarly, the AddStructuralFeatureValueAction and AddVariableValueAction overwrite any previous value, instead of adding an additional value. Finally, the ReadStructuralFeatureAction and ReadVariableAction can only return single values.
- With respect to the execution algorithm outlined in Chapter 6, the interpreter's actual execution is obviously much more defined, including significant error checking.

7.1 ACTi

ACTi is our interpreter for UML actions and activities. It was designed to conform to the three-layer semantics architecture in Figure 7.1 and covers the shaded portion

of that figure.

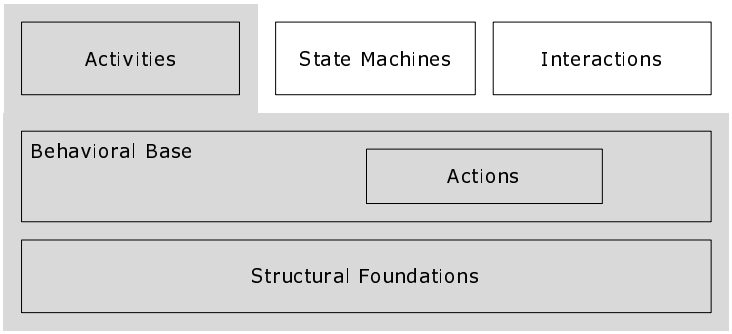


Figure 7.1: The UML three-layer semantic architecture. Shading indicates the parts covered by the interpreter

ACTi was developed using Java and consists of approximately 16,000 lines of code, including the implementation of the System Model and actions. The structure of the implementation is shown in Figure 7.2. The System Model package implements the structural foundation of UML, the Actions package implements the individual UML actions, and the interpreter itself implements activities. As in the three-layer architecture, each layer depends on the layers below, but not vice versa.

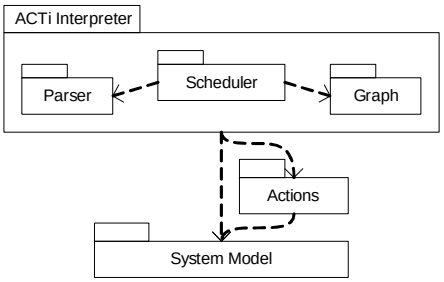


Figure 7.2: Structure of ACTi implementation conforms to semantics hierarchy in Figure 7.1

A high-level view of the process of using ACTi to execute an activity is shown in

Figure 7.3. In essence, ACTi can take as input a text file representing an activity and provide as output one, or if possible, many paths[1] detailing the execution of an activity. As the activity executes, the underlying System Model state is modified. With the aid of a third-party tool, it is possible to visualize, and even navigate, the various states of the System Model.

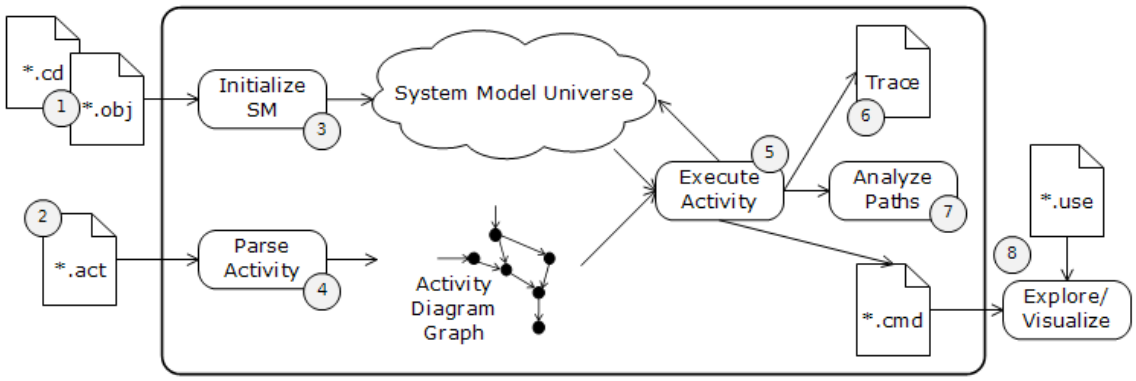


Figure 7.3: High-level view of how ACTi can be used to execute an activity. All steps inside the large rounded rectangle are automated. Eight elements (steps or artifacts) in the figure are labelled; these elements are discussed in detail in Section 7.1.1

7.1.1 Example Execution

We use a running example to explore how ACTi can be used to execute and analyze an activity. Consider the activity in Figure 7.4. This activity takes a *Car* object, and changes the value of its *year* attribute. The new value of the attribute depends on the result of a decision. In this case, both outgoing edges of the decision have true guards, meaning that either outcome is equally possible.

[1]Activities are concurrent by nature. The implementation, however, is restricted to sequential execution that simulates concurrency through interleaving.

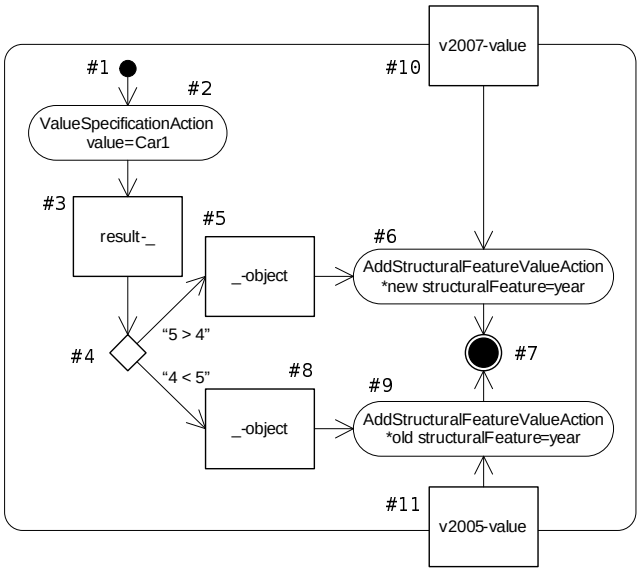


Figure 7.4: This activity will change the value of *Car1.year*. The new value depends on the non-deterministic outcome of the decision node

The steps below relate to the labelled elements shown in Figure 7.3:

① **cd and obj Files** The user is responsible for providing two text files that permit ACTi to initialize the System Model universe. The first, a cd file, details the static structure of objects that will be used—in essence, it is a simple representation of a class diagram. The second file is an optional obj file, used to detail the set of objects that exist when the activity begins execution, i.e., a user-defined initial state. Samples are shown in Figures 7.5 and 7.6. The formats of cd and obj files are described in Appendix D.

```
type Int
  value v2000
  value v2005
  value v2007

class Car
  attr year:Int
```

Figure 7.5: cd file. Describes *Int* type name with three values, and *Car* class with one attribute

```
obj Car1:Car
  attr year=v2000
```

Figure 7.6: obj file. Describes one *Car* object with *year* attribute set

② **act File** The user is also responsible for providing the activity to be executed, in the form of an act file. An activity file is written in the Activity Diagram Linear Form (ADLF) [31], a lightweight textual representation of UML activities, intended to be easy to read and write. Figure 7.7 contains an ADLF representation of the activity in Figure 7.4. The format of act files is described in Appendix C.

```
<InitialNode>
 -> (ValueSpecificationAction value=Car1)
 -> [result-_]
 -> <DecisionNode>{
   "5 > 4" -> [_-object] ->
    (AddStructuralFeatureValueAction *new structuralFeature=year)
   -> <ActivityFinal *final>,
   "4 < 5" -> [_-object] ->
    (AddStructuralFeatureValueAction *old structuralFeature=year)
   -> <*final>
   };
[v2007-value] -> (*new);
[v2005-value] -> (*old).
```

Figure 7.7: ADLF representation of activity in Figure 7.4

③ **Initialize SM** The interpreter prepares the underlying System Model universe using the cd and obj files.

④ **Parse Activity** The interpreter parses the incoming act file, creating a graph representing the activity. Each element of the activity is represented as a type of graph node, e.g., action nodes (shown as rounded rectangles) are linked to action instances. All nodes are labelled for identification purposes and instances representing control tokens are placed on the initial nodes[2] of the resulting graph. At this point, the activity is ready for execution.

⑤ **Execute Activity**

ACTi supports six different modes of execution, permitting users to explore activities in various ways. Table 7.1 shows these six modes, categorized along two dimensions. One dimension refers to the level of abstraction employed during execution. A 'deep' execution applies the standard semantics of actions and thus avoids all abstraction. By default, actions are assumed to be specific UML actions, e.g., CreateObjectAction, ReadVariableAction, etc. However, it is also possible for a user to define their own actions in the 'deep' mode. These actions must be accompanied by executable code,[3] which makes changes to the underlying System Model state. On the other hand, a 'shallow' execution applies a non-standard semantics, which treats every action as a 'no-op', leaving the data store unchanged but advancing tokens. This type of execution is useful for experimenting with control flow, e.g., looking for different paths through an

[2]Initial nodes in the graph include initial nodes in the activity, as well as any node without incoming edges. In the case of the activity in Figure 7.4, the initial nodes are `#1`, `#10`, `#11`.

[3]Since ACTi has been implemented in Java, the user must provide a Java class. See Section 8.1 for an example.

activity, checking for deadlock, etc.

Table 7.1: ACTi modes of execution

	Default Initial State	**User-Defined Initial State**
Deep Detail	Random Guided	Random Guided
Shallow Detail	Random Guided	

The other dimension refers to the initial state of the System Model when the activity starts executing. With the 'default' initial state, the System Model universe has been defined by the user, but no objects exist. Alternatively, after defining the universe, the user can also define a set of objects (with values for their attributes) that should exist when the activity begins execution.

In three of the quadrants[4] of the modes table there are listed two types of execution:

- **Random** During execution, all non-deterministic choices are resolved using randomization. For instance, when there are multiple enabled nodes ready to be executed, one node is chosen at random. Also, if there are multiple true edges out of a decision node (with targets that are willing to accept the token offer), one edge is chosen at random to be traversed. Finally, when competition occurs, e.g., an object node is the source of

[4]Note that there are no execution types listed in the lower-right quadrant of Table 7.1. When performing a shallow execution, there is no advantage in setting the initial state in the System Model, as the execution will not modify the System Model at all.

multiple edges, the winning edge is chosen at random. This randomization ensures that the more executions that are performed on a particular activity, the more likely it is that different paths will be found.[5]

- **Guided** It is also possible to suggest a path to the interpreter, forcing it to execute certain nodes in a specific order. This mode of execution is useful for exploring how the System Model changes in response to a specific execution. The user is notified if the specified path cannot be followed.

We will now describe deep, random execution in more detail; this is probably the most commonly used mode of execution. We will assume that the initial state has been defined by the obj file shown in Figure 7.6. Note that a guided execution would be similar, except that no choices need be made with respect to the next node to execute; that would be predetermined by the user. On the other hand, a shallow execution would make use of the same randomness described above; however, no changes would be made to the system model by the actions themselves. Finally, the use of the default initial state would not affect the described execution.

ACTi includes a scheduler that uses the token passing semantics detailed in the UML specification. As an activity is executed, the underlying System Model state is changed as described by the activity in Figure 6.3 on page 169. In other words, the transition from state to state is a macro step, composed of three micro steps: 'choose next token', 'execute node/action' and 'offer and pass tokens'.

[5]It should be noted that the interpreter makes no claims of fairness, i.e., there is no guarantee that every possible path will be chosen eventually. However, experiments have demonstrated that ACTi's choices are indeed random, i.e., follow a probability distribution.

⑥ **Trace** It is possible to have ACTi provide differing levels of trace information as output. At the very minimum, ACTi can simply output a list of nodes; the order of these nodes constitutes the resulting path through the graph. At the other end of the extreme, ACTi can perform a full trace, detailing exactly what is happening at each point in the execution, including how the scheduler is choosing tokens for execution, how token offers are proceeding, how the System Model is being modified, etc. In fact, the trace can also include a detailed listing of the System Model universe as each action is executed. A trace with a medium amount of information is shown in Figure 7.8. It contains information about which node is executed, and how tokens are being offered and passed.

⑦ **Analyze Paths** When configuring the execution of the desired activity, the user can also request different types of analysis. Although analysis can be performed on any single execution (random or guided), it is more useful when performed on multiple random executions. See Section 7.2 for a discussion of the analysis techniques that can be employed.

⑧ **Explore/Visualize** In addition to providing a textual trace output, ACTi can also generate a series of cmd files that the user can incorporate as input into a third-party tool for visualization. The UML-based Specification Environment (USE) [80] is a powerful tool created at the University of Bremen that is used for the specification and analysis of systems. The user provides a USE model to create the structure of the specification. The tool provides for the visualization of the system states. We have created a USE model of the underlying System Model. That model, combined with the cmd files created when executing an activity, permits us to statically explore and visualize the state of the System

Model as each node in the activity is executed. Chapter 8 contains an example of this type of visualization.

7.1.2 Token Passing

As discussed in Section 2.4.2, the purpose of the token offer semantics is to avoid deadlock while executing an activity. The process of dealing with token offers and token passing is performed by the scheduler at the activity level (see Figure 6.3). When each node has completed execution, it creates token offers for its targets. In general, only action nodes can accept or reject offers; this means that an offer often needs to be propagated through the activity until an action node is encountered. All outstanding offers are maintained in a collection by the scheduler. The last part of the macro step is to process all offers in a batch. The scheduler examines each offer in the collection, determining whether or not the offer can be accepted by its target. Typically, when a token offer is accepted by a target node, the token gets passed to that node. However, many exceptions exist, for instance, a node with two incoming edges can only accept a token on one edge if it has also been offered a token on the other edge. To implement this semantics while processing the token offers, we perform a fixed-point iteration. Offers are iteratively processed until no more tokens can be transferred and no more offers have been propagated.

The trace in Figure 7.8 shows how complicated token passing can be. This is the trace of the macro step in Figure 7.4 in which the ValueSpecificationAction executes. The token offer and passing is complicated because the action must offer a token to another action node, e.g., either (or both) of the two AddStructuralFeatureValueAction nodes, which are on the other side of a decision node.

```
// Executed so far: #1, #10, #11
38.--- Macro Step ------------------------------------------------------------------------------
39.   ...ACTi.....Looking for enabled token/node.....................................................
40.        Out of all tokens: Token_4 (on #2), Token_2 (stalled on #11), Token_3 (stalled on #10)
41.        Scheduler has chosen Token_4, sitting on enabled #2(ValueSpecificationAction value=Car1)
42.        [Token_4] is/are at node #2, which is about to fire
43.   ...Node Level.....Executing Node.............................................................
44.        Firing #2(ValueSpecificationAction value=Car1)
45.   ...ACTi.....Offer Process.....................................................................
46.      #2 --?-> #3 #2 has offered Car1 to #3
47.      Looking at offer [#10] --v2007--> (#6)       //Rejected
48.      Looking at offer [#11] --v2005--> (#9)       //Rejected
49.      Looking at offer (#2) --Car1--> [#3]
50.      #3 --?-> #4                                  #3 has offered (temporarily) Car1 to #4
51.      #4 --?-> #8                                  #4 has offered (temporarily) Car1 to #8
52.      #8 --?-> #9                                  #8 has offered (temporarily) Car1 to #9
53.      #8 <-Y-- #9                                  #9 has accepted the offer from #8
54.      #4 <-Y-- #8                                  #8 accepted the offer from #4
55.      #4 --?-> #5                                  #4 has offered (temporarily) Car1 to #5
56.      #5 --?-> #6                                  #5 has offered (temporarily) Car1 to #6
57.      #5 <-Y-- #6                                  #6 has accepted the offer from #5
58.      #4 <-Y-- #5                                  #5 accepted the offer from #4
59.      #3 <-Y-- #4                                  #4 has accepted the offer from #3
60.      #2 <-Y-- #3                                  #3 has accepted the offer from #2
         //Random choice: #9 wins
61.      #2 --!-> #3                                  #3 has won the offer from #2
62.      Passing new control Token_5 from #2 to #9.         #2 -o-> #9
63.      Deleting old control Token_4 from #2
64.      Looking at offer [#10] --v2007--> (#6)       //Rejected
65.      Looking at offer [#11] --v2005--> (#9)
66.      #11 <-Y-- #9          #9 has accepted the offer from #11
67.      #11 --!-> #9          #9 has won the offer from #11
68.      Passing new control Token_6 from #11 to #9.        #11 -o-> #9
69.      Deleting old control Token_2 from #11
70.      Looking at offer [#10] --v2007--> (#6)       //Rejected
```

Figure 7.8: Partial trace of execution of activity in Figure 7.4. This trace shows the macro step in which the ValueSpecificationAction executes, and the subsequent token offers and token passing

When node #2 finishes execution, it creates an offer to its immediate target, the object node #3. At this point, there are three offers in the scheduler's collection (because #10 and #11 have already been 'executed'). As a pin, #3 cannot accept the offer; it must pass the offer to its immediate target(s). In this case, the sole target is a decision node, which also cannot accept an offer. The decision node passes the offer along to all of its targets with positive guards, in this case, both of the

AddStructuralFeatureValueAction nodes, #6 and #9. As can be seen from lines 53 and 57, both actions accept the offer. This acceptance is due to the fact that the other offers that these actions are waiting for have already been noted in this offer iteration. If it had been the case that either #10 or #11 had not yet executed, one or both of the actions would have rejected the offer.

In this case, both actions accept the offer; however, only one of them can be the recipient, resulting in competition for the token. The scheduler randomly chooses among the accepting targets. Here, #9 is the ultimate winner and a new token is passed to the action node.

The scheduler then iterates again through the offers. This time, the offer from #11 to #9 can now be accepted and passed also to the AddStructuralFeatureValueAction. At this point, that action is now considered enabled. The remaining offer from #10 to #6 can not be accepted, and indeed, will never be accepted in this execution.

7.2 Analysis

As discussed in Section 7.1.1, there are six specific modes available for executing activity diagrams. One useful analysis is to determine that there is indeed a suitable path through the activity. Another very useful analysis is to find as many paths as possible by randomly executing the activity multiple times. For example, there are 30 unique execution paths through the activity in Figure 7.4; see Figure 7.9. Technically, ACTi does not have the ability to examine the activity and exhaustively enumerate all possible paths. However, in random execution mode, the more times an activity is executed, the more likely it is that all paths will be found. ACTi also outputs the

probability[6] of each path being chosen.

```
Executing activity diagram n = 10000 times...
==============================================
There are at least 30 paths through the
diagram, with the following probabilities:

        1.   1 10 11 2 6 7    293/10000  = 2.93%
        2.   1 10 11 2 9 7    245/10000  = 2.45%
        3.   1 10 2 11 6 7    206/10000  = 2.06%
        4.   1 10 2 6 11 7    195/10000  = 1.95%
        5.   1 10 2 6 7       176/10000  = 1.76%
             ...
```

Figure 7.9: Listing multiple paths found for activity in Figure 7.4

7.2.1 Path Analysis

In addition to being able to find paths through a specific activity, ACTi also has several analysis capabilities. These analyses are most useful when executing an activity multiple times. The result is that ACTi can function as a limited model checker, i.e., the analysis is performed for each path found. The following analyses are available:

- **Desirable Nodes** The user defines a set of desirable nodes. Any path that does not contain all desirable nodes is considered a failure. This analysis is especially useful in larger activities incorporating decisions, merges, forks and joins. It can be used for reachability analysis, i.e., asserting that a desirable node is always reached.

[6]Experiments have demonstrated that ACTi accurately calculates both the number of paths and the probability of each path being chosen. We have manually calculated the expected probability of each path for activities with up to 36 paths; the actual probabilities computed by ACTi closely match these expected probabilities.

- **Undesirable Nodes** The user defines a set of undesirable nodes. Any path that contains any undesirable node is considered a failure. Similar to the previous analysis, this can be used to assert that an undesirable outcome is never reached.

- **Mutual Exclusion** The user defines a pair of nodes. Any path that contains both nodes is considered a failure. This analysis can be used in situations where it is desirable that either one node or another is encountered, but never both. For example, analysis of the activity in Figure 7.4 shows that nodes `#6` and `#9` are indeed mutually exclusive. Figure 7.10 shows the results of performing this analysis with our example activity.

```
Checking for Mutual Exclusivity
================================================
Expecting only one of #6 and #9 to occur in any
path.  Paths where *both* of these occur are
marked with an asterisk.

        1.   1 10 11 2 6 7
        2.   1 10 11 2 9 7
        3.   1 10 2 11 6 7
             ...
       29.   11 10 1 2 6 7
       30.   11 10 1 2 9 7

Summary:
  Have any paths contained both #6 and #9? false
  0 / 30 = 0.0% of paths contain both nodes.
```

Figure 7.10: Results of checking for mutual exclusion between nodes `#6` and `#9` in the activity from Figure 7.4

- **Precedence** The user defines an ordered pair of nodes. Any path that contains both nodes, but out of order, is considered a failure. This analysis can be

used for debugging activities where it is desired that if two given nodes are encountered, they are encountered in a specific order.

- **Execute Node n Times** Some activities contain loops; this analysis can be used to confirm that a specific node is encountered at least n times in every path.

- **Assertion Checking** It is possible to create basic assertions over object attributes. For instance, for the activity in Figure 7.4, we could assert that $Car1.year = v2007$. This analysis would check this assertion at the end of every encountered path.

  As mentioned earlier, there is non-determinism inherent in the activity in Figure 7.4. The non-determinism is caused, in part, by the fact that both outgoing branches of the decision node are true. Given that either branch could be taken, we would expect that the value of $Car1.year$ would be set to $v2005$ about half the time; the other half the time, it would be set to $v2007$. Figure 7.11 shows the results of checking the assertion $Car1.year = v2007$ for our example.

```
Checking assertion [Car1.year == v2007]...
==========================================
Those paths for which the assertion does
*not hold* are marked with an asterisk.

        1.   1 10 11 2 6 7
   *    2.   1 10 11 2 9 7
        3.   1 10 2 11 6 7
             ...
   *   28.   11 1 2 9 7
       29.   11 10 1 2 6 7
   *   30.   11 10 1 2 9 7

Summary:
  Is the assertion [Car1.year == v2007]
    true for every path? false
  15 / 30 = 50.0% of paths do not satisfy
    the assertion
```

Figure 7.11: Results of checking assertion *Car1.year == v2007* on the activity from Figure 7.4. The assertion is true for half of the paths

7.2.2 Unused Tokens or Offers

The ACTi scheduler will continue to execute, choosing the next token/node to execute, until there are no more tokens on enabled nodes. When this occurs, there may be tokens and/or offers remaining in the activity. Sometimes, this may be acceptable, as in our example. For instance, in the activity in Figure 7.4, a decision is made, choosing one outgoing branch over another. When the activity in our example finishes execution, there is an outstanding offer remaining on the un-chosen branch, as can be seen by the output in Figure 7.12. In this case, the leftover offer does not constitute an error.

```
<Scheduler.transitionThroughStatesRandomly>
        There are no more tokens active in the diagram.
        Execution of the diagram is complete.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! WARNING (POTENTIALLY BAD DIAGRAM) <Scheduler.transitionThroughStatesRandomly>
! Execution of the activity diagram(s) has completed as much as
! possible. At this point, there are one or more offers still active
! in the diagram(s):
!
! . [#10] --v2007--> (#6)
!
! The diagram may or may not be considered stalled; that is up to the
! user to decide.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Figure 7.12: Warning message after executing activity from Figure 7.4. The leftover offer does not constitute a bad diagram in this case

In other cases, having leftover tokens or offers indicates that there is a problem with the activity diagram, e.g., deadlock. Consider the activity in Figure 7.13. It is impossible for this diagram to terminate. The initial node will offer a token to action `A`; however `A` can never accept this token, because it also requires an offer from action `B`.

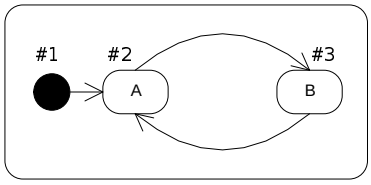


Figure 7.13: Simple activity that causes deadlock. Action `A` cannot execute until action B executes; action B cannot execute until action `A` terminates

When the scheduler can no longer find an enabled node to execute, it warns the

user if there are tokens or offers remaining in the activity; as can be seen by the output in Figure 7.14. It is up to the user to confirm if these leftover tokens or offers indicate a deadlock or not, based upon the specific activity being executed.

```
**************************************************************************
*   (3) End with 1 leftover offers:
*         <#1> --null--> (#2)
**************************************************************************

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! WARNING (Potentially Bad Diagram) <Scheduler.transitionThroughStatesRandomly>
! Execution of the activity diagram(s) has completed as much as
! possible. At this point, there are one or more tokens still active
! in the diagram(s), but none of the nodes that they are sitting on
! are ready to fire. This could be caused by an error, e.g., a call
! action that that doesn't have an appropriate response in the
! diagram. Or, an accept action might be waiting for a synchronous
! vs. asynchronous call, or vice versa. Finally, the diagram might
! just be designed such that not all tokens get used, e.g., an accept
! call action waiting for a call that never occurs.
!
! The tokens that remain are:
!  Token_1 on #1<InitialNode>
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!


Printing the order in which nodes were visited - (1)
[1]
```

Figure 7.14: Warning message after executing activity from Figure 7.13

7.2.3 Sanity Checks

ACTi also performs various 'sanity checks' during execution. Wherever possible, constraints from the specification are incorporated into the interpreter. For example, the following situations will raise warnings or errors:

- Flows into an initial or out of a final node.
- Mismatched flows into/out of a decision, merge, fork or join node.
- Missing or incorrect arguments for actions.
- Missing or incorrect static information for an action's attributes and/or associations.
- More than one 'else' branch leaving a decision node.
- A node is executed more than a predetermined number of times, i.e., rudimentary cycle checking.

Chapter 8

Evaluation

As discussed in the previous chapter, the ACTi interpreter supports six modes of execution. The running example in that chapter focused on a deep execution of an activity consisting of actual UML actions. This type of execution is useful for finding a path (or multiple paths) through a low-level activity. The purpose of this chapter is to explore, in more detail, some of ACTi's other capabilities.

For instance, ACTi supports the use of user-defined, low-level actions. We demonstrate this ability with an activity containing a user-defined action. In addition, a shallow execution can be used to explore the flow of control in any activity, not just one composed of UML actions. We demonstrate how ACTi can be used to analyze a typical activity in order to validate it.

For the most part, the examples presented thus far have not been related to programming, or even pseudocode. We present a simple example of pseudocode and then provide the activity that represents the same behaviour. Because the UML actions are so low-level, a simple pseudocode statement is often represented by several actions.

In addition to the analysis capabilities provided natively by ACTi, the interpreter also supports the creation of cmd files. These files can be used with the USE tool to visualize the elements of the System Model universe at each state of an activity's execution. This visualization is static, i.e., performed after the activity has completed execution, but can be useful for learning about the structure of the System Model and how individual UML actions affect it. We show the visualizations that can be created by executing our simple activity example above.

We also discuss in further detail the use of AcceptCallAction and AcceptEventAction to receive an asynchronous call. We provide an example showing how the two different actions can be used to perform the same function. Finally, we demonstrate how ACTi supports dynamic binding when inheritance of behaviour is encountered.

8.1 User-defined Action

The UML actions act directly on the System Model universe. The interpreter executes each action, changing the underlying state accordingly. It is also possible for the interpreter to execute user-defined actions, defined in a similar fashion to the UML actions. To create an action, the user must have some knowledge of the structure of the System Model and implement an action that specializes the basic `Action` class. Figure 8.1 contains a class diagram of a user-defined action. The purpose of this action is similar to that of the TestIdentityAction, except that instead of testing for equality, it tests for a 'less than' relation between two inputs.

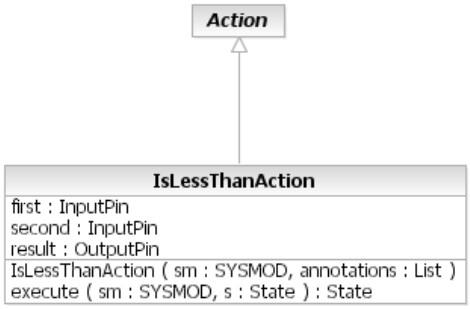


Figure 8.1: Class diagram of user-defined IsLessThanAction. This action is similar to the TestIdentityAction but tests for a 'less than' relation between two inputs

The following rules must be followed in order to create a user-defined action:

- The action must specialize the `Action` class in order to access the input and output pins of that action.
- The implementation must be contained in the `UserActions` package, which we have added to hold all user-defined actions.
- The implementation must provide a constructor that takes as input an instance of `SYSMOD` (representing an instance of the System Model) and a list of annotations. This constructor must create any input/output pins for the action.
- The implementation must provide an `execute()` method that takes as input an instance of `SYSMOD` and a System Model `State`.

If all of these conditions are met, ACTi will recognize an activity containing the user-defined action, and execute that action according to the instructions contained in the `execute()` method. To demonstrate what the actual implementation looks like, the `IsLessThan` java class is listed in Figure 8.2.

```
package actions.UserActions;

import...

public class IsLessThanAction extends Action {

    public InputPin first;
    public InputPin second;
    public OutputPin result;

    //Must have constructor with SYSMOD and List args
    public IsLessThanAction(SYSMOD sm, List annotations) {

        //Assign incoming annotations to attributes and associations
        HashMap incoming = getIncomingHash(annotations);
        //***This action has no attributes or associations to worry about.
        //Every annotation leftover is an error
        if(incoming.size() > 0) {
            Global_sm.error("WARNING","ANNOTATION","TestIdentityAction constructor", "This action " +
             "has additional annotation information, which is not being used: " + incoming);
        }

        //Create Pins
        first = new InputPin("first");
        second = new InputPin("second");
        input.add(first);     //superclass action has set of input pins
        input.add(second);
        result = new OutputPin("result");
        output.add(result);   //superclass action has set of output pins
    }

    //Every action must define this method
    public State execute(SYSMOD sm, State s) {
        //Create a shallow clone of the new state
        State sprime = (State)s.clone();

        //Make sure System Model universe contains vTrue and vFalse, since that's what must be returned.
        Value vTrue = SMReader.getValueFromSM(sm,"vTrue");
        Value vFalse = SMReader.getValueFromSM(sm,"vFalse");
        if(vTrue == null || vFalse == null) {
            Global_sm.error("ERROR","NO BOOLEAN","TestIdentityAction.execute", "...");
            System.exit(1);
        }

        //Actual behaviour of this action is here
        //-------------------------------------------------------------
        //Retrieve the values from the pins
        Value v1 = (Value)first.getContents();
        Value v2 = (Value)second.getContents();
        //Rem that Value has comparable and uses just straight string comparison to check.
        if(v1.compareTo(v2) < 0) { //< means less than
            result.setContents(vTrue);  //Call to setContents method of OutputPin
        } else {
            result.setContents(vFalse); //Call to setContents method of OutputPin
        }
        //-------------------------------------------------------------
        return sprime;
    }
}
```

Figure 8.2: Java code for the `IsLessThanAction` class

The activity in Figure 8.3 contains an example of our IsLessThanAction user-defined action. This action compares two values, each of which is the *year* attribute of a previously-defined *Car* object. If the first value is less than the second, the result is true.

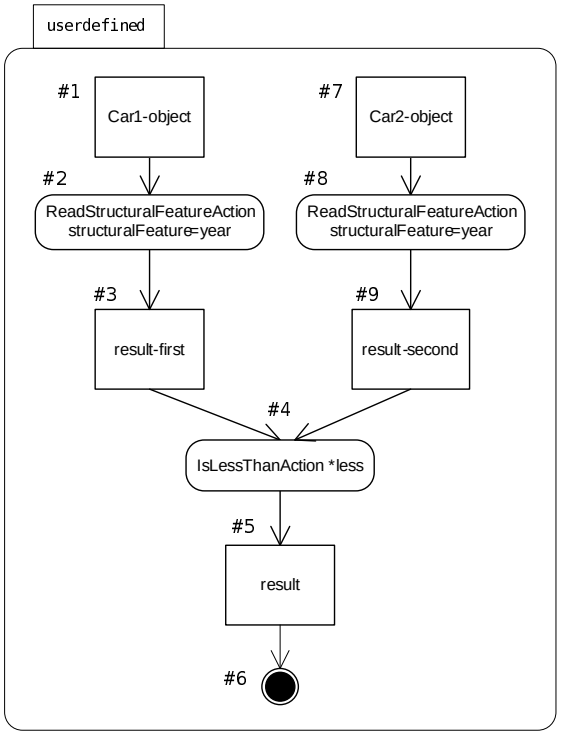


Figure 8.3: Activity to execute the user-defined IsLessThanAction

The ADLF representation of this activity is contained in Figure 8.4. The cd file and obj files, used to initialize the universe before executing this activity, are shown in Figures 8.5 and 8.6. Output from a sample execution is shown in Figure 8.7.

```
[Car1-object]
-> (ReadStructuralFeatureAction structuralFeature=year)
-> [result-first]
-> (IsLessThanAction *less)
-> [result]
-> <ActivityFinal>;

[Car2-object]
-> (ReadStructuralFeatureAction structuralFeature=year)
-> [result-second]
-> (*less).
```

Figure 8.4: ADLF representation of activity in Figure 8.3

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

class Car
 attr year:Int
```

Figure 8.5: cd file

```
obj Car1:Car
 attr year=v2005

obj Car2:Car
 attr year=v2008
```

Figure 8.6: obj file

```
Printing the order in which nodes were visited - (6)
[7, 8, 1, 2, 4, 6]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #1[Car1-object] = Car1
.  #3[result-first] = v2005
.  #5[result] = vTrue
.  #7[Car2-object] = Car2
.  #9[result-second] = v2008
.....................................................................

Running time = 471 ms = 0 sec = 0 min
```

Figure 8.7: Partial output of a sample execution of activity in Figure 8.3

8.2 Shallow Exploration

In addition to executing activities that act on the underlying System Model, ACTi can also be used to execute other, more high-level, activities. This execution is performed using the shallow mode and is useful for exploring and analyzing the type of activity diagrams normally found in software/system documentation.

Take, for instance, the diagram in Figure 8.8. This is an activity provided as part of a sample application for use in the CERAS [18] project. The purpose of the diagram is to explain, at a high level, the process a user must use to reinstate an account after it has been rendered inactive.

We use ACTi to explore this activity. Note that there are several decisions that will control the flow of control through the activity. In the interest of exploring the activity, we change the outgoing edges of these decisions so that every guard is

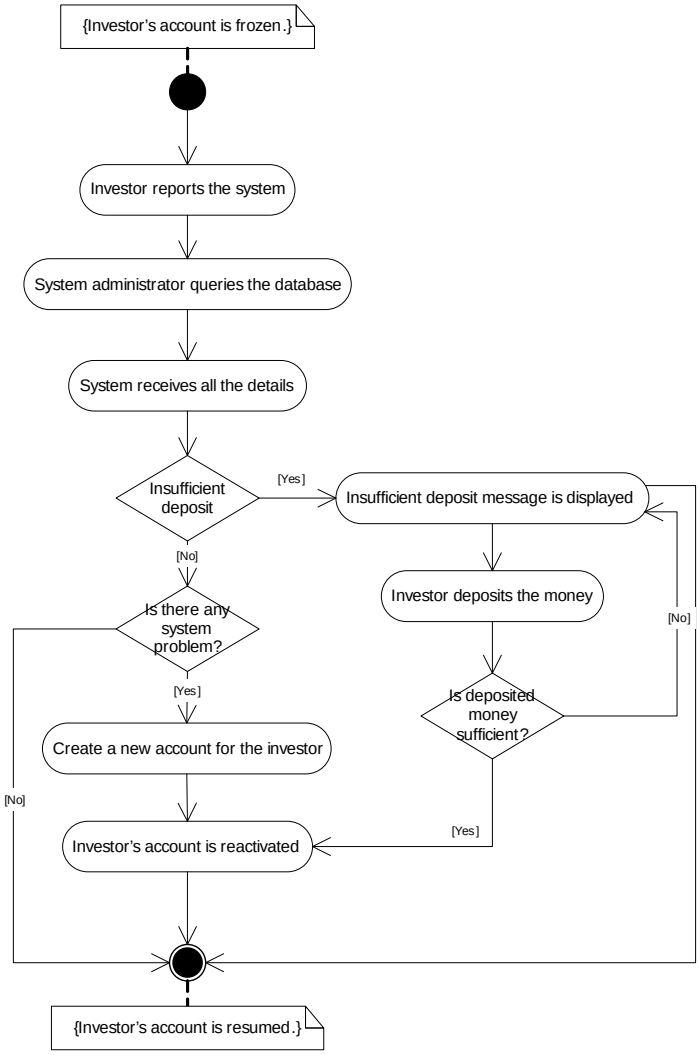


Figure 8.8: Sample activity for account reactivation, provided as part of documentation for CERAS project

```
Executing activity diagram n = 10000 times...
================================================================================
There are 2 unique paths with the following probabilities, based on n = 10000 executions.
(This means that there are *no less than* 2 paths through the diagram.)

        1.   1 2 3 4 11   5224/10000  = 52.24%
        2.   1 2 3 4 9    4776/10000  = 47.76%

Running time = 17114 ms = 17 sec = 0 min
```

Figure 8.9: Executing the activity in Figure 8.8 10,000 times results in only two paths

simply *true*. This will let us execute the diagram multiple times to see what paths are possible in general.

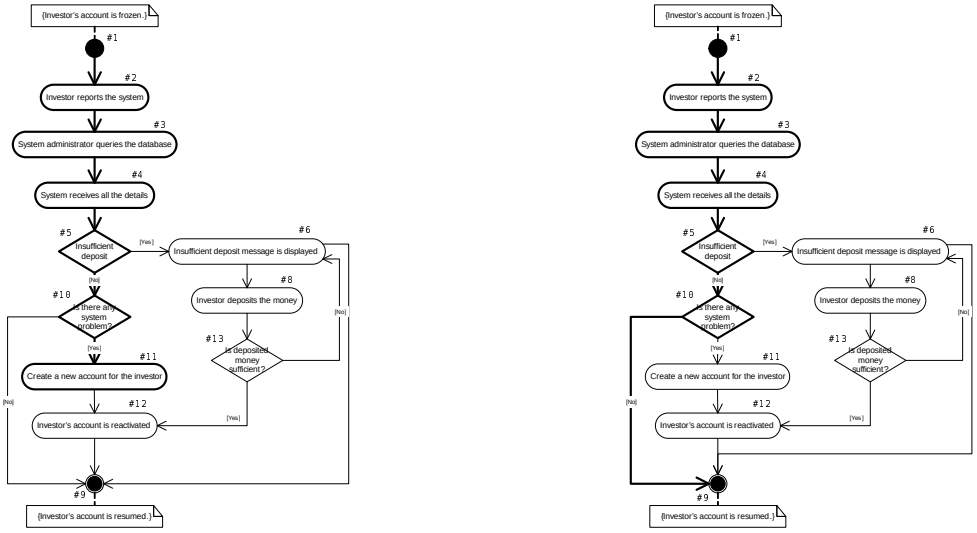
To analyze the activity, we first execute it 10,000 times to see how many potential paths there are. We expect many, simply due to the multiple decisions. We were thus surprised to find only two paths, as shown in Figure 8.9.

These two paths have been highlighted in Figure 8.10. The path in Figure 8.10(a) travels through two of the decision nodes and then ends at the activity final node. On the other hand, the path in Figure 8.10(b) halts at an action node and cannot proceed any further.

There are seven action nodes in this activity, and yet only four of these are ever reached. One of the two paths through the activity does not even reach the final node. There are obviously serious problems with this diagram, assuming that the original modeller intended for all of the nodes to be reachable.

After studying the activity, and inferring much from the names of the actions, we find the following flaws:

- The decision at node `#5` has two outgoing branches, both of which have been



(a) Path `1 2 3 4 11`

(b) Path `1 2 3 4 9`

Figure 8.10: Highlighting the two paths possible in the activity in Figure 8.8

set to *true* for our experiment. However, only one branch is ever chosen. The other branch leads to node `#6`. This node actually has two incoming edges, one from node `#5` and one from the decision at node `#13`. Because an action is only enabled if it has sufficient incoming tokens along *all* of its incoming edges, the action node `#6` will never be able to execute (the decision node `#13` can only be reached if node `#6` is executed). The description of node `#6` is "insufficient deposit message is displayed" and occurs if there is an insufficient deposit (decision `#5`) and if the deposited money is insufficient (decision `#13`). Logically, it would make more sense if this message was displayed after either of these decisions, not both. This can be accomplished by using a merge node to combine the flows from these two decisions.

- Similarly, one path always stops at node `#11` and cannot proceed any further.

That's because the action node `#12` has two incoming edges, one from `#11` and one from the decision node `#13`. Again, because this decision is never reached, node `#12` can never be enabled. As above, the way to avoid the problem with this node is to make use of a merge node to merge the control flows from node `#11` and decision `#13`.

- There is another problem with the action node `#6`. Closer examination of the activity shows that there are actually two outgoing edges for this action, one of which leads directly to the activity final node. It seems as there is an implicit decision being made at this action, i.e., a message is displayed to the investor, who then chooses to deposit more money or quits the process. When an action node with two outgoing edges is executed, control tokens will be passed along each edge, meaning that in this case the investor would deposit more money, and quit, at the same time. The solution to this problem would be to make the decision explicit with the addition of a decision node right after node `#6`, where the investor decides to either invest more, or quit.

The activity in Figure 8.11 shows the new activity, taking into account the solutions discussed above.

Executing this activity 100,000 times finds no less than 19 paths, as seen in Figure 8.12. In addition, node `#10` is apparent in some of these paths, showing that it is possible to reinstate the investor's account at least some of the time.

Careful examination of the activity shows that there is actually a cycle[1] formed

[1]Note that ACTi has a built-in check for infinite cycles and stops after a node has been visited ten times. This value can be modified in order to accommodate activities where visiting any node that often is expected.

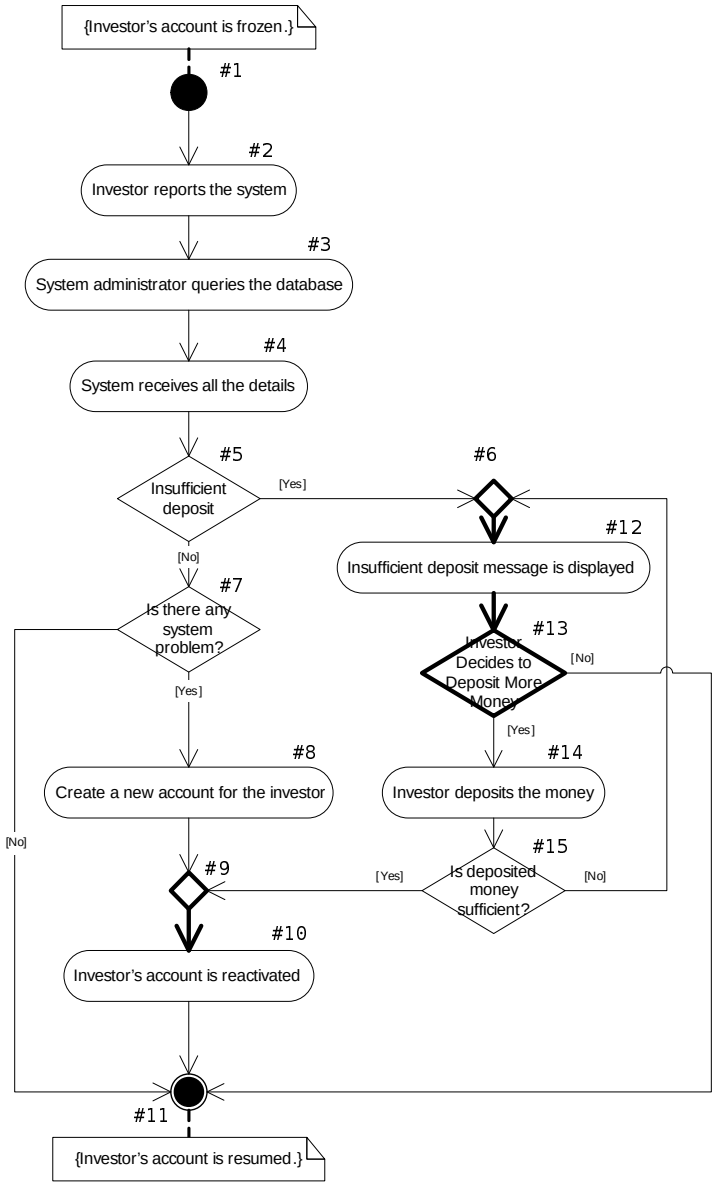


Figure 8.11: Activity from Figure 8.8 corrected to solve problems with implicit decisions and merges. The new nodes are highlighted

```
Executing activity diagram n = 100000 times...
================================================================================
There are 19 unique paths with the following probabilities, based on n = 100000 executions.
(This means that there are at least 19 paths through the diagram.)

       1.   1 2 3 4 11                                                  31585/100000 = 31.58%
       2.   1 2 3 4 12 11                                              15647/100000 = 15.65%
       3.   1 2 3 4 12 14 10 11                                         8968/100000 = 8.97%
       4.   1 2 3 4 12 14 12 11                                         4235/100000 = 4.23%
       5.   1 2 3 4 12 14 12 14 10 11                                   2385/100000 = 2.38%
       6.   1 2 3 4 12 14 12 14 12 11                                   1088/100000 = 1.09%
       7.   1 2 3 4 12 14 12 14 12 14 10 11                             677/100000 = 0.68%
       8.   1 2 3 4 12 14 12 14 12 14 12 11                             304/100000 = 0.3%
       9.   1 2 3 4 12 14 12 14 12 14 12 14 10 11                       181/100000 = 0.18%
      10.   1 2 3 4 12 14 12 14 12 14 12 14 12 11                        98/100000 = 0.1%
      11.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 10 11                 41/100000 = 0.04%
      12.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 11                 23/100000 = 0.02%
      13.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 10 11           17/100000 = 0.02%
      14.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 11           11/100000 = 0.01%
      15.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 10 11       6/100000 = 0.01%
      16.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 11       1/100000 = 0.0%
      17.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 14 10 11  1/100000 = 0.0%
      18.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 11  1/100000 = 0.0%
      19.   1 2 3 4 8 10 11                                            34731/100000 = 34.73%

Checking for Desirable Nodes...
================================================================================
The desirable nodes are: [10]
Those paths which have *not hit all desirable* nodes have been marked with an asterisk.

  *    1.   1 2 3 4 11
  *    2.   1 2 3 4 12 11
       3.   1 2 3 4 12 14 10 11
  *    4.   1 2 3 4 12 14 12 11
       5.   1 2 3 4 12 14 12 14 10 11
  *    6.   1 2 3 4 12 14 12 14 12 11
       7.   1 2 3 4 12 14 12 14 12 14 10 11
  *    8.   1 2 3 4 12 14 12 14 12 14 12 11
       9.   1 2 3 4 12 14 12 14 12 14 12 14 10 11
  *   10.   1 2 3 4 12 14 12 14 12 14 12 14 12 11
      11.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 10 11
  *   12.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 11
      13.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 10 11
  *   14.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 11
      15.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 10 11
  *   16.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 11
      17.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 14 10 11
  *   18.   1 2 3 4 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 14 12 11
      19.   1 2 3 4 8 10 11

Summary:  Have all desirable nodes, along all paths, been hit? false
          10 / 19 = 52.63% of paths do not hit all desired nodes.

Running time = 271480 ms = 271 sec = 9 min
```

Figure 8.12: Executing the activity in Figure 8.11 finds no less than 19 paths

by nodes `#6`, `#12`, `#13`, `#14`, `#15`. This cycle shows up as `12 14 12 14 ...` in the path listing because merge and decision nodes are not shown in the paths. This cycle occurs if the investor chooses to invest more money, but it is still not enough. If this activity were to be implemented, a check would have to be performed to ensure that the investor only invested a non-zero amount of money.

By examining the paths produced by ACTi, we see that there are basically five ways that this diagram executes:

1. Investor's account is frozen. There is sufficient deposit, and no system problem. The activity ends.

2. Investor's account is frozen. There is sufficient deposit, but there is a system problem. A new account is created. The account is reactivated. The activity ends.

3. Investor's account is frozen. There is insufficient deposit. The investor decides not to deposit any more money. The activity ends.

4. Investor's account is frozen. There is insufficient deposit. The investor decides to deposit (perhaps repeatedly) but it is not enough and the investor eventually decides not to deposit any more money. The activity ends.

5. Investor's account is frozen. There is insufficient deposit. The investor decides to deposit (perhaps repeatedly) until there is enough money. The account is reactivated. The activity ends.

Thus far, we have been able to examine the activity and correct some technical flaws, e.g., missing merge and decision nodes. These errors were most likely caused

by incorrect modelling; e.g., modelling an activity as if it were a more traditional flowchart. As discussed in Section 2.4.3.1 on page 29, although similar in appearance, there are subtle differences between flowcharts and UML activities.

Our interpreter allows us to execute an activity multiple times, in this case 100,000 times. By doing so, we are able to find the different paths through the activity. While it is true that these paths can be found by manually performing a depth-first search on the activity, using the interpreter is much faster, especially on complex activities.[2] In this case, we have found five different ways that an investor's account is (or is not) reinstated. This information can be passed back to the original modellers in order to validate the activity. We feel that there are actually further problems with the activity, but these areas must be confirmed by the original modellers:

- If the user has an insufficient deposit, there is no check of system status. What happens if the system has a problem *and* the deposit was not sufficient?
- If there is no system problem, and the deposit is sufficient, the activity just ends. There is actually no action performed to fix the user's account. Is it just assumed that the account is 'good'? But if that was the case, why would the account be frozen in the first place, i.e., why would the activity be executed at all?
- If a new account is created, the action to reinstate the investor's account is still performed. Is this appropriate? Or is the assumption that the second action is

[2] As discussed in Section 7.2, ACTi does not exhaustively explore the activity and then find all paths. Instead, the interpreter executes the activity multiple times, making random choices where required. If the activity is executed sufficiently often, we can be confident that all of the paths have been found. For instance, in this example, we execute the activity 100,000 times and find five major paths (two of which include loops). We are confident that these are the only paths through the activity.

actually a reinstatement of the newly created account?

- The note attached to the final node indicates that the investor's account is resumed; however, we have already shown with our paths that this is not always the case.

The original technical problems were discovered by using our interpreter to execute the activity. These high-level questions were inspired by executing the corrected activity numerous times. This example demonstrates that ACTi is useful for evaluating, and validating, activities, even when they are executed in the shallow mode.

8.3 Simple Activity

UML actions are the fundamental units of behaviour; each performs some very low-level behaviour, such as assigning a value to a variable, sending a signal, etc. That said, activities composed of UML actions are not particularly concise. In this section, we demonstrate how complex a 'simple' activity can be.

We are interested in creating an activity that performs the same behaviour as provided by this snippet of pseudocode:

```
Car c = new Car();     //A
c.year = v2005;        //B
int x = c.year;        //C
```

Figure 8.13: Pseudocode instructions

8.3.1 Sequential Activity

The activity shown in Figure 8.14 is a (mostly) sequential activity, which represents the pseudocode in Figure 8.13. A little parallelism is included because the AddStructuralFeatureValueAction requires two inputs, and it does not matter which input is provided first. As can be seen by the highlighted areas of the activity, each line of pseudocode can be represented by a few actions; these collections can be composed together sequentially.

Sections of the activity have been outlined and labelled; each represents one line of pseudocode:

(A) This part of the activity represents the creation of a new *Car* object and its assignment to the local variable c.

(B) This part of the activity represents the assignment of the value *v2005* to the new car's *year* attribute.

(C) This part of the activity represents the assignment of the new car's *year* attribute to the local variable x. This is accomplished by first retrieving the new car from the local variable c, retrieving the value of its attribute, and then assigning that value to a different local variable.

8.3.2 Concurrent Activity

One feature of activities is that they permit concurrent execution. The activity in Figure 8.14 is quite sequential but we can take advantage of the concurrency inherent in activities to potentially optimize the activity. Consider the activity in Figure 8.15.

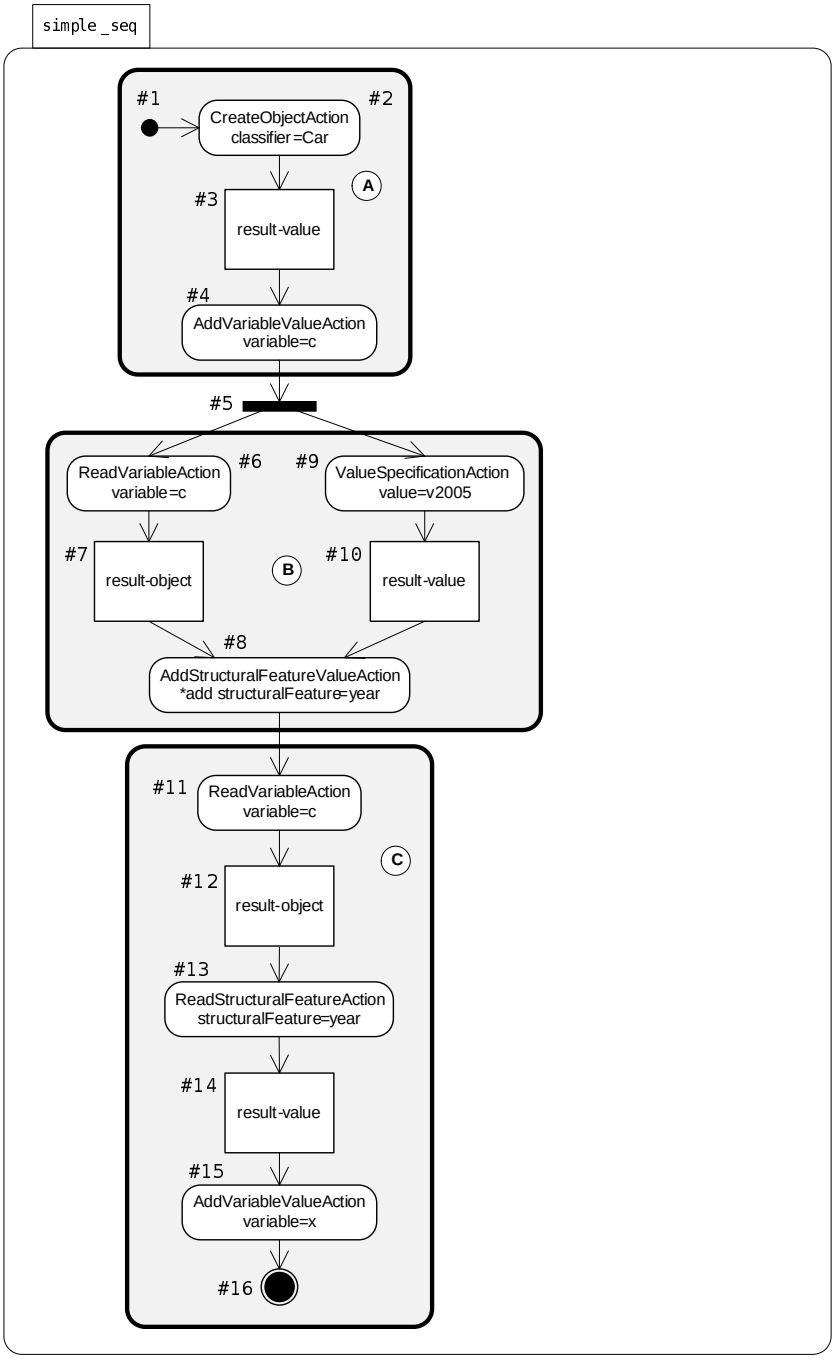


Figure 8.14: Activity representing the same behaviour as in pseudocode statements in Figure 8.13. This activity is (mostly) sequential; each highlighted segment represents one statement of pseudocode

This activity also represents the pseudocode in Figure 8.13, but takes advantage of the fact that some actions can be performed in parallel with others.

It is interesting to note how intricate and subtle a 'simple' activity can be, especially with respect to the control flow connecting the various segments described above. For instance, segment (B) does not need to wait for all of (A) to complete execution, but rather only that part of (A) concerned with actually creating the new object. Likewise, part of (C) can execute in parallel with (B), but when it comes time to actually read the attribute value, both (A) and (B) must be complete, hence the control flow connecting node `#8` and `#11`.

In fact, if this control flow were missing, the resulting activity would execute correctly some of the time, but not necessarily all of the time. Consider the modified activity in Figure 8.16.

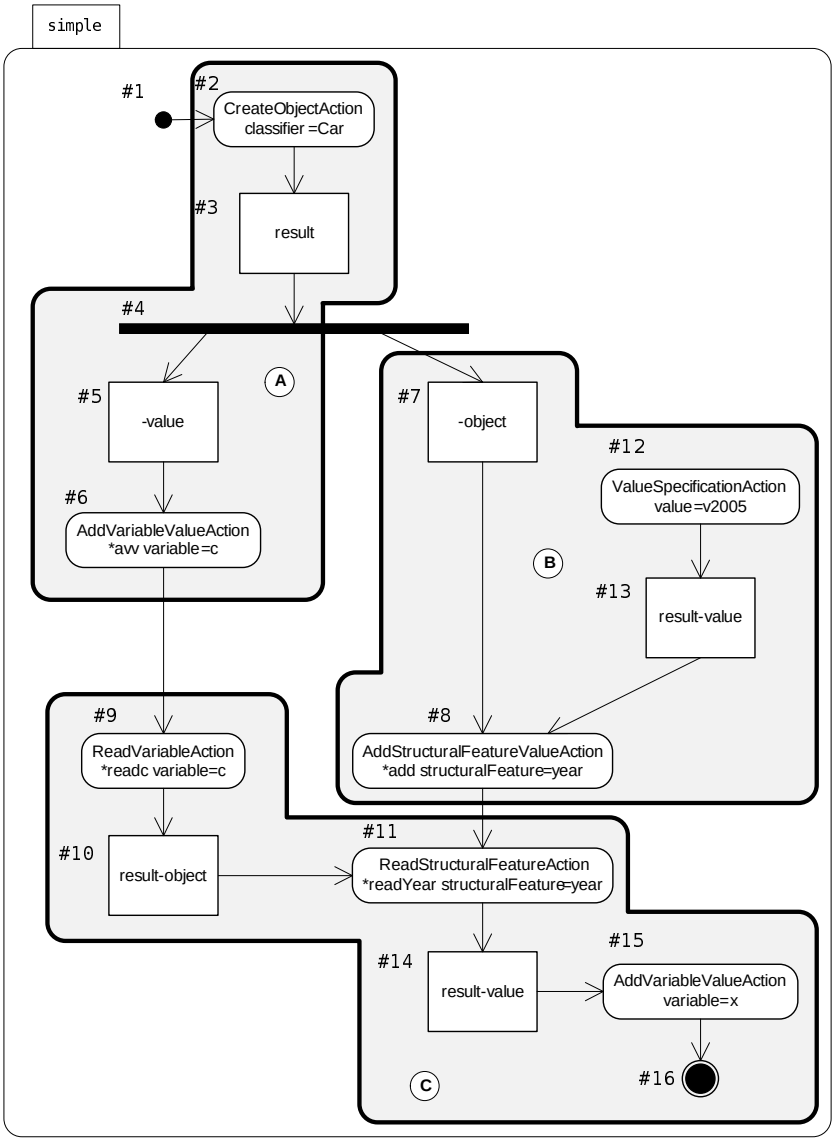


Figure 8.15: Activity representing the same behaviour as in pseudocode statements in Figure 8.13. Each highlighted segment represents one statement

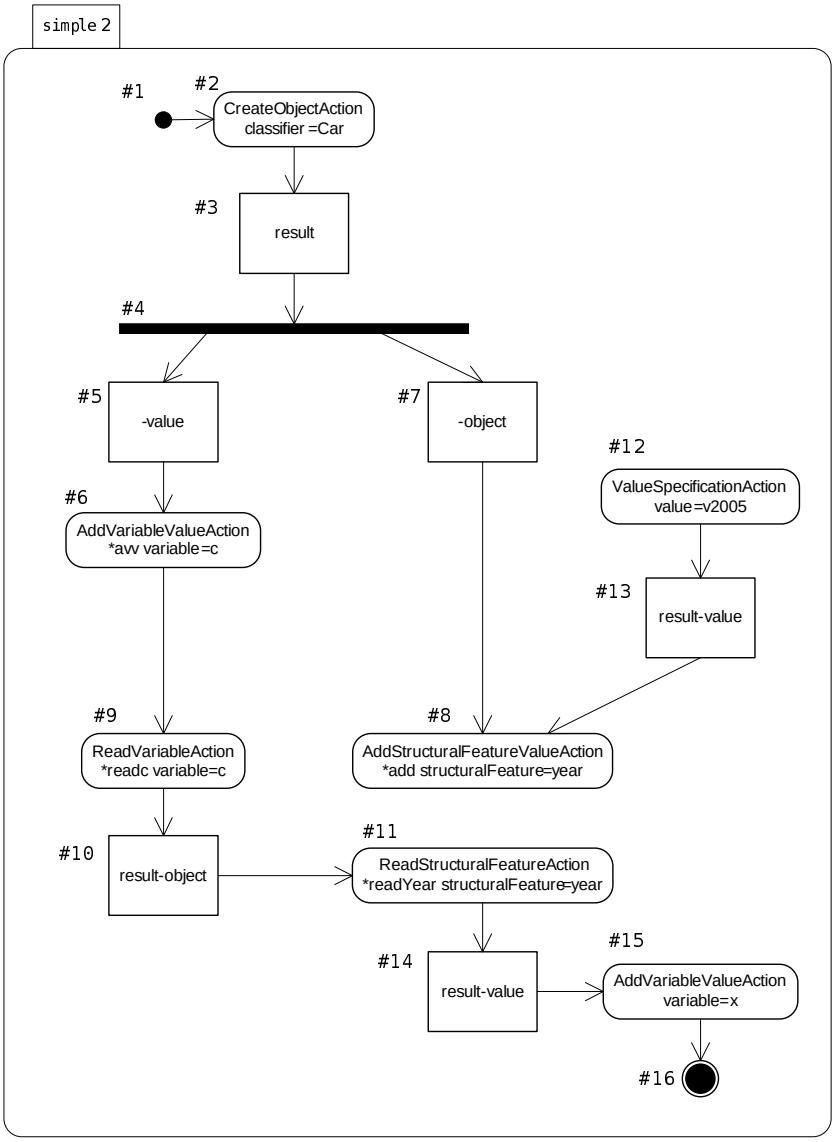


Figure 8.16: Identical activity to that in Figure 8.15, except that control flow from nodes `#8` to `#11` is missing

This activity is identical to our previous example, except that there is no control flow between nodes #8 and #11. The modeller may execute the activity[3] a few times and be comfortable that it is accurate, e.g., with the output shown in Figure 8.17.

```
Printing the order in which nodes were visited - (11)
[13, 1, 2, 4, 6, 9, 10, 8, 12, 16, 17]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #3[result] = OIDfor_Car_1
.  #5[_-value] = OIDfor_Car_1
.  #7[_-object] = OIDfor_Car_1
.  #11[result-object] = OIDfor_Car_1
.  #14[result-value] = v2005
.  #15[result-value] = v2005
.....................................................................

Running time = 290 ms = 0 sec = 0 min
```

Figure 8.17: Partial output of a successful execution of the activity in Figure 8.16

Without further testing, the modeller might be feel that the activity is valid. This is where the analysis capabilities of ACTi demonstrate their usefulness. From our original pseudocode fragment, we can deduce that $x == v2005$ at the end of the execution. We can use ACTi to check this assertion for every path found. In this case, we would execute the activity 10,000 times, and then confirm that the local variable x has indeed been set correctly in every possible execution.

When we perform this test, we find that the interpreter halts early, with the warning message shown in Figure 8.18.

[3]The mode of execution for this example is deep/random, i.e., finding a random path through the activity, with modifications to the underlying System Model universe.

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! ERROR (Bad Diagram) <Action.fixPins>
! class actions.StructuredActions.AddVariableValueAction
! The diagram does not have enough incoming information to satisfy
! all input pins.
!   Incoming: []
!   Input Pins: [value:null]
! CRASHING.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Figure 8.18: Partial output of an unsuccessful attempt to execute the activity in Figure 8.16 10,000 times

By examining this error message in conjunction with the activity, we see that there is a problem with one of the AddVariableValueAction nodes in the activity. Further exploration with the interpreter (e.g., using the debug output) allows us to determine that the error is caused when the ReadStructuralFeatureAction (node #11) is executed *before* the AddStructuralFeatureValueAction node (node #8). When this occurs, the *year* attribute of the car object has not been set before node #11 attempts to access its value. The ReadStructuralFeatureAction completes its execution successfully, but no value is placed on its output pin (because there was no value to read).[4] Then, when the AddVariableValueAction attempts to assign that (undefined) value to the local variable, it fails.

The solution to this problem is to force the ReadStructuralFeatureAction to execute *after* the AddStructuralFeatureValueAction. This restriction is enforced by connecting the two nodes with a control flow, as in our original activity in Figure 8.15.

If we run the original activity 10,000 times, we see that all of the found paths

[4]Note that it is not considered an error to read from a variable that has no value assigned; it just means that there is no value. On the other hand, trying to read from a variable that does not exist would result in an error.

result in the value *v2005* being assigned to the local variable x in every known path, as shown in the output in Figure 8.19.

```
Executing activity diagram n = 10000 times...
================================================================================
There are 15 unique paths with the following probabilities, based on n = 10000 executions.
(This means that there are at least 15 paths through the diagram.)

        1.   1 12 2 4 6 8 9 11 15 16    548/10000  = 5.48%
        2.   1 12 2 4 6 9 8 11 15 16    264/10000  = 2.64%
        3.   1 12 2 4 8 6 9 11 15 16   1691/10000  = 16.91%
        4.   1 2 12 4 6 8 9 11 15 16    254/10000  = 2.54%
        5.   1 2 12 4 6 9 8 11 15 16    125/10000  = 1.25%
        6.   1 2 12 4 8 6 9 11 15 16    869/10000  = 8.69%
        7.   1 2 4 12 6 8 9 11 15 16    157/10000  = 1.57%
        8.   1 2 4 12 6 9 8 11 15 16     61/10000  = 0.61%
        9.   1 2 4 12 8 6 9 11 15 16    397/10000  = 3.97%
       10.   1 2 4 6 12 8 9 11 15 16    223/10000  = 2.23%
       11.   1 2 4 6 12 9 8 11 15 16     94/10000  = 0.94%
       12.   1 2 4 6 9 12 8 11 15 16    305/10000  = 3.05%
       13.   12 1 2 4 6 8 9 11 15 16   1118/10000  = 11.18%
       14.   12 1 2 4 6 9 8 11 15 16    572/10000  = 5.72%
       15.   12 1 2 4 8 6 9 11 15 16   3322/10000  = 33.22%

Checking assertion x == v2005...
================================================================================
Those paths for which the assertion does *not hold* are marked with an asterisk.

        1.   1 12 2 4 6 8 9 11 15 16
        2.   1 12 2 4 6 9 8 11 15 16
        3.   1 12 2 4 8 6 9 11 15 16
        4.   1 2 12 4 6 8 9 11 15 16
        5.   1 2 12 4 6 9 8 11 15 16
        6.   1 2 12 4 8 6 9 11 15 16
        7.   1 2 4 12 6 8 9 11 15 16
        8.   1 2 4 12 6 9 8 11 15 16
        9.   1 2 4 12 8 6 9 11 15 16
       10.   1 2 4 6 12 8 9 11 15 16
       11.   1 2 4 6 12 9 8 11 15 16
       12.   1 2 4 6 9 12 8 11 15 16
       13.   12 1 2 4 6 8 9 11 15 16
       14.   12 1 2 4 6 9 8 11 15 16
       15.   12 1 2 4 8 6 9 11 15 16

Summary:  Is the assertion x == v2005 true for every path? true
          0 / 15 = 0.0% of paths do not satisfy the assertion

Running time = 56151 ms = 56 sec = 1 min
```

Figure 8.19: Partial output of executing the activity in Figure 8.15 10,000 and asserting that $x == v2005$ for each path

This example highlights the fact that actions are designed to be "fundamental units of behaviour" [59, §6.3.2]. Interestingly enough, the UML specification also likens events to "executable instructions in traditional programming languages" [59, §6.3.2]. However, it seems that actions are in fact slightly more low-level than executable statements; perhaps more like assembly-level. For instance, the assignment `c.year = v2005` requires no less than three actions: one each to retrieve the entities referenced by `c` and `v2005`, and one to actually assign the value.

8.4 USE Visualization

As mentioned in the previous chapter, it is possible to use the UML-based Specification Environment (USE) [80] to visualize state changes in the System Model as an activity is executed.[5] This visualization is not dynamic in that it is not performed as the activity is executed. Rather, as the activity executes, a series of files are created (like a trace) that are then used as input to recreate the state changes.

For this example, we will use the activity from Figure 8.15. Executing the activity provides the following random path:

```
[12, 1, 2, 4, 6, 9, 8, 11, 15, 16]
```

The interpreter also creates a series of files required to perform the visualization. We show partial snapshots of the System Model universe at the following points of the execution:

1. State `s0`. This is the initial state of the System Model, after it has been initialized and before the execution starts. See Figure 8.20.

[5] The activity must be executed in the deep mode.

2. State `s3`. This is the state of the System Model universe after the CreateObjectAction has been executed. The data store now refers to a new object identifier. See Figure 8.21.

3. State `s5`. This is the state after the first AddVariableValueAction has been executed. The control store now shows a mapping between the local variable c and its new value. See Figure 8.22.

4. State `s7`. This is the state after the AddStructuralFeatureValueAction has been executed. The data store now contains a mapping for the car's *year* attribute. See Figure 8.23.

5. State `s9`. This is the state after the second AddVariableValueAction has been executed. The control store now shows a mapping between the local variable x and its new value. See Figure 8.24.

The other states encountered during the execution are uninteresting in that no changes have been made, except to the program counters and threads, which we have elided for these examples. It should be noted that interpreting these visualizations requires an in-depth knowledge of the structure of the System Model universe.

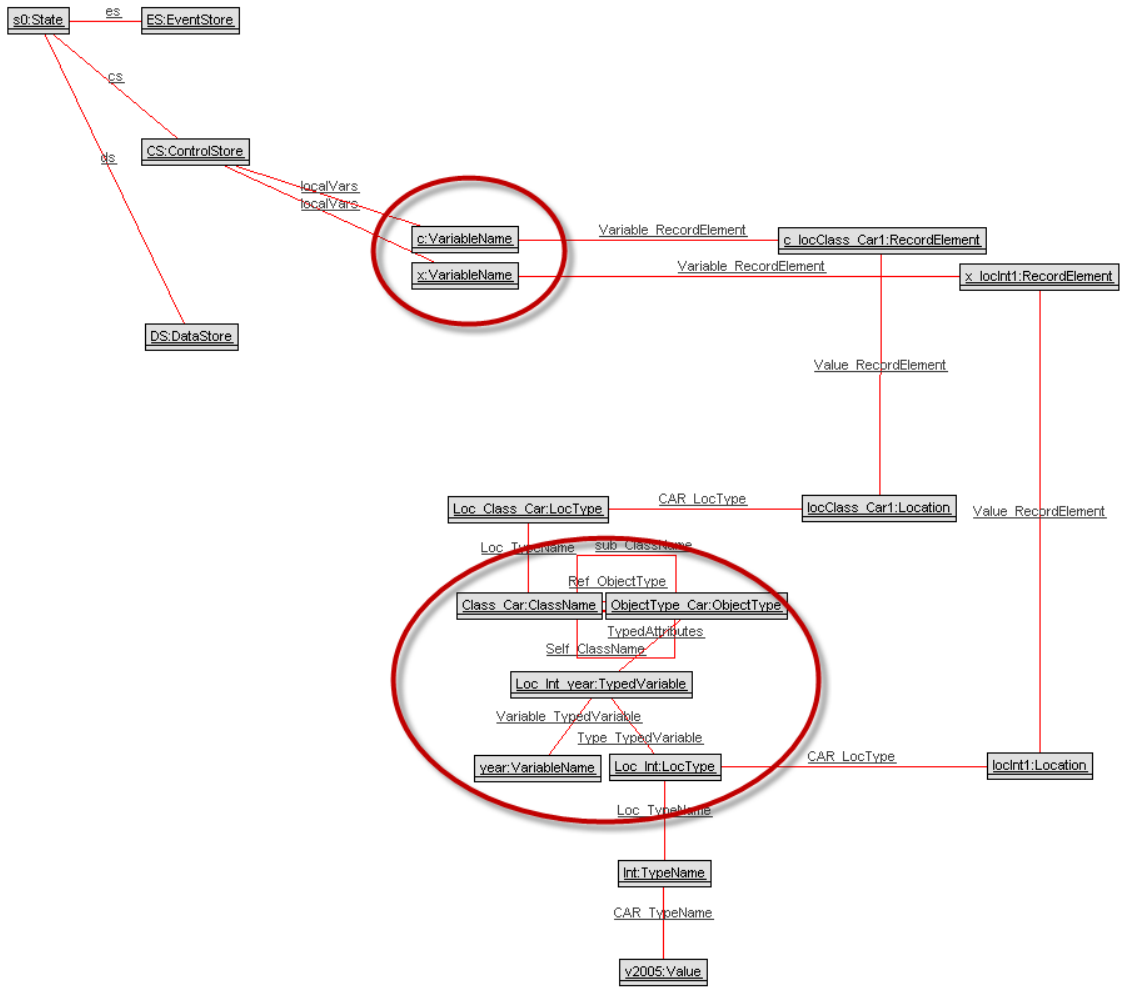


Figure 8.20: Partial snapshot of System Model universe at state `s0`, before the activity in Figure 8.15 begins execution. The state is composed of the data store, control store and event store. At this stage, only the control store has any information, i.e., the two local variables c and x (shown in a circle). The types of these variables can be deduced by following the *VariableName* through the associated *RecordElement*. In this case, variable c is of type *Car*. The elements in the large ellipse together represent the structure of the *Car* class, i.e., a *ClassName*, a related *ObjectType*, which contains a *TypedVariable*, composed of a *VariableName* and a *Location*. The value *v2005* is shown at the bottom of the diagram; it is in the carrier set of the *TypeName Int*. Other values exist in the universe, but have been elided

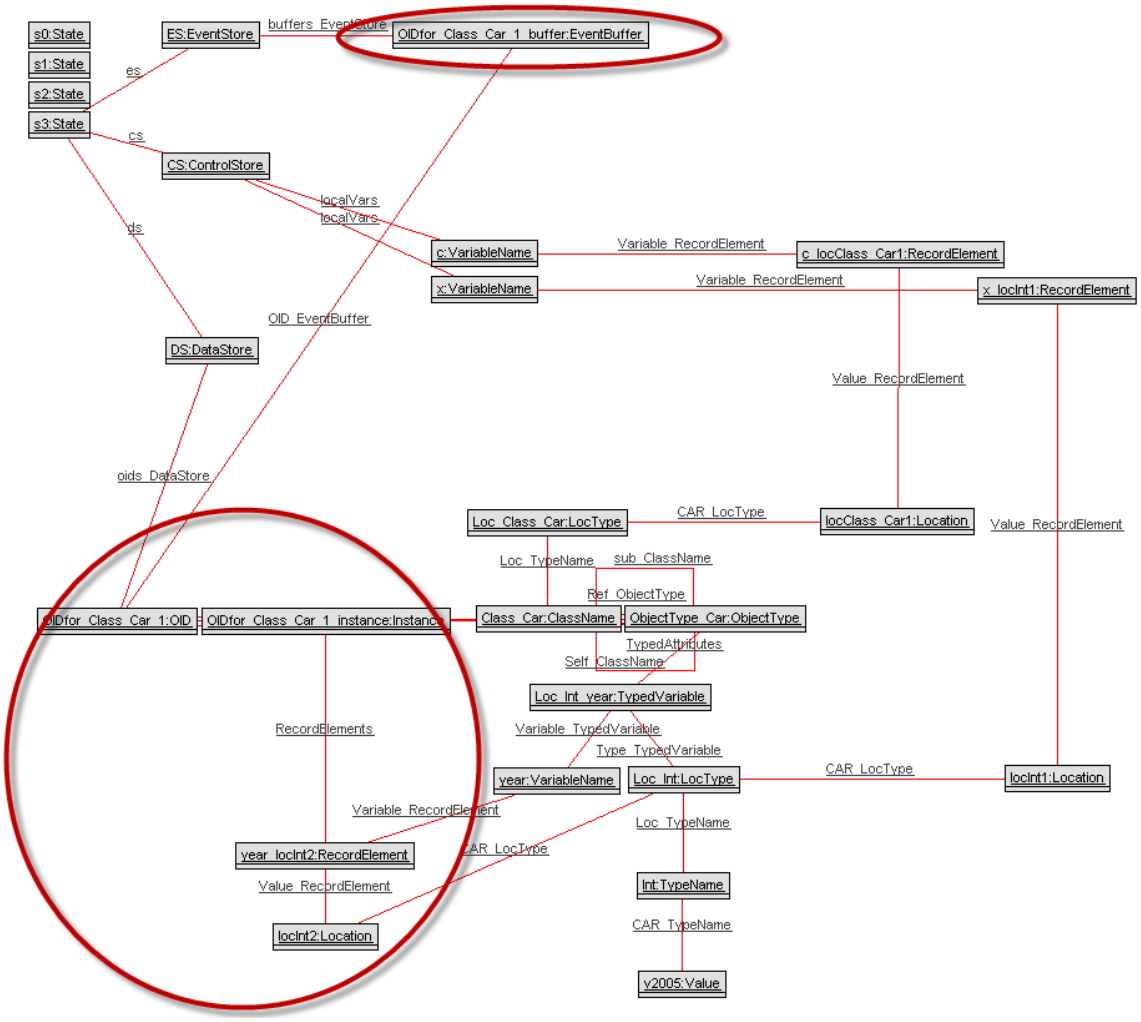


Figure 8.21: Partial snapshot of System Model universe at state s3, after the CreateObjectAction has been executed. The large circle shows those elements that compose the new *Car* object. These objects consist of an *OID* for the object identifier, a Instance for the dereference of the *OID*, and a RecordElement, itself composed of a VariableName and a Location. This new object is referenced by the data store. The smaller ellipse shows that there is an event buffer, referred to by the event store

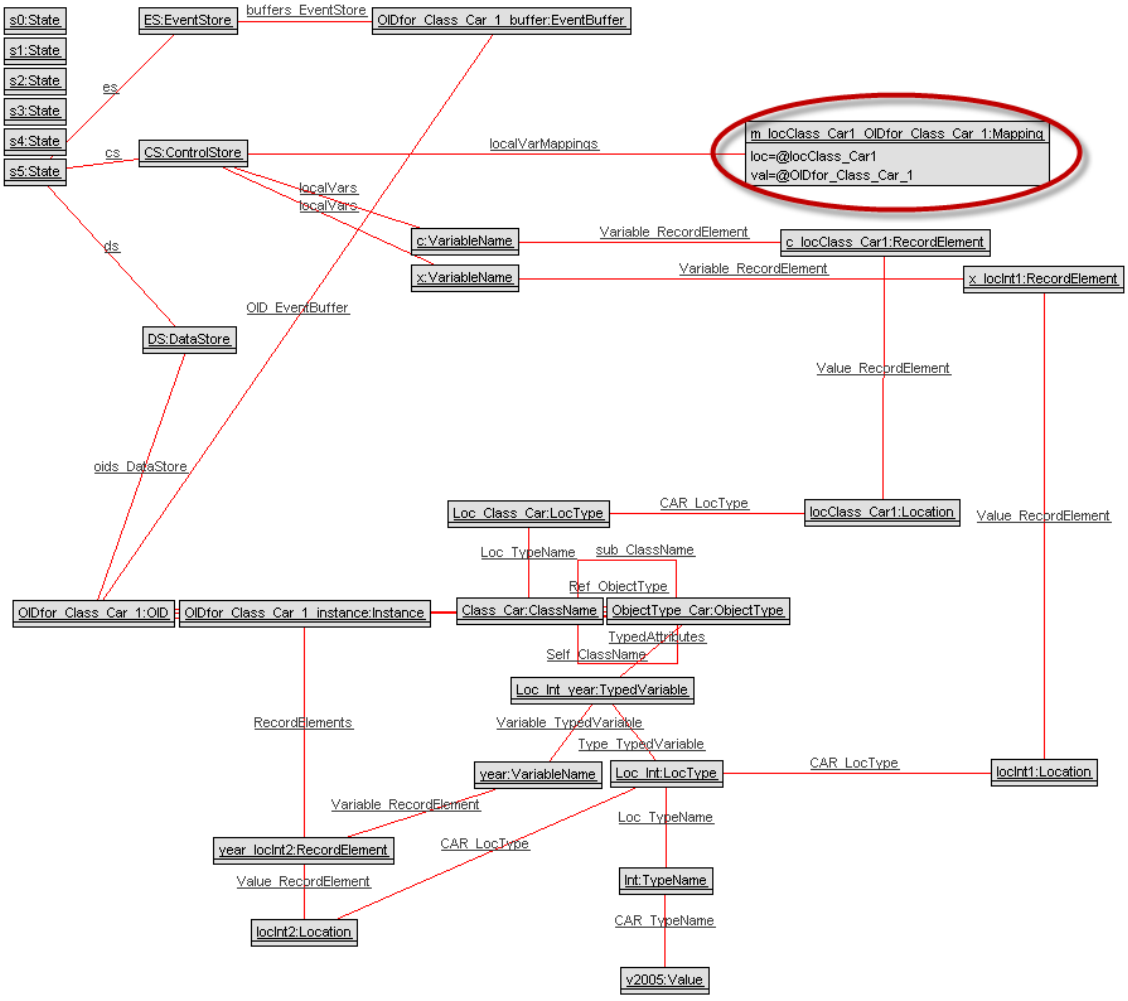


Figure 8.22: Partial snapshot of System Model universe at state s5, after the first AddVariableValueAction has been executed. The ellipse shows a new mapping in the control store; this mapping shows that the local variable c is now associated with the new car object that was just created

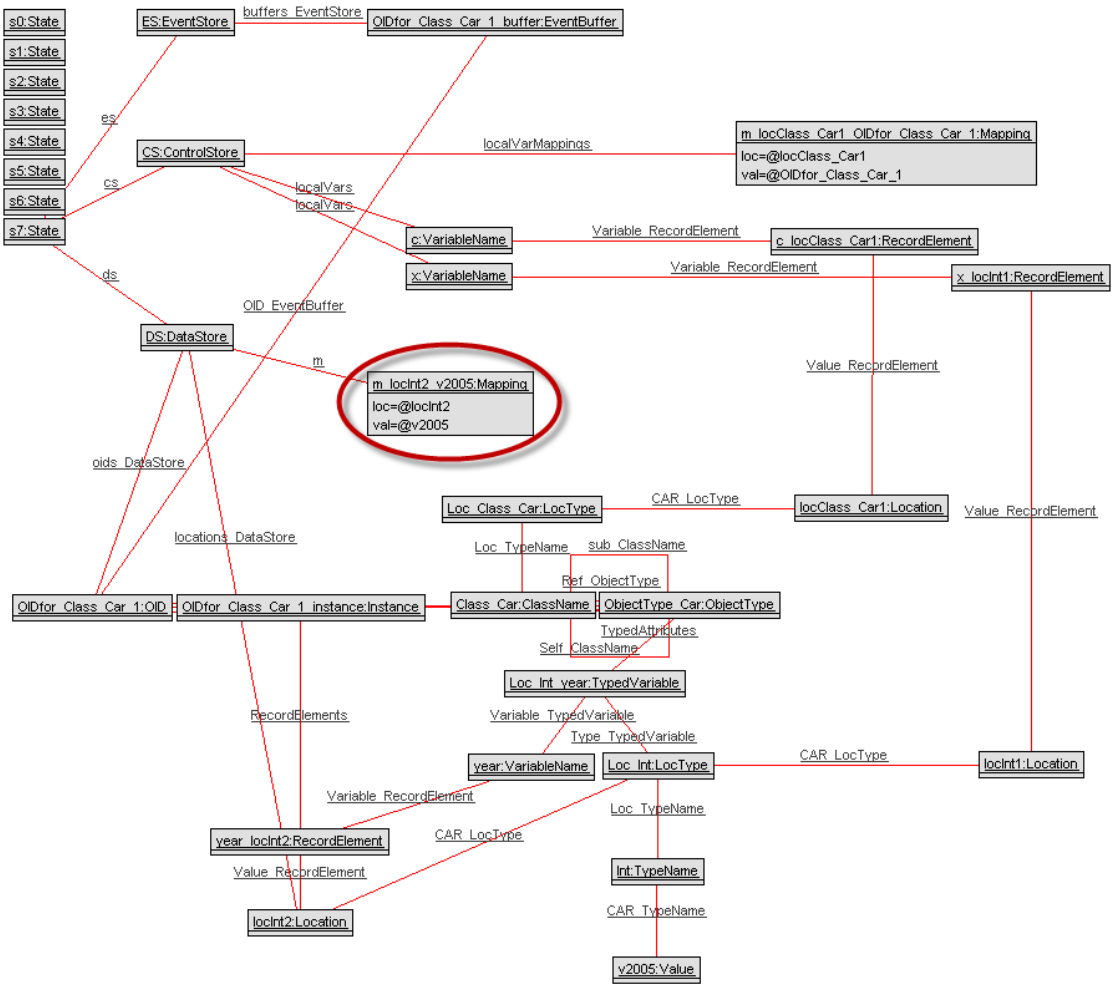


Figure 8.23: Partial snapshot of System Model universe at state `s7`, after the AddStructuralFeatureValueAction has been executed. The ellipse shows a new mapping in the data store; this mapping shows that the new car object's *year* variable is mapped to the value *v2005*

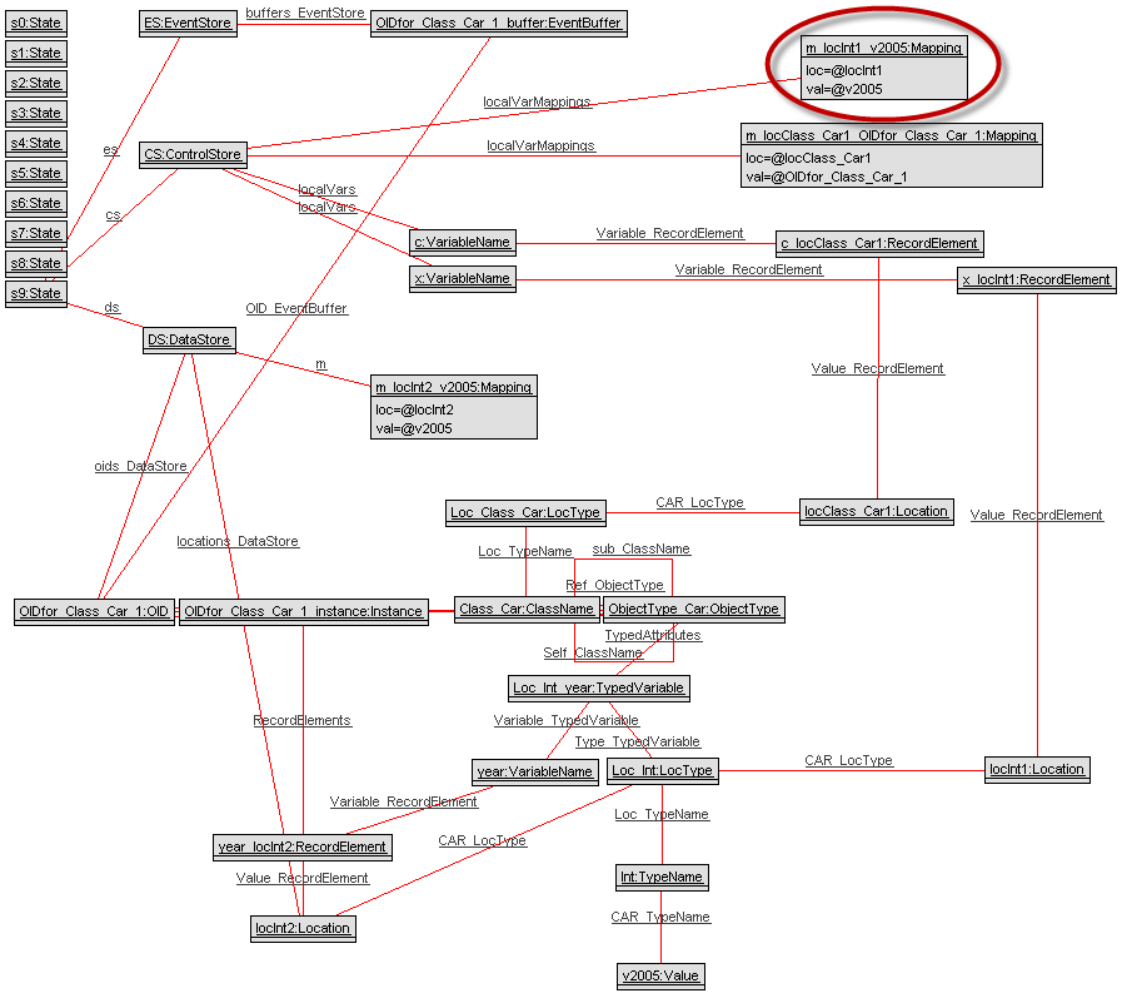


Figure 8.24: Partial snapshot of System Model universe at state `s9`, after the second AddVariableValueAction has been executed. The ellipse shows a new mapping in the control store; this mapping shows that the local variable x is now associated with the value *v2005*, which is the value of the new car object's *year* attribute

8.5 Accept Call vs. Accept Event

In Section 5.3.2.1, we discussed how, for various reasons, we have chosen to use an AcceptCallAction to receive calls from all CallOperationActions, whether or not they

were synchronous. It is technically possible to use an AcceptEventAction to receive an asynchronous call. In this section, we compare how the activities would look if we used the AcceptEventAction instead.

Consider our original CallOperationAction example, seen in either Figure 5.12 on page 131 or Figure B.37 on page 303. If we were to make that call asynchronous, the main activity would look like that in Figure 8.25 below.

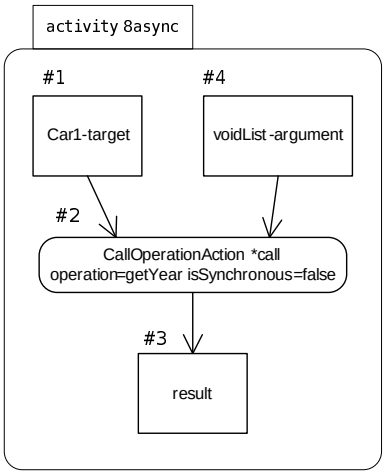
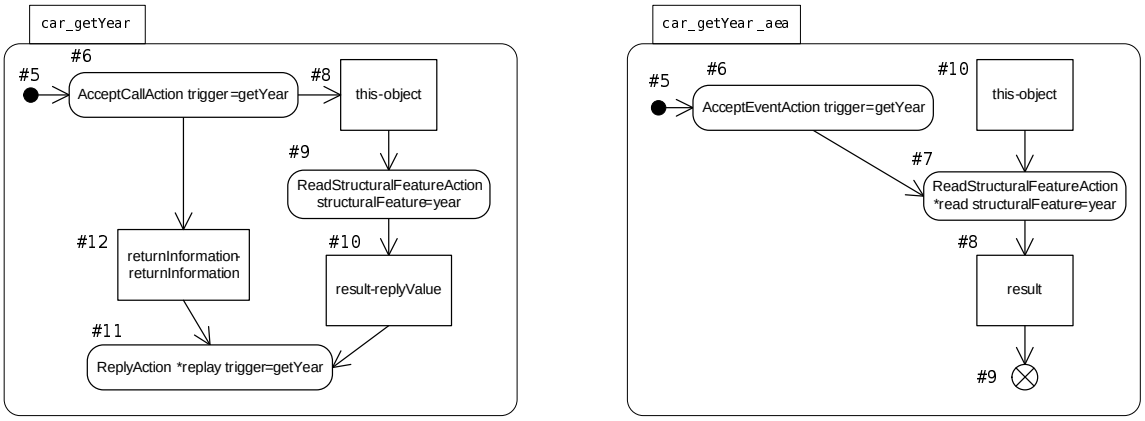


Figure 8.25: Asynchronous version of 'activity8' from Figure 5.12 on page 131 (or Figure B.37 on page 303). The activity is identical, except for the annotation on the CallOperation-action

While the main activity is almost identical (except for the annotation on the CallOperationAction), the called activity[6] is much different. Instead of using the AcceptCallAction (which must be paired with a ReplyAction), we use the AcceptEventAction. We show both options below: Figure 8.26(a) shows the activity representing the

[6]Typically, we would not call a 'get' method asynchronously, because the result would be desired. However, this example is just for illustration.

getYear behaviour using the AcceptCallAction and Figure 8.26(b) shows the same behaviour using the AcceptEventAction. In the latter case, there is no ReplyAction whose purpose it is to return a value to the caller. The AcceptEventAction is only used to receive asynchronous calls, so no return is ever necessary. In addition, according to the UML specification, the AcceptEventAction cannot be used to pass parameter arguments to its targets when receiving a call from a CallOperationAction.



(a) Using an AcceptCallAction to receive a synchronous (or asynchronous) call

(b) Using an AcceptEventAction to receive an asynchronous (but not a synchronous) call

Figure 8.26: Comparing use of AcceptCallAction vs. AcceptEventAction in invoked behaviour

8.6 Inheritance and Dynamic Binding

As discussed in Chapter 3, the System Model provides support for subclassing, or inheritance. Our interpreter has the ability to perform dynamic binding. When a behaviour is invoked, the most appropriate behaviour is chosen by the interpreter. Note that, according to the UML specification, the "mechanism for determining the

method to be invoked as a result of a call operation is unspecified" [59, §11.3.10]. Our interpreter chooses the most specialized behaviour, as in Java.

In this section, we demonstrate how ACTi handles dynamic binding. The activity in Figure 8.27 is our main behaviour; we simply invoke a call to the *renew* operation on our car object.

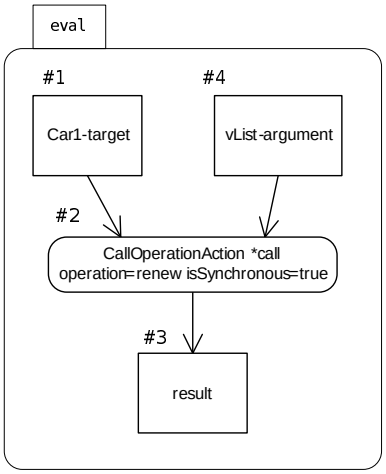


Figure 8.27: Activity to invoke *renew* operation on *Car*

The cd file in Figure 8.28 shows that we are defining two class names, one of which specializes the other. Both class names support the renew operation. Figure 8.29 contains the obj file for our example, in which we set up a user-defined initial state containing a car object and a list for the arguments to be passed into the CallOperationAction.

Each of the behaviours defined in the cd file are represented by an activity; these are shown in Figures 8.30 and 8.31. Notice that the two behaviours are very different; in fact, the *renew* operation on the *Veh* class is essentially an empty activity.

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

class Veh
 attr year:Int
 op renew(newYearV:Int):Void vehrenew.act

class Car extends Veh
 op renew(newYearC:Int):Void carrenew.act

value vList:Collect
```

Figure 8.28: cd file

```
obj Car1:Car

collection vList
  v2005
```

Figure 8.29: obj file

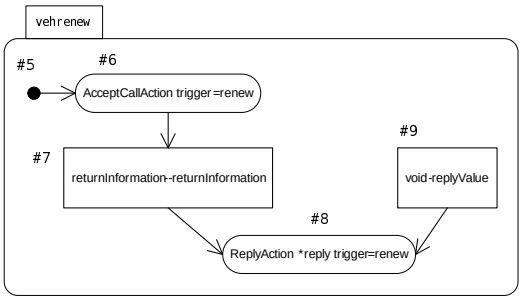


Figure 8.30: Activity for *Veh.renew*

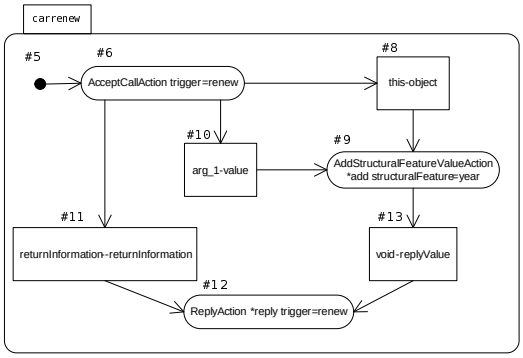


Figure 8.31: Activity for *Car.renew*

By using an empty behaviour for the *Veh* class, we can confirm that our interpreter is indeed dynamically binding the call to the *Car* class's version of the *renew* operation. In the output shown in Figure 8.32, we see that at the end of every found path through the activities, the value of *Car1.year* has indeed been set, which is the behaviour implemented by the *Car.year* operation.

```
Executing activity diagram n = 10000 times...
================================================================================
There are 2 unique paths with the following probabilities, based on n = 10000 executions.
(This means that there are at least 2 paths through the diagram.)

        1.   1 4 2 5 6 9 12 2 3   5137/10000  = 51.37%
        2.   4 1 2 5 6 9 12 2 3   4863/10000  = 48.63%

Printing the value of Car1.year...
================================================================================

        1.   1 4 2 5 6 9 12 2 3                                  v2005
        2.   4 1 2 5 6 9 12 2 3                                  v2005

Running time = 53937 ms = 53 sec = 1 min
```

Figure 8.32: Partial output from executing the activity in Figure 8.27 10,000 times. For every path found, the value of *Car1.year* has been set to *v2005*, indicating that the *Car.year* operation was executed

On the other hand, if we remove the definition of the *renew* operation from the *Car* class's description, the behaviour implemented in the *Veh* class would be executed. As we can see by the output in Figure 8.33, when this occurs, the value of *Car1.year* is never set.

```
Executing activity diagram n = 10000 times...
================================================================================
There are 6 unique paths with the following probabilities, based on n = 10000 executions.
(This means that there are at least 6 paths through the diagram.)

        1.   1 4 2 5 6 9 8 2 3   1216/10000  = 12.16%
        2.   1 4 2 5 9 6 8 2 3   1252/10000  = 12.52%
        3.   1 4 2 9 5 6 8 2 3   2494/10000  = 24.94%
        4.   4 1 2 5 6 9 8 2 3   1251/10000  = 12.51%
        5.   4 1 2 5 9 6 8 2 3   1218/10000  = 12.18%
        6.   4 1 2 9 5 6 8 2 3   2569/10000  = 25.69%

Printing the value of Car1.year...
================================================================================

        1.   1 4 2 5 6 9 8 2 3                                    null
        2.   1 4 2 5 9 6 8 2 3                                    null
        3.   1 4 2 9 5 6 8 2 3                                    null
        4.   4 1 2 5 6 9 8 2 3                                    null
        5.   4 1 2 5 9 6 8 2 3                                    null
        6.   4 1 2 9 5 6 8 2 3                                    null

Running time = 46126 ms = 46 sec = 1 min
```

Figure 8.33: Partial output from executing the activity in Figure 8.27 10,000 times. In this case, no behaviour has been defined in the *Car* class; therefore, the *Veh.renew* operation is executed. For every path found, the value of *Car1.year* has not been set; the *Veh.renew* operation performs no changes

Readers might be curious as to why there are more paths when the *Veh.renew* behaviour is executed, especially since that particularly activity contains fewer nodes. Although the *Car.renew* activity contains more nodes, these nodes are actually more tightly constrained by control flows.

Chapter 9

Related Work

Actions were originally introduced into UML 1.5, and in UML 2.0, activities were brought more in line with these actions to produce a more procedural model [67]. Little related work exists on the semantics of current UML actions. For instance, [55] provides a semantics for an older version of UML actions, but is obsolete because pins have since been introduced. [39] creates a concrete syntax for a surface language suitable for UML 1.5 actions, based on OCL expressions.

The fact that activities have also substantially changed with the introduction of UML 2 means that all previous research into the semantics of activities is no longer relevant to our purposes. However, there has been some research on defining the semantics of UML 2 activities based on several formalisms, such as Abstract State Machines [69], Petri nets [77, 70], and dynamic metamodelling [29]. Other research has focused on model checking activities [30]. Finally, activities are becoming quite popular in terms of business process modelling [68, 82]. However, little of this research on activities actually supports individual UML actions.

9.1 Surface Language and Executable UML

As discussed in Section 2.3.3, the UML specification describes the *action semantics*, i.e., it describes the effects of a set of "fairly low-level actions sufficient to describe behavior" [67] but it does not specify a formal semantics or a concrete syntax [33] for actions. The action semantics could be viewed as an assembly language needing a surface action language that would encompass both primitive actions and UML's behavioural control mechanisms. The generation of code from models, or the ability to execute models directly, requires the use of an action language to augment UML. Indeed, any executable UML foundation must be accompanied by such an action language.

As there is no standard action language, many have been created, both for commercial use (where the focus is mainly on executable UML) and academic research (where the focus tends to be on examining semantics or analyzing UML models). We are interested in semantics, specifically of the individual UML actions; in order to examine the execution of actions, we too have required the use of an action language. UML activities are a natural fit as our surface language, as they encompass both actions and control mechanisms.

9.1.1 Academic Initiatives

Academic action languages tend to be part of tools focused on the analysis and/or testing of models. Jumbala is an action language for state machines, based on Java, includes an interpreter and is part of a prototype tool suite for analyzing UML models [27]. OxUML is an OCL-based action language that supports actions with side effects. The result is an executable UML, supported by a UML virtual machine [45].

[36] suggests a profile for UML that provides "complete and precise Aspect-Oriented behavior modeling", including UML's action semantics. After weaving, the complete model can be executed in order to observe the behaviour of the modelled aspects. This initiative is supported by the recently published Pópulo tool [35, 34]. An Eclipse plugin, Pópulo is a "model debugger that interprets the UML action language" [35].

ActiveChartsIDE [69] is a well-developed plugin for Microsoft Visio, with an interpreter that supports the simulation and debugging of activity diagrams. A Java-like action language, JAL, has been created to assist in the testing of UML models [23]. JAL supports actions such as call operation actions, create/destroy objects/links and read/write variable/links. A prototype UML Animator and Tester (UMLAnT) tool automates the test execution and animation [22].

Another approach, [64], focuses not on executing UML models, but on creating a generic action language to counteract the myriad of other more specific action languages in existence. The result is a language based on +CAL [48], an algorithmic language that can be translated to a TLA+ [47] specification and then model checked. This generic action language can be used to "write complete and unambiguous function descriptions of the actions of operations and states" [64].

9.1.2 Commercial Initiatives

There are several commercial tools that support executable UML; each of these includes their own proprietary action language, some of which predate, and indeed, were inspirations for, the UML action semantics. xUML [65] is a subset (or profile) of UML, used by Kennedy-Carter in their iUML tool, which incorporates the Action Specification Language (ASL) and permits the construction of executable models.

Another initiative, xtUML [50], contains the Object Action Language (AL), and is implemented by Mentor Graphics' BridgePoint tool suite. Telelogic [78] also has a UML modelling tool with its own action language, this one with Java-like syntax. Finally, the Kabira [46] Design Center tool uses an action language for executable model development.

There also exists a plugin for IBM's Rational Modeling tools, which "enables the execution, debugging and testing of UML models" [25]. This debugger supports state machines and activities and can even be guided by the user when an incomplete model is executed. This execution engine does not specifically support UML actions; it uses Java as an action language.

Interestingly, the final submission for the Action Semantics [56], which was incorporated into UML 2.0, contains a comparison of action languages of three of the submission's supporters, namely ASL (from Kennedy-Carter), AL (from Mentor Graphics) and Kabira AS (from Kabira). Figure 9.1 shows how each of these languages represents the creation of an object and the assignment of that object to a local variable. Each of the action languages uses one (or two) textual lines to represent the behaviour.

```
ASL:
new_customer = create Customer
AL:
create object instance new_customer of Customer;
Kabira AS:
declare Customer new_customer;
create new_customer;
```

Figure 9.1: Creation of an object and assignment to a local variable, represented in the action languages of three commercial modelling tools. Figure from [56, Figure 33]

The activity that we would use to represent this behaviour is shown in Figure 9.2.

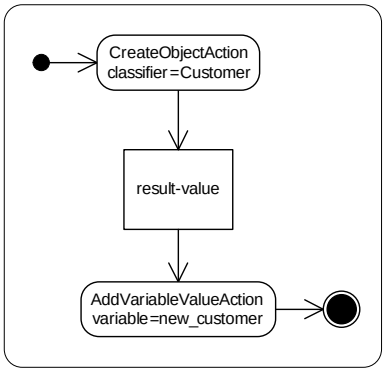


Figure 9.2: Our activity representation of the behaviour shown in Figure 9.1

9.2 Executable UML Foundation

The OMG has been presented with one submission [63] in response to their RFP for an Executable UML Foundation. This submission, which is in the process of being finalized by the OMG [62], defines the Foundational Subset for Executable UML (fUML) and is supported by many of the vendors of current modelling technologies, including IBM, Kennedy Carter and Mentor Graphics. Our research into the semantics of actions is most closely related to this submission. There are many similarities between our approach and that suggested by the fUML submission. For instance, at the most basic level, both approaches declaratively define actions and then define the execution of actions/activities in a more operational manner. While we map our actions to the underlying System Model universe, each element of the fUML is mapped to a base UML (bUML); the formal semantics of the bUML is expressed in first-order logic axioms [63, §10.1]. These axioms are expressed in the Process

Specification Language (PSL) [42], defined in the Common Logic Interchange Format (CLIF) [43]. While we discuss our interpreter in general terms, with examples, their *execution model* is itself an fUML model; it defines the "operational procedure for the dynamic changes required during the execution of a fUML model" [63, §8.1].

Overall philosophy aside, both approaches implicitly support concurrent execution, but do not require that threads be physically executed in parallel. Similarly, both approaches assume that communications are reliable, that events are dispatched in a FIFO basis (although alternate scheduling could be provided) and that the result of polymorphic choice of operations depends on the "context and target of the invocation" [63]. In terms of actions, neither approach supports exceptions (and thus the RaiseExceptionAction), the OpaqueAction or the UnmarshallAction. In terms of activities, neither approach provides support for central buffer nodes, activity groups, or partitions.

Our approach can be seen as both more shallow, and more broad, than the fUML approach. For instance, the fUML submission is much more detailed and covers a larger subset of UML, e.g., defining the exact subset of UML supported, including classes, types, expressions, actions, activities, etc. We focus exclusively on UML actions and activities, using only as much static structure as required to describe actions. We also use the System Model as our domain, while the fUML *foundation* rests on a slightly smaller bUML *base*.

On the other hand, because we are not interested in producing a 'compact' subset of UML, we are able to cover different elements, such as call events. These, along with AcceptCallAction and subsequent ReplyAction, are not supported by fUML. Actually, no receive events are supported by fUML as only asynchronous signals are

permitted [63, §7.3.3.1]. In other words, only the asynchronous CallOperationAction and CallBehaviorAction, as well as the SendSignalAction, are supported as invocation actions. We, however, support the SendObjectAction and BroadcastSignalAction, as well as the synchronous versions of the CallOperationAction and CallBehaviorAction. In terms of activities, fUML does not include the FlowFinalNode or local variables (and hence the variable actions). They do, however, support loop and condition nodes, as well as most association and link actions.

9.3 Summary

Our research is unique in that it:

- Uses the System Model as its semantic domain. The System Model has been previously to describe the semantics of interactions [16] and state machines [17].
- Focuses specifically on UML actions. We have formally defined the behaviour of two-thirds of UML actions.
- Uses UML activities as the action language.
- Provides a working interpreter for UML actions and activities.

With respect to the semantics of actions, the closest related work is the fUML submission to the OMG's RFP for an Executable UML foundation. That submission can be seen as more complete and polished, and backed up by a standard formalism, i.e., both the PSL [42] and CLIF [43] are defined by ISO standards. On the other hand, the fUML submission is a joint effort sponsored by several of the modelling powerhouses, e.g., IBM, Kennedy Carter, Mentor Graphics, among others. Both the

fUML submission and our research provide for an executable subset of UML (i.e., actions and activities); neither can be considered more 'right' than the other, since both are based on the behavioural semantics defined in the UML specification.

With respect to the interpretation or execution of actions, the Pópulo tool is most related to our interpreter. Both tools read in a textual form of an activity composing UML actions. Both tools support user-defined behaviour, and interpret the actions directly. Neither tool supports animation of the activity, although the Pópulo tool has the advantage of being an Eclipse plugin and more graphical by nature. On the other hand, Pópulo does not support the shallow execution of activities, or analysis in general.

Chapter 10

Conclusion

One of the main notational contexts in which model-driven software development has been studied is the Unified Modeling Language (UML), the *de facto* standard in software modelling. The current trend is not just towards the use of models, but the use of executable models. In 2006, the OMG issued a Request for Proposal [58], soliciting the definition of an Executable UML Foundation, which would be a computationally complete and compact subset of UML, with a fully specified executable semantics. Any such foundational subset would require the definition of an action language, preferably one which has been formally specified.

We formally define the execution semantics of two-thirds of the UML actions. This definition is expressed in terms of state changes to a global state machine that represents an executing UML model. In addition, we have created ACTi, an interpreter for UML actions and activities. This interpreter was designed in accordance with the runtime semantics described in the UML specification and provides analysis capabilities that have been successfully used to identify problems even in published activity diagrams.

In this chapter we detail our specific contributions, as well as discuss potential future work.

10.1 Contributions

10.1.1 Implementing the System Model

One major contribution of our research is the implementation of the System Model. Although the purpose of the System Model itself is to represent the structure of UML, we found it useful to use class diagrams to model the structure of the System Model universe. Then, we implemented this universe in Java. There is not much behaviour described in the System Model *per se*; however, we have implemented methods for creating, editing and accessing the universe. These methods allow us to build a portion of the universe for use as the structural foundation over which an activity (composed of UML actions) executes.

By implementing the System Model, we were able to truly understand it, e.g., the symmetric nature of the type and value hierarchies as shown in Figure 3.2, the difference between references and locations, the relationship between *Operation*, *Method* and *MethodName*, etc. We have also been able to make several observations about the System Model, which are detailed below.

10.1.2 Mapping Actions

We have formally defined 24 concrete actions (representing two-thirds of all UML actions) by mapping their execution to state changes in an underlying System Model

universe. The effects of individual actions are represented in terms of pre- and post-conditions.

10.1.3 Action/Activity Interpreter

We have created ACTi, an interpreter for UML actions and activities that can be used to execute (and analyze) activities. ACTi supports a large subset of UML activities, including the following concepts: sequential and concurrent composition, fork, join, merge, and decision (including the parsing and evaluation of basic guards[1]) control nodes. We do not support partitions, exceptions, interruptible regions, structured activity nodes (e.g., loop and conditional), buffer nodes, weights on edges, or multiplicities other than 1..* on pins.

The interpreter can be used to execute and analyze activities composed of UML, or even user-defined, actions. The execution can be performed in either a deep or shallow fashion. Deep executions modify the state of the underlying System Model. In addition, a user-defined initial state may be specified before the activity execution commences. Execution of an activity can either be random (resulting in a non-deterministic path through the activity) or guided (the user provides a chosen path). In addition, the random execution can be performed multiple times, resulting in a list of possible paths through the activity.

While configuring the execution of an activity, the user can request various types of analysis. When performed with multiple random executions, these analyses are akin to limited model checking; assertions are checked for every one of the paths that the interpreter discovers. Analysis techniques include: checking for (un)desirable

[1]ACTi can interpret guards involving object attributes and variables, e.g., $Car1.year > v2005$ and $done! = vTrue$.

nodes in a path, checking for mutual exclusion between two nodes in a path, checking for precedence between two nodes in a path, and checking an assertion at the end of a path. Deadlock detection and various sanity checks are also performed during execution.

10.1.4 Observations about the System Model

Diagrams The System Model uses very few diagrams. The lack of such diagrams, typically common for explaining structure, can make it difficult to understand the underlying structure of the System Model. There is one diagram, shown below in Figure 10.1, that assists in comprehending the System Model. If we assume that 'containment' in Figure 10.1 implies specialization, then this diagram, and the class diagram we provide in Figure 3.2 on page 39 are very similar. In fact, the diagram in Figure 10.1 was the inspiration for the type and value hierarchies shown in Figure 3.2.

However, there are obviously some errors in Figure 10.1. For instance:

- The word "deref" is shown in the lower left corner, but there is no arrow indicating which two elements are part of the deref relationship. We suggest that this missing arrow is between *Reference*s and *UVAL*.
- There is a missing *CAR* relationship between *LocTypes* and *ULOC*, as well as between *ReferenceTypes* and *References*.
- Although *INSTANCE* is contained by *RecordValues*, there is no similar containment on the type side. Instead, the System Model simply refers to the concept of *dereference* of a type name, e.g., $*C$, to describe the record type

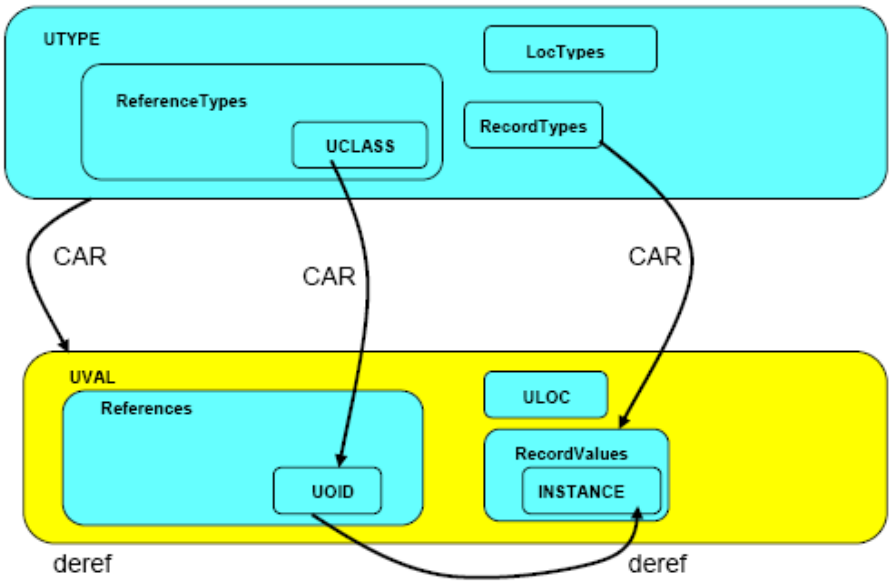


Figure 10.1: Overall picture of the semantic universe [13, Figure 4]

structure of a class name. We have made the concept concrete and have named it *ObjectType*.

Static vs. Dynamic One of the guiding factors of the System Model is that there should be a strict separation between static and dynamic data. Consider, for instance, the diagram in Figure 3.5 on page 43, which shows the structure of an object identifier and how dereferencing is used to read the current value of that object's attributes. Although this two-stage dereferencing is acceptable, and indeed, has been incorporated into both our formal mapping and our interpreter, there is some question as to whether or not this separation between static and dynamic data is necessary.

Ref* vs. *Loc In the same vein, due to the two-stage dereferencing discussed above, there are two types of 'pointers' described in the System Model. References are pointers to values but are considered mathematically defined functions, and therefore do not change over time. In other words, references always refer to the same value. On the other hand, locations are also pointers, but the value of a location is mutable. In fact, the value of a location depends on the current state of the underlying state machine. Again, both our formal mapping and our implementation incorporate both references and locations. However, there is some question as to the necessity of two separate pointer types, even if the two-stage dereferencing discussed above is maintained.

Imperative Bias Although the System Model was created to avoid potential biases of other formalisms, e.g., Abstract State Machines, Z, etc., it does in fact have an inherent bias towards imperative programming. For instance, the control store is mainly composed of program counters, stacks and stack frames. These concepts match nicely with the notion of a typical imperative program executing on a generic computer, but do not necessarily marry well with UML's behavioural formalisms, e.g., state machines, activities, etc. For instance, the System Model documentation states that a thread is active if its stack is non-empty. However, a stack is only used when a call is made; calls are only made in activities when the CallOperationAction is executed. This is only one of several invocation actions used in activities, and only one of 45 actions in general. In other words, most of the time, a thread (of which there will be one per token, i.e., several per activity) stack will always be empty. In order to accommodate activities, we ignore this caveat about the status of threads and treat all threads as active. In fact, the driving force behind the execution of an

activity is not threads, but the tokens associated with those threads, and whether or not the action nodes associated with the tokens are enabled or not.

Mapping m Part of the data structure is the mapping from the locations representing an object's attributes to their actual values, i.e., $m : ULOC \rightarrow UVAL$. The System Model documentation describes this function as total, i.e., every location maps to some value. In actuality, however, it is not always the case that a location maps to a value. For instance, when a new object is created, e.g., with the CreateObjectAction, a new object identifier is created, and new locations are added to the mapping. However, by definition, these mappings cannot actually point to a value (even the *void* value provided by the System Model). The UML specification is very clear that an object's attributes are not initialized, i.e., "the new object has no structural feature values" [59, §11.3.16]. For the purposes of our formal mapping and interpreter, we assume that the mapping m is actually a partial function, i.e., it is possible that a mapping for some location is undefined.

10.1.5 Observations About the UML Specification

References One of the problematic aspects of this research is the lack of complete detail in the UML specification [59] itself. Although comprehensive, there are occasionally questions for which no answer in the specification can be found. Sometimes it is possible to use other, trusted resources, such as the UML Reference Manual [67], or Conrad Bock's series of articles [2, 3, 4, 5, 6, 7]. Unfortunately, even those references cannot answer all questions. At times, we have been able to ask questions of both Conrad Bock and Bran Selic, authors of parts of the specification.

Fork The specification makes the following claims [59, §12.3.30] about how tokens flow through the fork control node: "tokens arriving at a fork are duplicated" and"duplicates of the token are made and one copy traverses each edge". No mention is made in the specification about exactly how data (e.g., an object) would be duplicated when moving through a fork node. However, Conrad Bock stated in an email [8] that a fork node only copies the "reference to the data". He later clarified this by stating that "token copying is a deep [copy], but object tokens are just references to objects so only the reference is copied, not the object itself." [9].

Unnamed Edge Recall from Section 2.4.1.4 that there are two types of flow in an activity–control flow and object flow. Both types of flow are represented as edges in an activity diagram; although they appear identical, they are "distinguished by usage" [2]. There is actually one other kind of edge that is not well-explained by the specification, namely the edge between an action node and its pin. In the standard pin notation, no edge is visible, as the pin is attached directly to the action node (see Figure 2.18(a) on page 26). However, in the standalone pin notation, it becomes obvious that such an edge exists, connecting an action node to its pins (see Figure 2.18(b) on page 26). The specification states that the two edges shown in Figure 2.18(b) "denote a single object flow edge between two pins in the underlying model" [59, §12.3.37]. However, there needs to be some edge, at least conceptually, between an action node and its pin, otherwise, "there is no way to get tokens between them" [8]. The specific description of this type of edge is not important for our formal mapping (we simply assume that an action node can access its pins), but for our interpreter, we treat all edges equally. In other words, by using the standalone pin notation, we force the edges between actions and their pins to be explicit.

Join The specification clearly states that, with respect to the join node, "if any tokens are offered to the outgoing edge, they must be accepted or rejected for traversal before any more tokens are offered to the outgoing edge. If tokens are rejected for traversal, they are no longer offered to the outgoing edge" [59, §12.3.34]. We raised this issue with Conrad Bock; if such a constraint were implemented, it could lead to activities stalling prematurely because offers were being discarded. In no other situation is an offer simply discarded; even with competition, one offer is accepted. Bock suggested [10] that this was a bug in the specification, although this was never confirmed. In fact, the most current version of the UML specification [61] still contains this statement.

DestroyObjectAction The purpose of the DestroyObjectAction is to destroy an object, i.e., remove it from the current data store. However, every action exists in the context of an activity, and every activity is owned by some object. Theoretically, it would be possible for a DestroyObjectAction to destroy the object owning the activity containing the action itself. The UML specification does not address this issue. Is this activity well-formed? Could the object be destroyed but the behaviour remain active? Should the activity be destroyed along with the object?

Invocation Actions We have discussed invocation actions in great detail in Section 5.4. Some observations specific to invocation actions are:

- **StartClassifierBehaviorAction** For some reason, the StartClassifierBehaviorAction is not considered an invocation action. However, we feel that it should be an invocation action, as discussed in Section 5.4, as its effect is to "start the classifier behavior of the input" [59, §11.3.46].

- **CallOperationAction and AcceptCallAction** The UML specification is very clear on the fact that an AcceptCallAction must be paired with a ReplyAction. However, the requirement that a CallOperationAction must be received by an AcceptCallAction is not clear at all, especially when reading the description of the former action. Only by reading about all three actions can the user determine that these three must be used together in order to represent calls.

- **AcceptCallAction vs. AcceptEventAction** The UML specification indicates that an AcceptCallAction must be used when a synchronous CallOperationAction is invoked. It also implies that a AcceptEventAction should be used when a call is invoked asynchronously. This constraint would force modellers of the invoked behaviour to know exactly how their behaviour would be called, because, as shown in Section 8.5, these two accept actions are used very differently. As discussed in Section 5.3.2.1, we simply use the AcceptCallAction for all calls, regardless of whether they are invoked synchronously or asynchronously.

- **Syntactic Sugar** As discussed in Section 5.4.1, we feel that some of the invocation actions are purely syntactic sugar and could be eliminated to streamline the UML specification. Specifically, the StartClassifierBehaviorAction, SendObjectAction, and even the BroadcastSignalAction, could be removed. This would leave the CallBehaviorAction for direct asynchronous invocations and the CallOperationAction and SendSignalAction for indirect asynchronous invocations.

Misinterpretation Although more a lack of education rather than a flaw in the UML specification, we have observed a worrisome tendency for users to misuse activity diagram notation. Specifically, we find that modellers are treating activities like

flowcharts (see our discussion in Section 2.4.3.1 on page 29) and are ignoring the constraint that an action cannot execute unless it receives tokens along all of its incoming edges. Even the UML Reference Manual clearly states, "An action may begin execution when input values are available on all its input pins and control tokens are available on all incoming control edges" [67, page 136].

The example presented in Section 8.2 is just one example of this misinterpretation. We originally believed that this example was a rare incident of one user misinterpreting the UML specification (and UML Reference Manual). However, in the most recent proceedings [21] of the Model Driven Engineering Languages and Systems (MoDELS) conference (the premier venue for model-driven topics), there were no less than three separate papers containing erroneous activities (i.e., missing the explicit merge of flows into an action node). It is concerning that such a critical mistake is being made by the leading researchers in the modelling field. With these subtle errors, modellers are creating activities that will never execute as expected.

10.1.6 ADLF Notation

A side effect of our research was the naming and documentation of the Activity Diagram Linear Form (ADLF). This notation was developed in a government technical report and was used to make activities accessible to screen-readers. At the time, the notation was nameless and undocumented. After seeking out the original creator of the notation, we created an EBNF grammar and LL(1) parser for our own use. Concurrently, the original author created a BNF grammar and LR parser. We collaborated on a technical report [31] documenting the notation, along with the author of the Petri Net Linear Form (PNLF), upon which the newly-named ADLF was based.

10.1.7 The Three-layer Semantics Architecture

Another major contribution of our work is that we have demonstrated that the three-layer semantics architecture, shown in Figure 2.1 on page 8, is useful for describing the nature of behaviour in UML. By using activities as our action language, we successfully map a higher-level behavioural formalism to actions. Then, we map these actions to the state transformers over the structural foundation, described by the System Model, a semantic domain specifically designed to capture the semantics of UML models on a general level.

10.2 Future Work

Now that the System Model is implemented, UML actions are formally defined and implemented, and we have a working interpreter for actions and activities, there is much interesting future work to be pursued.

10.2.1 Coverage

Multiple Values The UML specification supports multiple values for structural features and variables. The System Model supports the notion of one value per variable/attribute. For simplicity, we assume only one value is used in any structural feature or variable. We would like to modify our mappings to fully support multiple values. For instance, the System Model read functions return a single value; we could maintain this but make use of a collection value to hold multiple values. Similarly, the System Model functions for addition of values, e.g., into the data store, would have to be slightly modified to allow for the placement of values into a specific place

in an ordered collection. Once the System Model and mappings were modified, we could incorporate these changes into the interpreter.

Link and Association Actions In UML, links are instances of associations, specifically, a link is a "tuple with one value for each end of the association" [59, §7.3.3]. Although the System Model supports associations in general, it clearly does not permit the instantiation of associations, i.e., "associations are somehow contained within the [data] store...they are somehow part of objects and locations and association relations do not extend the store" [13, page 22]. Instead of being able to create/retrieve link instances, the System Model relies on retrieval functions to "derive the actual links for any n-ary association based on the current store" [13, page 22]. That is not to say that the System Model could not be modified to permit actual instantiations of associations; rather, a philosophical decision was made early on to restrict the data store to objects and mappings to values instead.

We would like to modify the System Model to incorporate links, perhaps as another element of the data store, and then formally define and implement these actions. Given that the finer points of the semantics of UML associations can be quite complicated [24, 51], it may or may not be a trivial issue to modify the System Model structure to permit the use of links as instantiations of associations. Before implementing the link/association actions, further research would be required in order to ensure that the UML specification was well understood and that these actions were implemented as intended.

Remaining Actions We would like to define and implement several other actions that were not covered for various reasons:

- The ReclassifyObjectAction. The System Model supports dynamic reclassification, but only "according to the given subclass hierarchy" [15, page 18]. In other words, an object could only be reclassified to another classifier already related to it, e.g., similar to casting. Plus, this "only changes the external viewpoint of an object, but neither its internal structure...nor its behaviour." [15, page 18]. On the other hand, UML supports a more general form of reclassification. Even if the System Model were modified to allow for a general dynamic reclassification, there are certainly issues [26] with simply reclassifying objects without considering the side effects. For instance, when an object is reclassified, does it retain its old attributes? Does it gain the attributes of its new classifier? What if an old and new attribute same the same $UVAR/UTYPE$–which attribute remains?

- The RaiseExceptionAction is not supported simply because we do not support exceptions or structured activities. In order to perform the formal mapping, we would have to introduce the concept of exceptions into the System Model. In addition, changes would have to be made to the ADLF grammar and parser, as well to the interpreter's scheduler in order to implement this action.

- The ReduceAction and UnmarshallAction do not necessarily pose any serious problems, either with respect to the System Model or the interpreter. However, for simplicity, we have not mapped or implemented these actions.

- The OpaqueAction would be relatively simple to incorporate into our interpreter. In fact, its implementation would be very similar to how we handle the user-defined actions discussed in Section 8.1. On the other hand, formally defining

the execution of a general opaque action would be impossible. However, it is possible to require that users provide pre- and post-conditions for opaque actions, as well as their implementation.

Activity Coverage As mentioned above, the current interpreter does not support several aspects of activities. We would like to expand the coverage of activities, e.g., by providing support for concepts such as: partitions, structured activity nodes, weights on edges, multiplicities other than 1..* on pins, etc. In order to support some of these notions, we would need to modify the ADLF grammar and parser, e.g., to add in partitions, weights on edges, etc.

Virtual Machine ACTi currently represents only a partial implementation of the kind of virtual machine envisioned in [58]. We would like to broaden the scope of the interpreter to cover additional diagram types. Our formalization and implementation of actions, together with the fact that the System Model was designed to support different types of behavioural specification diagrams (initial formalizations of interactions and state machines can be found in [16, 17]) will be extremely helpful with this task.

10.2.2 Interface

As it stands right now, the ACTi interpreter is a stand-alone, text-based tool. We would very much like to make it possible to incorporate ACTi into a tool chain, as suggested by the OMG RFP for executable UML [58], e.g., by incorporating the following ideas:

- **cd/obj files** The contents of the cd and obj files are used as input to ACTi. The cd file contains the static structure of the classifiers used to create objects in the System Model universe; the obj file contains actual instances of these classifiers. A tool, such as the Human-Usable Textual Notation (HUTN) [57] implementation described in [66], could be used to create equivalent input files for the interpreter. This would allow a standard notation, HUTN, to be used to describe both the classifiers and objects to be used in an execution. The advantage of such a tool is that the user could create the input files using a graphical interface, and some constraint checking (e.g., of multiplicities) could be performed automatically by that tool. The portion of ACTi responsible for reading cd and obj files could be rewritten to read the resulting hutn files as input instead. Finally, the tool discussed in [66] allows for the creation of static structure by example. In other words, the user would create the instances of the classifiers, and the tool would create the structure representing these instances.

- **act files** Similarly, instead of using an ADLF textual representation of an activity as input to ACTi, we would like to create a plugin (e.g., for Rational Software Architect [40]) that would allow the user to create an activity graphically and then use the interpreter to execute that activity. This could be accomplished by either re-implementing ACTi's front-end to read the underlying representation of an activity (e.g., in the XML Metadata Interchange (XMI) [60] format), or perhaps using some kind of intermediate process to transform the activity into an ADLF representation.

- **Output** Once a plugin exists for inputting an activity into the interpreter, we would like to provide the output to the user graphically. In other words, instead

of just listing possible paths textually, we would like to animate the activity itself. Similarly, the results of the analysis could be demonstrated graphically.

10.2.3 Analysis

Soundness Activities are becoming quite popular in terms of business process modelling/workflow modelling [68, 82], including different standards for analysis of activities. For instance, [29] discusses a set of *soundness* rules, originally described in terms of workflow modelling [81]. Essentially, an activity can be considered sound, if the following conditions hold [29]:

- There is exactly one initial node, and one final node.
- It must be possible to execute every action—not necessarily in every execution, but in at least one execution.
- When a token arrives at the final node, there must not be any other tokens left in the activity.
- A token eventually arrives at the final node.

Our interpreter could be used to check activities with respect to these soundness rules.

Well-formedness The soundness rules discussed above are actually quite restrictive and limit 'sound' activities to a very narrow definition. Given the complicated nature of the semantics of activities though, it would be useful to compile a collection of more general well-formedness[2] rules. Once standards for well-formedness have been identified, it would be possible to add a well-formedness check to the analysis capabilities of ACTi.

A major benefit of performing this type of analysis would be to relieve some of the burden placed on modellers. Currently, the modeller must ensure the well-formedness of an activity. For instance, it is the modeller's responsibility to: consider how multiple streams of tokens interact [59, §12.3.4], arrange that each token through a decision node can traverse only one edge [59, §12.3.22], ensure that downstream joins do not depend on tokens travelling through forks with guarded edges [59, §12.3.30], etc. In other words, the modeller is responsible for ensuring that their activities conform to the more complicated aspects of execution semantics.

Model Checking The creation of a true model checker for activities would be very useful. As it stands now, ACTi performs a naive type of model checking by simply executing a given activity multiple times. A true model checker would require rewriting of the interpreter's scheduler to be able to exhaustively explore each path, e.g., a depth-first search through the activity graph, with appropriate record keeping and backtracking.

[2] A well-formed model is "correctly constructed", "satisfies all the pre-defined and model-specific rules and constraints" and "has meaningful semantics" [67]. In the UML specification, well-formedness rules are often defined by OCL constraints; however, some of the more esoteric rules are simply described in text.

Bibliography

[1] Bison: GNU parser generator, 2006. `http://www.gnu.org/software/bison/`.

[2] C. Bock. UML 2 activity and action models. *Journal of Object Technology*, 2(4):43–53, 2003.

[3] C. Bock. UML 2 activity and action models, part 2: Actions. *Journal of Object Technology*, 2(5):41–56, 2003.

[4] C. Bock. UML 2 activity and action models, part 3: Control nodes. *Journal of Object Technology*, 2(6):7–23, 2003.

[5] C. Bock. UML 2 activity and action models, part 4: Object nodes. *Journal of Object Technology*, 3(1):27–41, 2004.

[6] C. Bock. UML 2 activity and action models, part 5: Partitions. *Journal of Object Technology*, 3(7):37–56, 2004.

[7] C. Bock. UML 2 activity and action models, part 6: Structured activities. *Journal of Object Technology*, 4(4):43–66, 2005.

[8] C. Bock. "re: The execution model". Email to B. Selic, November 2, 2006.

[9] C. Bock. "re: The execution model, rules 2". Email to B. Selic, December 1, 2006.

[10] C. Bock. "re: Question about offers and tokens through join nodes". Email to M.L. Crane, February 14, 2008.

[11] C. Bock. "re: Token/offer semantics for activities". Email to J. Dingel, April 25, 2008.

[12] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer, 2003.

[13] M. Broy, M.V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The Structural Data Model. Technical Report TUM-I0612, TUM, 2006.

[14] M. Broy, M.V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The Control Model. Technical Report TUM-I0710, TUM, 2007.

[15] M. Broy, M.V. Cengarle, and B. Rumpe. Semantics of UML – Towards a System Model for UML: The State Machine Model. Technical Report TUM-I0711, TUM, 2007.

[16] M.V. Cengarle. System model for UML – the interactions case. In *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[17] M.V. Cengarle, H. Grönniger, and B. Rumpe. System model semantics of statecharts. Technical Report 2008-04, Technische Universität Braunschweig, 2008.

[18] CERAS. Centre of Excellence for Research in Adaptive Systems. `https://www.cs.uwaterloo.ca/twiki/view/CERAS`.

[19] M.L. Crane and J. Dingel. Towards a formal account of a foundational subset for executable UML models. In *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *LNCS*, pages 675–689. Springer, 2008.

[20] M.L. Crane and J. Dingel. Towards a UML virtual machine: Implementing an interpreter for UML 2 actions and activities. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 96–110. ACM, 2008.

[21] K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors. *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *LNCS*. Springer, 2008.

[22] T. Dinh-Trong. *A Systematic Approach to Testing UML Designs*. PhD thesis, Colorado State University, 2007.

[23] T. Dinh-Trong, S. Ghosh, R. France, and A.A. Andrews. A systematic approach to testing UML design models. In *4th International Workshop on Critical Systems Development Using Modeling Languages (CSDUML)*, 2005.

[24] Z. Diskin and J. Dingel. Mappings, maps and tables: Towards formal semantics for associations in UML 2. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 4199 of *LNCS*, pages 230–244. Springer, 2006.

[25] D. Dotan and A. Kirshin. Debugging and testing behavioral UML models. In *Proceedings of the 22nd ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)*, pages 838–839. ACM, 2007.

[26] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object reclassification: Fickle. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(2):153–191, 2002.

[27] J. Dubrovin. Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, March 2006.

[28] P. Naur (editor), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[29] G. Engels, C. Soltenborn, and H. Wehrheim. Analysis of UML activities using dynamic meta modeling. In *Proceedings of the 9th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4468 of *LNCS*, pages 76–90. Springer, 2007.

[30] R. Eshuis. Symbolic model checking of UML activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1):1–38, 2006.

[31] D. Flater, P.A. Martin, and M.L. Crane. Rendering UML activity diagrams as human-readable text. Technical Report NISTIR 7469, National Institute of Standards and Technology, November 2007.

[32] Flex: the Fast Lexical Analyzer, 2006. `http://flex.sourceforge.net/`.

[33] R.B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using UML 2.0: promises and pitfalls. *IEEE Computer Magazine*, 39(2):59–66, 2006.

[34] L. Fuentes, J. Manrique, and P. Sánchez. Execution and simulation of (profiled) UML models using Pópulo. In *MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering*, pages 75–81. ACM, 2008.

[35] L. Fuentes, J. Manrique, and P. Sánchez. Pópulo: A tool for debugging UML models. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 955–956. ACM, 2008.

[36] L. Fuentes and P. Sánchez. Towards executable aspect-oriented UML models. In *Proceedings of the 10th International Workshop on Aspect-oriented Modeling (AOM)*, pages 28–34. ACM, 2007.

[37] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[38] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.

[39] D. Harel and B. Rumpe. Meaningful modeling: What's the semantics of "semantics"? *IEEE Computer Magazine*, 37(10):64–72, 2004.

[40] IBM. Rational Software Architect. `http://www-01.ibm.com/software/awdtools/architect/swarchitect/`.

[41] ISO. Information technology—Syntactic metalanguage—Extended BNF. ISO/IEC 14977, International Organization for Standardization, 1996.

[42] ISO. Industrial automation systems and integration–process specification language–part 1: Overview and basic principles. 18629-1:2004, International Organization for Standardization, 2004.

[43] ISO. Information technology – common logic (cl): A framework for a family of logic-based languages. 24707:2007, International Organization for Standardization, 2007.

[44] JavaCC, the Java Compiler Compiler, 2006. `https://javacc.dev.java.net/`.

[45] K. Jiang, L. Zhang, and S. Miyake. An executable UML with OCL-based action semantics language. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 302–309, 2007.

[46] Kabira. Kabira design center. `http://www.kabira.com/Products/Application_Development`.

[47] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[48] L. Lamport. The +CAL algorithm language. `http://research.microsoft.com/users/lamport/pubs/pluscal.pdf`, February 2008.

[49] P.A. Martin. The Petri Net Linear Form. `http://www.phmartin.info/wf/pnlf/`, 2007.

[50] S.J. Mellor and M. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison-Wesley, 2002.

[51] D. Milicev. On the semantics of associations and association ends in UML. *Transactions on Software Engineering*, 33(4):238–251, April 2007.

[52] NIST. CRT draft 20050914. National Institute of Standards and Technology draft report `http://vote.nist.gov/20050929_20dwfWorkingDraft20050914.pdf`, September 2005.

[53] NIST. Improving U.S. Voting Systems: Other topics? National Institute of Standards and Technology presentation `http://www.ss.ca.gov/elections/vstsummit/presentations/guttman_barbara_additional_material.PPT`, 2005.

[54] T.S. Norvell. The JavaCC FAQ, 2007. `http://www.engr.mun.ca/~theo/JavaCC-FAQ/javacc-faq-moz.htm`.

[55] I. Ober, B. Coulette, and M. Gandriau. Action language for the UML. In *Langages et Modèles à Objets (LMO)*, pages 277–291. Hermes, 2000.

[56] OMG. Action semantics for the UML. Response to OMG RFP ad/98-11-01 OMG ad/2001-08-04, Object Management Group, 2001.

[57] OMG. Human-Usable Textual Notation Specification, v1.0. Document formal/04-08-01, Object Management Group, 2004.

[58] OMG. Semantics of a foundational subset for executable UML models. Request for Proposal ad/2005-04-02, Object Management Group, April 2005.

[59] OMG. Unified Modeling Language: Superstructure version 2.1. Document ptc/06-01-02, Object Management Group, January 2006.

[60] OMG. MOF 2.0/XMI Mapping Specification, V2.1.1. Document formal/07-12-01, Object Management Group, 2007.

[61] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. Document formal/2007-11-02, Object Management Group, November 2007.

[62] OMG. OMG TC Work in Progress. `http://www.omg.org/schedule`, October 2008.

[63] OMG. Semantics of a foundational subset for executable UML models. Second Revised Submission ad/2008-08-03, Object Management Group, August 2008.

[64] I. Perseil and L. Pautet. A concrete syntax for UML 2.1 action semantics using +CAL. In *Proceedings of the 13th IEEE International Conference on Engineering and Complex Computer Systems (ICECCS)*, pages 217–221, 2008.

[65] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[66] L.M. Rose, R.F. Paige, D.S. Kolovos, and F. Polack. Constructing models with the Human-Usable Textual Notation. In *11th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 5301 of *LNCS*, pages 249–263. Springer, 2008.

[67] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2005.

[68] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Proceedings of the 3rd Asia-Pacific Conference on Conceptual Modelling (APCCM)*, pages 95–104. Australian Computer Society, Inc., 2006.

[69] S. Sarstedt, J. Kohlmeyer, A. Raschke, M. Schneiderhan, and D. Gessenharter. ActiveChartsIDE. Poster at the European Conference on Model Driven Architecture (ECMDA'05), November 2005.

[70] T. Schattkowsky and A. Förster. On the pitfalls of UML 2 activity modeling. In *International Workshop on Modeling in Software Engineering (MISE)*, pages 8–8, 2007.

[71] B. Selic. On the semantic foundations of standard UML 2.0. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems (SFM-RT 2004)*, volume 3185 of *LNCS*, pages 181–199. Springer, 2004.

[72] B. Selic. An overview of UML 2.0. Presentation, 2004.

[73] B. Selic. UML 2: A model-driven development tool. *IBM Systems Journal*, 45(3):607–620, March 2005.

[74] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

[75] S. Shlaer and S.J. Mellor. *Object Lifecycles: Modeling the World in States*. Prentice Hall, 1992.

[76] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[77] H. Störrle and J. Hausmann. Towards a formal semantics of UML 2.0 activities. In *Software Engineering*, volume 64 of *LNI*, pages 117–128, 2005.

[78] Telelogic. Tau. `http://www.telelogic.com/`.

[79] UML 2 semantics project web page. `http://www.cs.queensu.ca/~stl/internal/uml2`, 2006.

[80] A UML-based Specification Environment. `http://www.db.informatik.uni-bremen.de/projects/USE/`, 2007. Web page of the USE project, from the University of Bremen.

[81] W. van der Aalst. Verification of workflow nets. In *Proceedings of the 18th International Conference on the Application and Theory of Petri Nets (ICATPN)*, volume 1248 of *LNCS*, pages 407–426. Springer, 1997.

[82] V. Vitolins and A. Kalnins. Semantics of UML 2.0 activity diagram for business modeling by means of virtual machine. In *9th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 181–194. IEEE Computer Society, 2005.

Appendix A

Metamodels of Actions

This appendix contains UML metamodels for the actions that have been mapped to the System Model and implemented in our interpreter. Only concrete actions are included. Those parts of the metamodel not supported by our mapping and implementation are marked. Those elements that are supported by our formal mapping, but not the interpreter, are marked with '/'. Those elements that are not supported by our formal mapping or the interpreter are marked with 'X'.

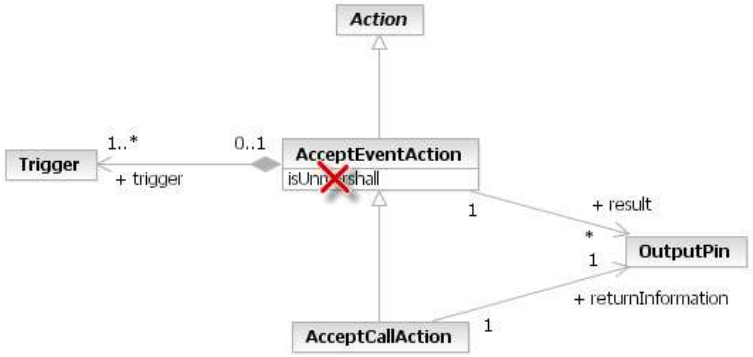


Figure A.1: UML metamodel of the AcceptCallAction. This action represents the receipt of a synchronous call request, i.e., it waits for a synchronous call [59, §11.3.1]

Figure A.2: UML metamodel of the AcceptEventAction. This action waits for an event, such as a signal or an asynchronous call [59, §11.3.2]

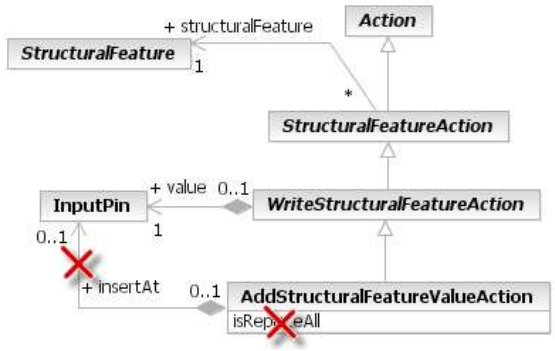


Figure A.3: UML metamodel of the AddStructuralFeatureValueAction. This action adds a value to a structural feature [59, §11.3.5]

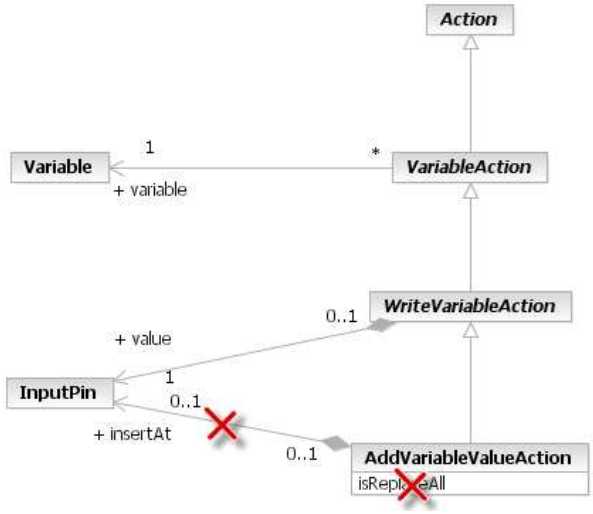


Figure A.4: UML metamodel of the AddVariableValueAction. This action adds values to a variable [59, §11.3.6]

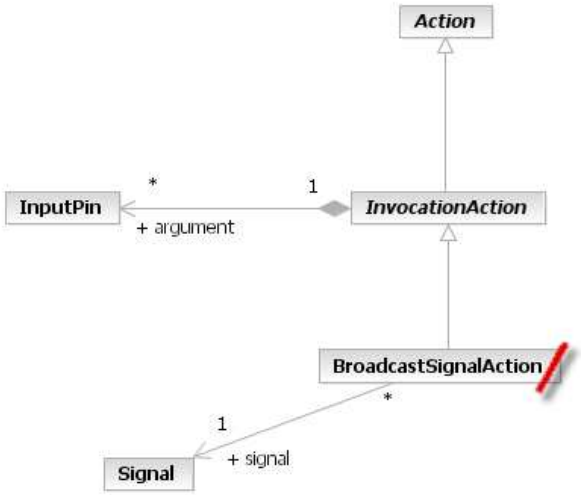


Figure A.5: UML metamodel of the BroadcastSignalAction. This action broadcasts a signal [59, §11.3.7]. This action is not supported by our interpreter

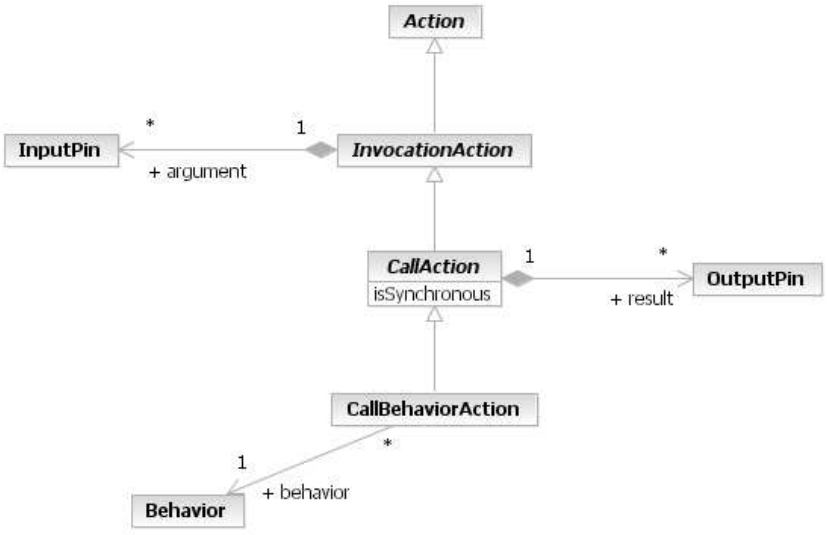


Figure A.6: UML metamodel of the CallBehaviorAction. This action invokes behaviour directly. It may be used synchronously or asynchronously [59, §11.3.9]

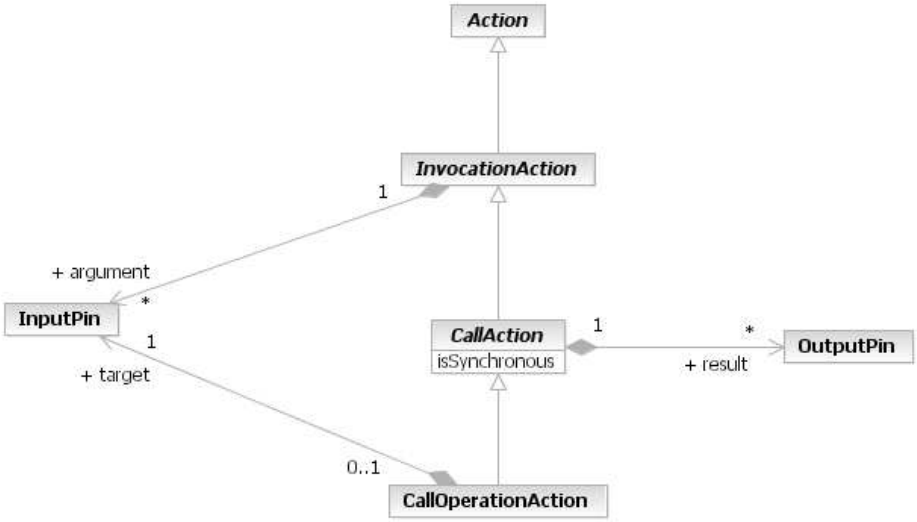


Figure A.7: UML metamodel of the CallOperationAction. This action invokes behaviour indirectly, e.g., a call request is submitted to a target, where it may invoke behaviour. It may be used synchronously or asynchronously [59, §11.3.10]

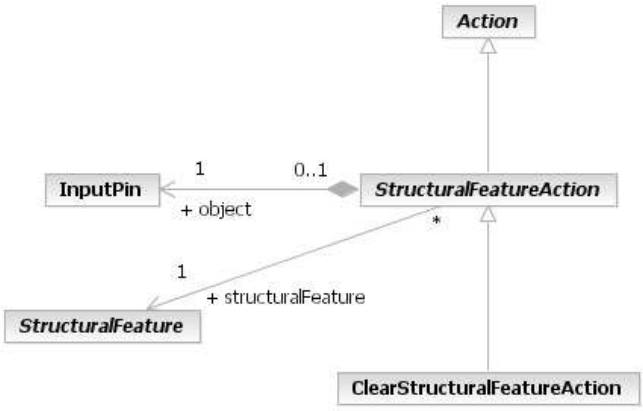


Figure A.8: UML metamodel of the ClearStructuralFeatureAction. This action removes all values from a structural feature [59, §11.3.12]

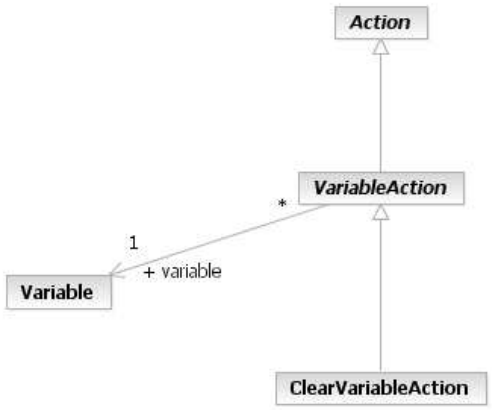


Figure A.9: UML metamodel of the ClearVariableAction. This action removes all values from a variable [59, §11.3.13]

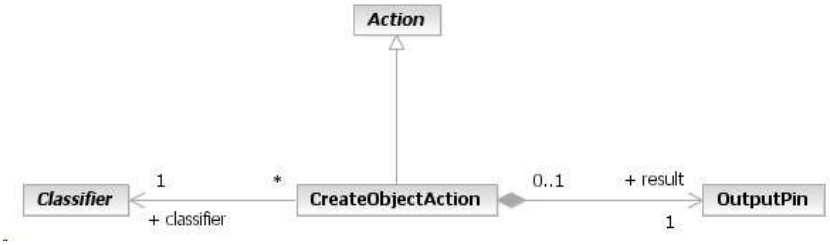


Figure A.10: UML metamodel of the CreateObjectAction. This action creates an object [59, §11.3.16]

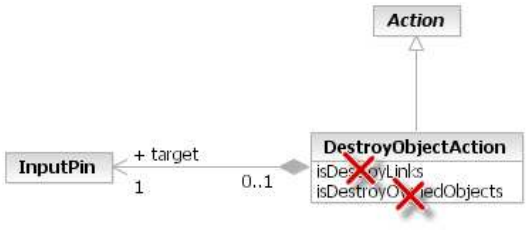


Figure A.11: UML metamodel of the DestroyObjectAction. This action destroys an object [59, §11.3.18]

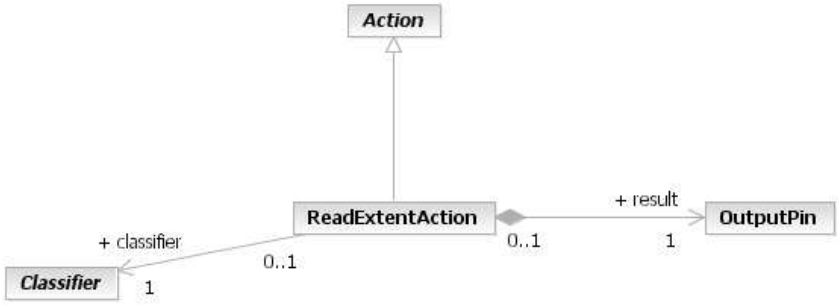


Figure A.12: UML metamodel of the ReadExtentAction. This action retrieves the current instances of a classifier [59, §11.3.31]

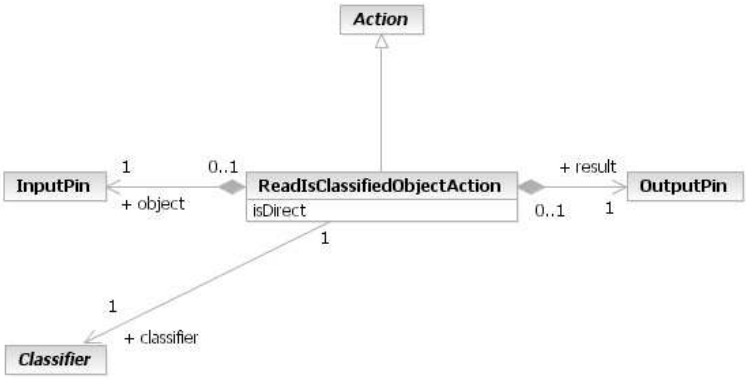


Figure A.13: UML metamodel of the ReadIsClassifiedObjectAction. This action determines whether or not a given object is an instance of a particular classifier [59, §11.3.32]

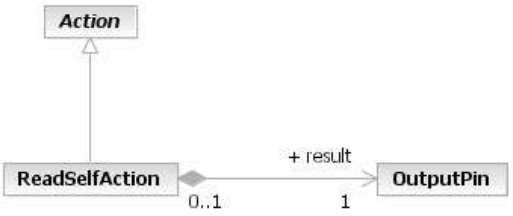


Figure A.14: UML metamodel of the ReadSelfAction. This action retrieves the host object of an action [59, §11.3.36]

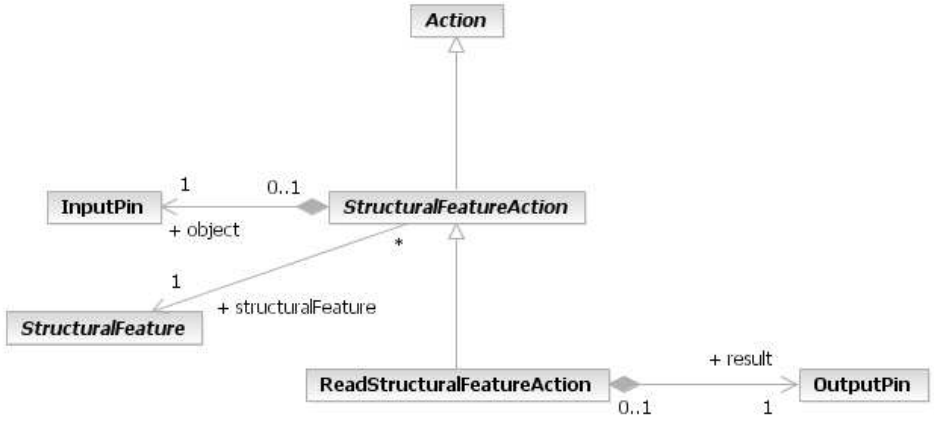


Figure A.15: UML metamodel of the ReadStructuralFeatureAction. This action retrieves the value(s) of a structural feature [59, §11.3.37]

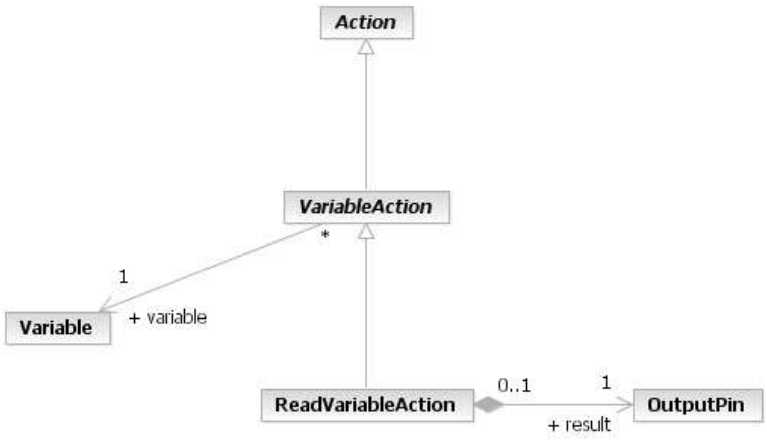


Figure A.16: UML metamodel of the ReadVariableAction. This action retrieves the values of a variable [59, §11.3.38]

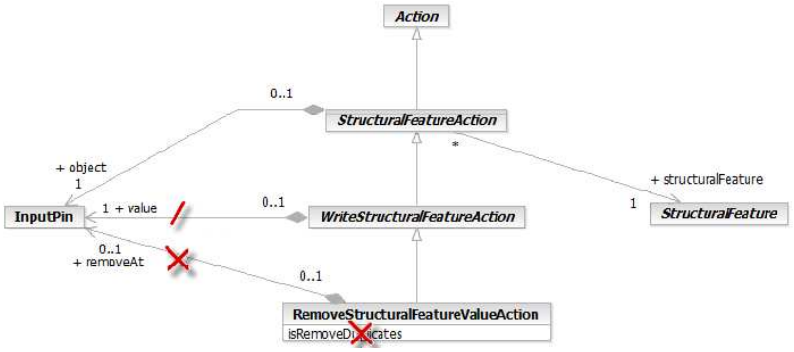


Figure A.17: UML metamodel of the RemoveStructuralFeatureValueAction. This action removes values from a structural feature [59, §11.3.41]

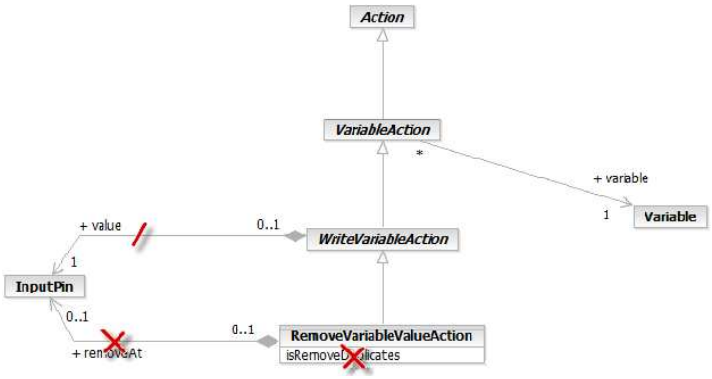


Figure A.18: UML metamodel of the RemoveVariableValueAction. This action removes values from a variable [59, §11.3.42]

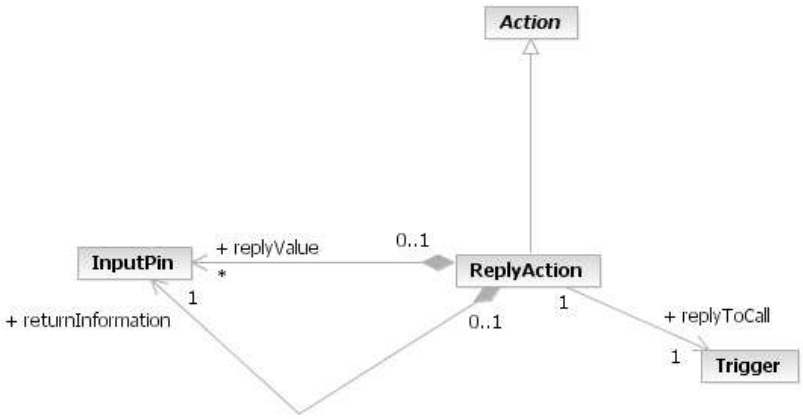


Figure A.19: UML metamodel of the ReplyAction. This action returns values to a calling action [59, §11.3.43]

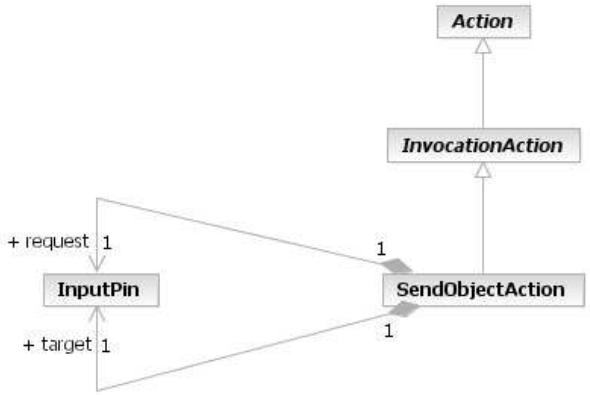


Figure A.20: UML metamodel of the SendObjectAction. This action sends an object to a target [59, §11.3.44]. This action is not supported by our interpreter

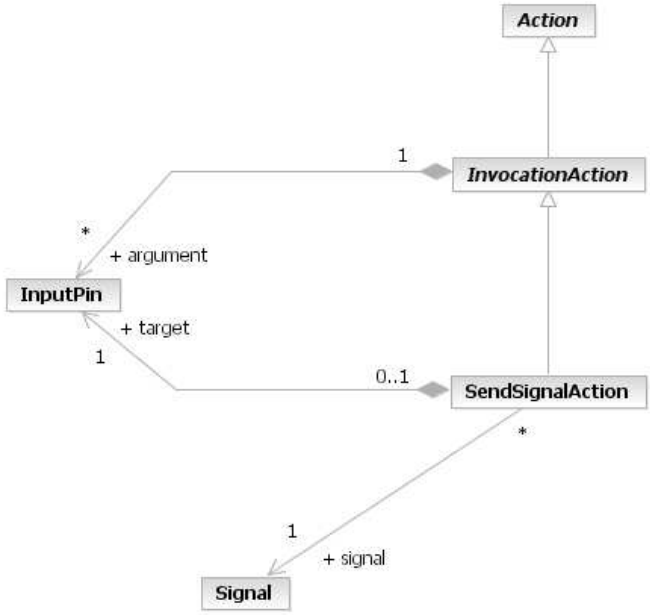


Figure A.21: UML metamodel of the SendSignalAction. This action sends a signal to a target [59, §11.3.45]

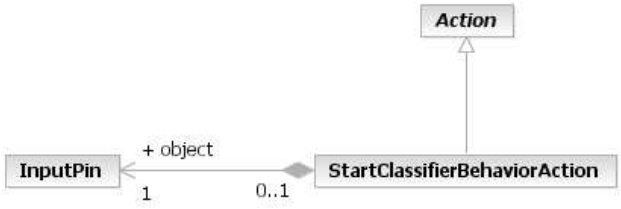


Figure A.22: UML metamodel of the StartClassifierBehaviorAction. This action starts the classifier behaviour of an object [59, §11.3.46]

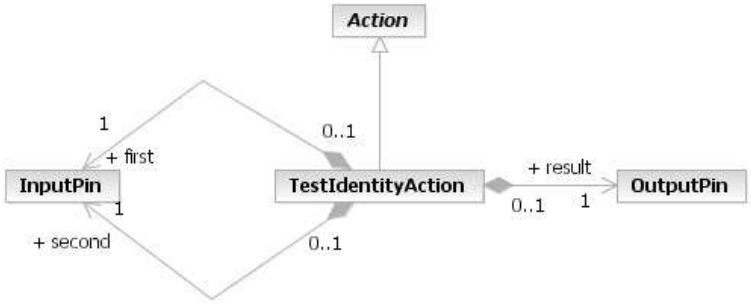


Figure A.23: UML metamodel of the TestIdentityAction. This action tests if two objects are identical [59, §11.3.48]

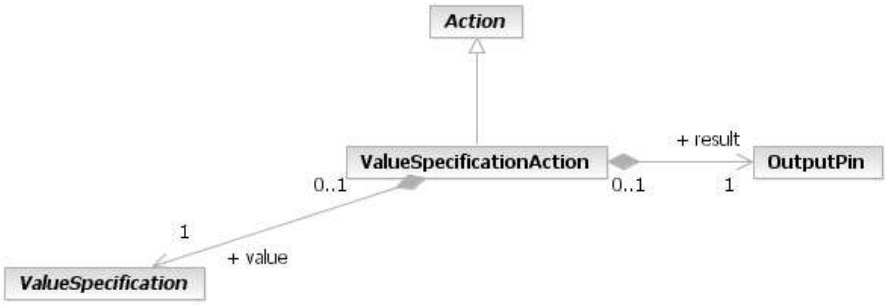


Figure A.24: UML metamodel of the ValueSpecificationAction. This action evaluates a value specification [59, §11.3.51]

Appendix B

Examples

This appendix contains additional information about the example activities presented in Chapter 5, such as input files. In addition, partial output from the interpreter is also presented, showing how the example activities can be executed.

B.1 Example 1: Create and Read Object

The activity shown in Figure B.1 creates an object of type *Car*, using the CreateObjectAction. It then checks that the classifier of the newly created object is in fact *Car*, using the ReadIsClassifiedObjectAction. The value of object node labelled #5 will be *true* if the newly created object is a car object.

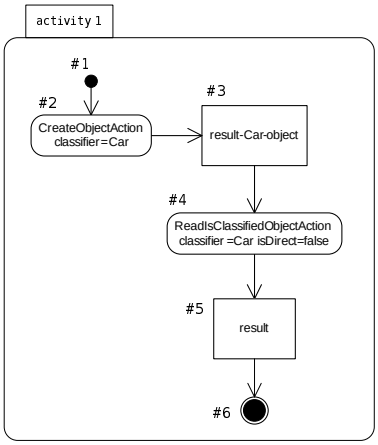


Figure B.1: Example activity to execute CreateObjectAction and ReadIsClassifiedObjectAction

The ADLF representation of this activity is contained in Figure B.2. The cd file, used to initialize the universe before executing this activity, is shown in Figure B.3. Output from a sample execution is shown in Figure B.4.

```
<InitialNode>
-> (CreateObjectAction classifier=Car)
-> [result-Car-object]
-> (ReadIsClassifiedObjectAction classifier=Car isDirect=false)
-> [result]
-> <ActivityFinal>.
```

Figure B.2: ADLF representation of activity in Figure B.1

```
type Bool
 value vTrue
 value vFalse

class Car
```

Figure B.3: cd file

```
Printing the order in which nodes were visited - (4)
[1, 2, 4, 6]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #3[result-Car-object] = OIDfor_Car_1
.  #5[result] = vTrue
.....................................................................

Running time = 541 ms = 0 sec = 0 min
```

Figure B.4: Partial output of a sample execution of activity in Figure B.1

B.2 Example 2: Specify Value and Test Identity

The activity shown in Figure B.5 uses the ValueSpecification action to retrieve existing objects from the System Model universe, and then uses the TestIdentityAction to compare the objects. The value of the object node #8 will be *true* if both retrieved objects are identical.

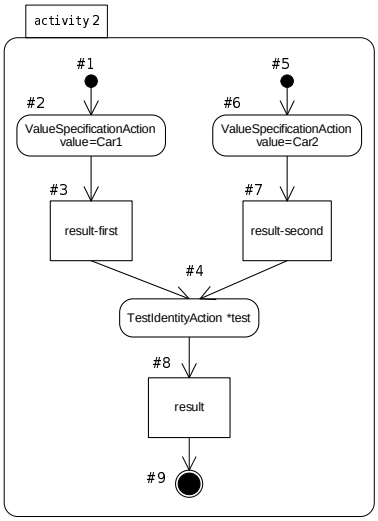


Figure B.5: Example activity to execute ValueSpecificationAction and TestIdentityAction

The ADLF representation of this activity is contained in Figure B.6. The cd and obj files, used to initialize the universe before executing this activity, are shown in Figure B.7 and B.8. Output from a sample execution is shown in Figure B.9.

```
<InitialNode>
  -> (ValueSpecificationAction value=Car1) -> [result-first]
  -> (TestIdentityAction *test);

<InitialNode>
  -> (ValueSpecificationAction value=Car2) -> [result-second]
  -> (*test);

(*test) -> [result] -> <ActivityFinal>.
```

Figure B.6: ADLF representation of activity in Figure B.5

```
type Bool
 value vTrue
 value vFalse

class Car
```

Figure B.7: cd file

```
obj Car1:Car

obj Car2:Car
```

Figure B.8: obj file

```
Printing the order in which nodes were visited - (6)
[5, 6, 1, 2, 4, 9]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #3[result-first] = Car1
.  #7[result-second] = Car2
.  #8[result] = vFalse
.....................................................................

Running time = 431 ms = 0 sec = 0 min
```

Figure B.9: Partial output of a sample execution of activity in Figure B.5

B.3 Example 3: Read Extent and Destroy Object

The activity shown in Figure B.10 reads the extent of the classifier *Car*; the result of this action is a collection of all instances of that type. Then, one of the instances is deleted, and the extent of *Car* is re-read.

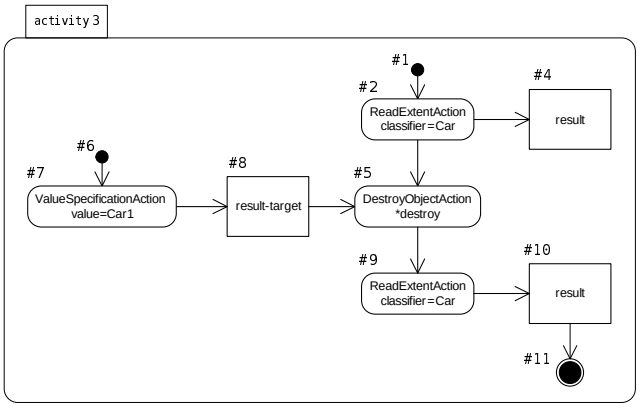


Figure B.10: Example activity to execute ReadExtentAction and DestroyObjectAction

The ADLF representation of this activity is contained in Figure B.11. The cd and obj files, used to initialize the universe before executing this activity, are shown in Figure B.12 and B.13. Output from a sample execution is shown in Figure B.14.

```
<InitialNode>
  -> (ReadExtentAction classifier=Car) {
-> [result],
-> (DestroyObjectAction *destroy)
};

<InitialNode>
  -> (ValueSpecificationAction value=Car1)
  -> [result-target]
  -> (*destroy);

(*destroy)
  -> (ReadExtentAction classifier=Car)
  -> [result]
  -> <ActivityFinal>.
```

Figure B.11: ADLF representation of activity in Figure B.10

```
type Bool
 value vTrue
 value vFalse

class Car
```

Figure B.12: cd file

```
obj Car1:Car

obj Car2:Car
```

Figure B.13: obj file

```
Printing the order in which nodes were visited - (8)
[1, 6, 2, 7, 4, 5, 9, 11]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #4[result] = vCollection0 : [Car1, Car2]
.  #8[result-target] = Car1
.  #10[result] = vCollection1 : [Car2]
.....................................................................

Running time = 561 ms = 0 sec = 0 min
```

Figure B.14: Partial output of a sample execution of activity in Figure B.10

B.4 Example 4: Structural Feature Actions

The activity shown in Figure B.15 performs several actions on a structural feature. In our example, the *year* structural feature, or attribute, of *Car1* is modified to the value *v2005* by the AddStructuralFeatureValueAction. The attribute is read by the ReadStructuralFeatureAction. Then, the attribute value is completely cleared with the ClearStructuralFeatureAction. Finally, the attribute is read again. At this point, the attribute has no value, resulting in a null result, as can be seen by the partial output

in Figure B.19 on page 294.

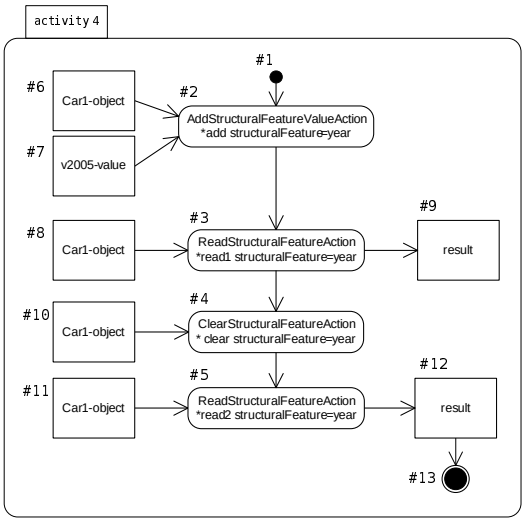


Figure B.15: Example activity to execute AddStructuralFeatureValueAction, ReadStructuralFeature-Action and ClearStructuralFeatureAction

The ADLF representation of this activity is contained in Figure B.16. The cd and obj files, used to initialize the universe before executing this activity, are shown in Figure B.17 and B.18. Output from a sample execution is shown in Figure B.19.

```
<InitialNode>
  -> (AddStructuralFeatureValueAction *add structuralFeature=year)
  -> (ReadStructuralFeatureAction *read1 structuralFeature=year)
  -> (ClearStructuralFeatureAction *clear structuralFeature=year)
  -> (ReadStructuralFeatureAction *read2 structuralFeature=year);

[Car1-object] -> (*add);
[v2005-value] -> (*add);

[Car1-object] -> (*read1) -> [result];

[Car1-object] -> (*clear);

[Car1-object] -> (*read2) -> [result] -> <ActivityFinal>.
```

Figure B.16: ADLF representation of activity in Figure B.15

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

class Car
 attr year:Int
```

Figure B.17: cd file

```
obj Car1:Car
```

Figure B.18: obj file

```
Printing the order in which nodes were visited - (12)
[1, 11, 6, 10, 7, 2, 8, 3, 9, 4, 5, 13]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #6[Car1-object] = Car1
.  #7[v2005-value] = v2005
.  #8[Car1-object] = Car1
.  #9[result] = v2005
.  #10[Car1-object] = Car1
.  #11[Car1-object] = Car1
.  #12[result] = null
.....................................................................

Running time = 731 ms = 0 sec = 0 min
```

Figure B.19: Partial output of a sample execution of activity in Figure B.15

B.5 Example 5: Variable Actions

The activity shown in Figure B.20 performs several actions on a (local) variable. In our example, the x variable is set to the value *v2008* by the AddVariableValueAction. The variable is read by the ReadVariableAction. Then the variable is completely cleared with the ClearVariableAction. Finally, the variable is read again. At this point, the variable has no value, resulting in a null result, as can be seen by the partial output in Figure B.23 on page 296.

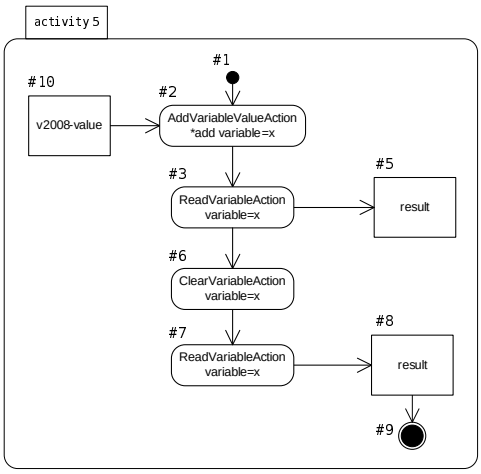


Figure B.20: Example activity to execute AddVariableValueAction, ReadVariableAction and ClearVariableAction

The ADLF representation of this activity is contained in Figure B.21. The cd file, used to initialize the universe before executing this activity, is shown in Figure B.22. Output from a sample execution is shown in Figure B.23.

```
<InitialNode>
  -> (AddVariableValueAction *add variable=x)
  -> (ReadVariableAction variable=x) {
       -> [result],
       -> (ClearVariableAction variable=x)
          -> (ReadVariableAction variable=x)
          -> [result]
          -> <ActivityFinal>
     };

[v2008-value] -> (*add).
```

Figure B.21: ADLF representation of activity in Figure B.20

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

var x:Int
```

Figure B.22: cd file

```
Printing the order in which nodes were visited - (8)
[1, 10, 2, 3, 6, 5, 7, 9]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #5[result] = v2008
.  #8[result] = null
.  #10[v2008-value] = v2008
.....................................................................

Running time = 510 ms = 0 sec = 0 min
```

Figure B.23: Partial output of a sample execution of activity in Figure B.20

B.6 Example 6: Start Classifier Behaviour and Read Self

The activities shown in Figure B.24 demonstrate two actions. The StartClassifier-BehaviorAction is used to initiate the behaviour associated with *Car1*, and instance of the *Car* class. The ReadSelfAction is executed once that classifier behaviour is started, and simply retrieves the instance that owns the activity. As can be seen from the partial output in Figure B.29 on page 299, the instance is *Car1*, as expected.

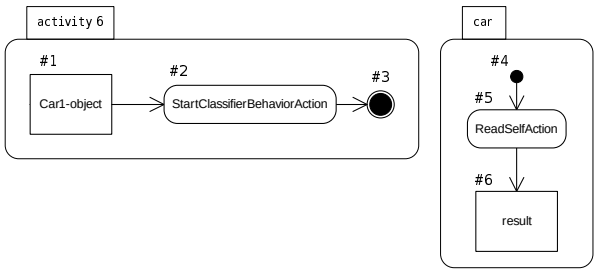


Figure B.24: Example activity to execute StartClassifierBehaviorAction and ReadSelfAction

The ADLF representation of these activities are contained in Figures B.25 and B.26. The cd and obj files, used to initialize the universe before executing these activities, are shown in Figure B.27 and B.28. Output from a sample execution is shown in Figure B.29.

```
[Car1-object] -> (StartClassifierBehaviorAction) -> <ActivityFinal>.
```

Figure B.25: ADLF representation of 'activity6' activity in Figure B.24

```
<InitialNode> -> (ReadSelfAction) -> [result].
```

Figure B.26: ADLF representation of 'car' activity in Figure B.24

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

class Car
 act activity6car.act
```

Figure B.27: cd file

```
obj Car1:Car
```

Figure B.28: obj file

```
Printing the order in which nodes were visited - (6)
[1, 2, 4, 5, 6, 3]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #1[Car1-object] = Car1
.  #6[result] = Car1
.....................................................................

Running time = 491 ms = 0 sec = 0 min
```

Figure B.29: Partial output of a sample execution of activities in Figure B.24

B.7 Example 7: Call Behaviour

The activities in Figure B.30 demonstrate the CallBehaviorAction, which is used to invoke behaviour directly.

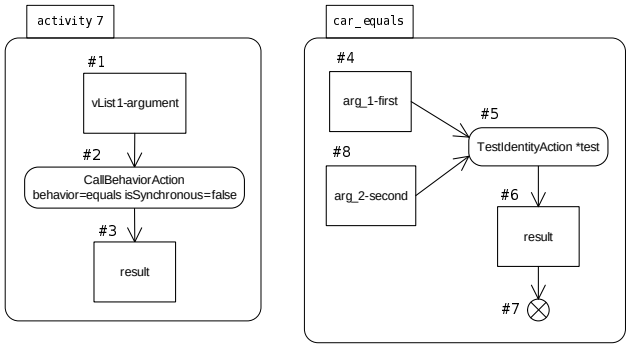


Figure B.30: Example activity to execute CallBehaviorAction asynchronously

The ADLF representation of these activities are contained in Figures B.31 and B.32. The cd and obj files, used to initialize the universe before executing these activities,

are shown in Figure B.33 and B.34. Output from a sample execution is shown in Figure B.35.

```
[vList-argument]
-> (CallBehaviorAction behavior=equals isSynchronous=false)
-> [result].
```

Figure B.31: ADLF representation of 'activity7' activity in Figure B.30

```
[arg_1-first] -> (TestIdentityAction *test) -> [result] -> <FlowFinal>;

[arg_2-second] -> (*test).
```

Figure B.32: ADLF representation of 'car_equals' activity in Figure B.30

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

class Car

op equals(first:Car,second:Car):Bool car_equals.act

value vList:Collect
```

Figure B.33: cd file

```
obj Car1:Car

obj Car2:Car

collection vList
    Car1
    Car2
```

Figure B.34: obj file

```
Printing the order in which nodes were visited - (7)
[1, 2, 8, 3, 4, 5, 7]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #1[vList-argument] = vList : [Car1, Car2]
.  #3[result] = void
.  #4[arg_1-first] = Car1
.  #6[result] = vFalse
.  #8[arg_2-second] = Car2
.....................................................................

Running time = 591 ms = 0 sec = 0 min
```

Figure B.35: Partial output of a sample execution of activities in Figure B.30

Note that the result of the CallBehaviorAction is `void`, even though the invoked behaviour can provide a result (i.e., the result in node #6). The result node #3 is always `void` because the "action completes immediately without a result, if the call is asynchronous" [59, §11.3.9]. Even if node #3 were encountered after #7 in the execution path, its value would still be `void`; see the example output in Figure B.36.

```
Printing the order in which nodes were visited - (7)
[1, 2, 4, 8, 5, 7, 3]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #1[vList-argument] = vList : [Car1, Car2]
.  #3[result] = void
.  #4[arg_1-first] = Car1
.  #6[result] = vFalse
.  #8[arg_2-second] = Car2
.....................................................................

Running time = 291 ms = 0 sec = 0 min
```

Figure B.36: Partial output of sample execution of activities in Figure B.30

B.8 Example 8: Call Operation

The activities in Figure B.37 demonstrate the execution of a synchronous CallOperationAction. The 'car_getYear' activity represents the called behaviour; note the required AcceptCallAction and ReplyAction. This pair of activities is equivalent to the pseudocode statement `Car1.getYear(voidList)`, where `voidList` is an empty list, and the result of the call will be captured by the 'result' node #3.

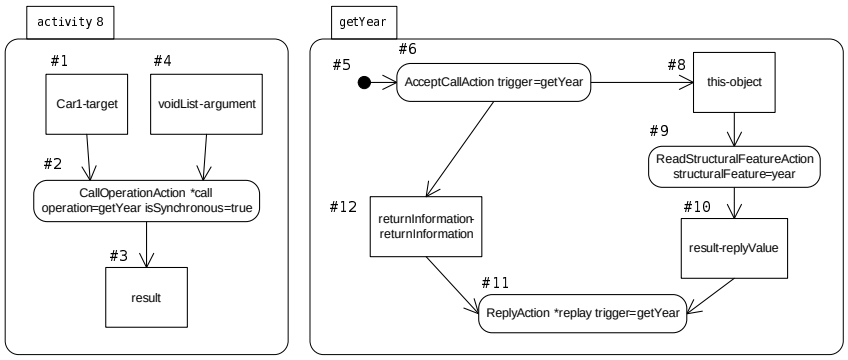


Figure B.37: Example activity to execute synchronous CallOperationAction, AcceptCallAction and ReplyAction

The ADLF representation of these activities are contained in Figures B.38 and B.39. The cd and obj files, used to initialize the universe before executing these activities, are shown in Figure B.40 and B.41. Output from a sample execution is shown in Figure B.42.

```
[Car1-target]
-> (CallOperationAction *call operation=getYear isSynchronous=true)
-> [result];

[voidList-argument]
-> (*call).
```

Figure B.38: ADLF representation of 'activity8' activity in Figure B.37

```
<InitialNode>
-> (AcceptCallAction trigger=getYear) {
    -> [this-object]
       -> (ReadStructuralFeatureAction structuralFeature=year)
       -> [result-replyValue]
       -> (ReplyAction *reply replyToCall=getYear),
    -> [returnInformation-returnInformation] -> (*reply)
}.
```

Figure B.39: ADLF representation of 'car_getYear' activity in Figure B.37

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

class Car
 attr year:Int
 op getYear():Int car_getYear.act

value voidList:Collect
```

Figure B.40: cd file

```
obj Car1:Car
 attr year=v2005
```

Figure B.41: obj file

```
Printing the order in which nodes were visited - (9)
[4, 1, 2, 5, 6, 9, 11, 2, 3]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #1[Car1-target] = Car1
.  #3[result] = v2005
.  #4[voidList-argument] = voidList : []
.  #8[this-object] = Car1
.  #10[result-replyValue] = v2005
.  #12[returnInformation-returnInformation]
.      = ReturnInfo[Message #710328602 SYNCHRONOUS
.      (init ---voidList : []---> Car1.getYearOp on thread T_3)]
.....................................................................

Running time = 491 ms = 0 sec = 0 min
```

Figure B.42: Partial output of a sample execution of activities in Figure B.37

B.9 Example 9: Send Signal and Accept Event

The activities in Figure B.43 demonstrate the execution of a SendSignalAction and AcceptEventAction. After the main 'activity9' activity starts, a StartClassifierBehaviorAction is executed to initiate the behaviour of the *Car1* object. Then, an 'initialize' signal is sent to the *Car1* object using the SendSignalAction. Then, the main diagram finishes execution. The second diagram represents the *Car* classifier behaviour; when started, it simply waits for an 'initialize' event to occur. When the signal is received, this activity simply clears the *year* structuralFeature of the owning object.[1] The sending of signals is an asynchronous invocation of behaviour; there will no value returned to the calling object.

[1]We use the 'this' object node as shorthand for the result of a ReadSelfAction.

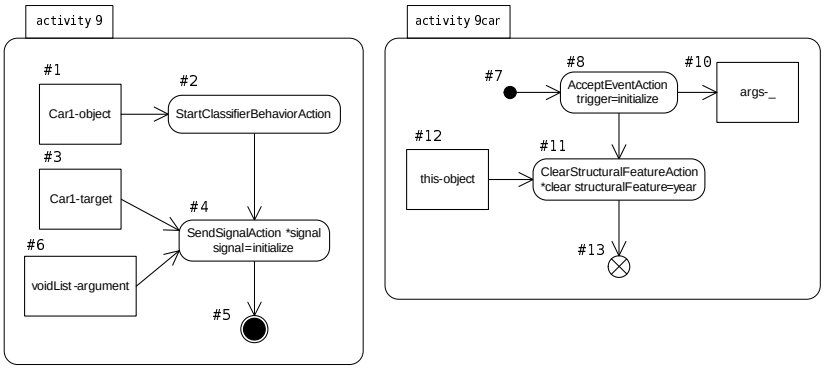


Figure B.43: Example activities to execute SendSignalAction and AcceptEventAction

The ADLF representation of these activities are contained in Figures B.44 and B.45. The cd and obj files, used to initialize the universe before executing these activities, are shown in Figure B.46 and B.47. Output from a sample execution is shown in Figure B.48.

```
[Car1-object] -> (StartClassifierBehaviorAction);

[Car1-target]
-> (SendSignalAction *signal signal=initialize)
-> <ActivityFinal>;

[voidList-argument] -> (*signal).
```

Figure B.44: ADLF representation of 'activity9' activity in Figure B.43

```
<InitialNode> -> (AcceptEventAction trigger=initialize) {
-> [args-_],
-> (ClearStructuralFeatureAction *clear structuralFeature=year)
};

[this-object] -> (*clear) -> <FlowFinal>.
```

Figure B.45: ADLF representation of 'activity9car' activity in Figure B.43

```
type Int
 value v2000
 value v2001
 value v2002
 value v2003
 value v2004
 value v2005
 value v2006
 value v2007
 value v2008

type Bool
 value vTrue
 value vFalse

class Car
 act activity9car.act
 attr year:Int

value voidList:Collect
```

Figure B.46: cd file

```
obj Car1:Car
 attr year=v2005
```

Figure B.47: obj file

```
Printing the order in which nodes were visited - (12)
[6, 5, 1, 2, 3, 4, 12, 7, 8, 11, 10, 13]

.....................................................................
.  Printing information about the values of the object nodes in the graph(s)
.  #1[Car1-object] = Car1
.  #5[Car1-target] = Car1
.  #6[voidList-argument] = voidList : []
.  #10[args-_] = voidList : []
.  #12[this-object] = Car1
.....................................................................

Running time = 651 ms = 0 sec = 0 min
```

Figure B.48: Partial output of a sample execution of activities in Figure B.43

Appendix C

ADLF

In this appendix, we describe the Activity Diagram Linear Form (ADLF), a modification of the Petri Net Linear Form (PNLF) [49], designed to render UML activity diagrams as human-readable text. When first used [53, 52], the language did not even have a name, let alone a formal grammar. That said, the language was well-described using examples, and we chose to use it as an efficient way to provide input to our interpreter. We therefore created a grammar and a parser for the language.

Our grammar is described in Extended Backus-Naur Form (EBNF) [41] and our parser were created using JavaCC [44], which resulted in an LL(1) parser [54] that works in a 'bottom-up' fashion. At the same time, a second grammar and parser (using Flex [32] and Bison [1]) was created independently. This second approach resulted in a LR parser that works in a 'top-down' fashion. In addition, Flex/Bison supports only 'plain' Backus-Naur Form (BNF) [28]. These differences in the two approaches result in two different grammars. In this document, we present our EBNF grammar for ADLF. The BNF grammar, along with further discussions of ADLF and PNLF, can be found at [31].

C.1 Motivation

Our grammar and parser for ADLF were created with a specific purpose in mind, i.e., to provide input to our interpreter. As such, there was no attempt to ensure that the grammar supports all of UML activities. Instead, we created our ADLF grammar to represent that subset of activities supported by our interpreter. For example, ADLF cannot be used to represent an activity using partitions, interruptible regions, etc. On the other hand, in some ways ADLF is more general than our interpreter. For instance, ADLF can be used to represent guards on any edge, as well as multiplicities.

A UML activity is a directed graph. ADLF can be used to represent that two-dimensional construct in a linear form, by slicing that graph into several segments, each representing a 'path' through the graph. Consider the activity in Figure C.1; the ADLF for this example is shown in Figure C.2. The activity has been sliced into three paths; in the diagram they are represented with various line weights/forms. In the ADLF, each path is shown separated by whitespace. Other examples of activity diagrams with associated ADLF representations are available in Appendix B.

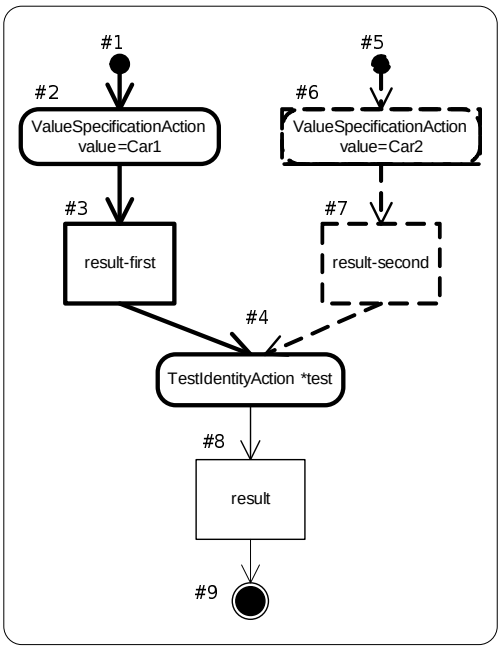


Figure C.1: Sample activity with slices differentiated with line weights

```
<InitialNode>
  -> (ValueSpecificationAction value=Car1)
  -> [result-first]
  -> (TestIdentityAction *test);

<InitialNode>
  -> (ValueSpecificationAction value=Car2)
  -> [result-second]
  -> (*test);

(*test) -> [result] -> <ActivityFinal>.
```

Figure C.2: ADLF representation of activity in Figure C.1

Some points to note about our ADLF grammar and how the resulting .act files are interpreted:

- Slices or paths through the activity are separated a ';'. Using a '.' indicates the end of the activity description.
- It is possible to have several different ADLF representations of the same activity, depending on how the graph is sliced.
- The interpreter assigns a numeric identifier to each node encountered during parsing—these are the numbers shown in the various activity examples. The number indicates the order in which that node was encountered during parsing.

- It is possible to use the *MultipleOutgoing* non-terminal to set up branching from a node. These non-terminals are considered 'nodes' by the parser, but they are elided from the resulting graph. This explains why sometimes a number seems to be missing in the sequence of graph node identifiers.

C.2 EBNF Grammar for ADLF

The following EBNF grammar describes ADLF. The following conventions are used:

- Tokens are enclosed in angle brackets, e.g., `<END>`.
- Non-terminals are not decorated, e.g., ActionNode.
- Optional items are enclosed in parenthesis, followed by a question mark, e.g., (Label)?.
- Items repeating zero or more times are enclosed in parentheses, followed by an asterisk, e.g., (`<IDENTIFIER>`)*.
- Items repeating one or more times are enclosed in parentheses, followed by a plus sign, e.g., (FullStatement)+.
- Alternate choices are separated by a pipe, e.g, (Identifier `|` `<NUMBER>`).

C.2.1 Tokens

`" " | "\t" | "\n" | "\r" | "\f" | "\013"`	(Whitespace ignored)
`"//..."`	(Java single-line comments ignored)[1]
`"/*...*/"`	(Java multi-line comments ignored)[1]
`"InitialNode"`	`<INITIAL>`
`"ForkNode"`	`<FORK>`
`"JoinNode"`	`<JOIN>`
`"DecisionNode"`	`<DECISION>`
`"MergeNode"`	`<MERGE>`
`"ActivityFinal"`	`<ACTIVITYFINAL>`
`"FlowFinal"`	`<FLOWFINAL>`
`";"`	`<SPLIT>`
`"->"`	`<SEQ>`
`"."`	`<END>`
`"{"`	`<PARBEGIN>`
`"}"`	`<PAREND>`
`","`	`<PARSPLIT>`
`(["0"-"9"])+`	`<NUMBER>`
`".."`	`<DOTDOT>`
`"*"`	`<STAR>`

`((["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[])+ |`
`(["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[])`
`(["a"-"z","A"-"Z","0"-"9","_","-"]|"\\"~[])*`
`(["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[]))`
	`<IDENTIFIER>`

`"\"" ( ~["\"","\\"] | "\\"~[] )* "\""`	`<DOUBLEQUOTED_STRING>`
`"'" ( ~["'","\\"] | "\\"~[] )* "'"`	`<SINGLEQUOTED_STRING>`
`"(^" ( "^"~[")"] | ~["^"] )* "^)"`	`<ANNOTATION_STRING>`
`<EOF>`	(End of file)

C.2.2 Grammar

The start symbol is Activity.

ActionName ::=	Identifier
ActionNode ::=	`"("` (ActionNodeInitial | ActionNodeReferral)
ActionNodeInitial ::=	ActionName (Label)? (AnnotationSeq)? `")"`

[1]The description of these tokens requires several lines in the JavaCC `jj` file. Essentially, single-line comments terminate with a carriage return; multi-line comments terminate with a closing `*/`. Any symbol is permissible within either type of comment.

ActionNodeReferral ::=	Label (AnnotationSeq)? `")"`
Activity ::=	(FullStatement)+ `<END>` `<EOF>`
ActivityFinalNode ::=	`<ACTIVITYFINAL>` (Label)? (AnnotationSeq)?
Annotation ::=	(Identifier (`"="` (Identifier | `<NUMBER>`))? | `<ANNOTATION_STRING>`)
AnnotationSeq ::=	(Annotation)+
ControlNode ::=	(ControlNodeInitial | ControlNodeReferral)
ControlNodeInitial ::=	(JoinNode | MergeNode | ActivityFinalNode | FlowFinalNode) `">"`
ControlNodeReferral ::=	Label (AnnotationSeq)? `">"`
DecisionNode ::=	`<DECISION>` (Label)? (AnnotationSeq)?

DecisionStatement ::=	DecisionNode `">"`
	`<PARBEGIN>`
	StatementSeq (`<PARSPLIT>` StatementSeq)*
	`<PAREND>`

FlowFinalNode ::=	`<FLOWFINAL>` (Label)? (AnnotationSeq)?
ForkNode ::=	`<FORK>` (Label)? (AnnotationSeq)?

ForkStatement ::=	ForkNode `">"`
	`<PARBEGIN>`
	StatementSeq (`<PARSPLIT>` StatementSeq)*
	`<PAREND>`

FullStatement ::=	(`"<"` (InitialNode `">"` (MultipleOutgoing)? | ControlNodeReferral) | ObjectNode | ActionNode) (StatementSeq (`<SPLIT>`)?)?
Guard ::=	Identifier
Identifier ::=	(`<IDENTIFIER>` | `<SINGLEQUOTED_STRING>` | `<DOUBLEQUOTED_STRING>`)
InitialNode ::=	`<INITIAL>` (Label)? (AnnotationSeq)?
JoinNode ::=	`<JOIN>` (Label)? (AnnotationSeq)?
Label ::=	`"*"` (Identifier | `<NUMBER>`)

```
Lower ::=                <NUMBER>

MergeNode ::=            <MERGE> (Label)? (AnnotationSeq)?

MultipleOutgoing :: =    <PARBEGIN>
                         StatementSeq ( <PARSPLIT> StatementSeq )*
                         <PAREND>

Multiplicity ::=         ( Lower <DOTDOT> )? UpperOrOnly

ObjectNode ::=           "[" Identifier (Label)? (AnnotationSeq)? "]"

Statement ::=            ( "<" ( ForkStatement
                               | DecisionStatement
                               | ControlNode )
                         | ActionNode (MultipleOutgoing)?
                         | ObjectNode (MultipleOutgoing)? )

StatementSeq ::=         ( (Guard)? <SEQ> (Multiplicity)? Statement )+

UpperOrOnly ::=          ( <STAR> | <NUMBER> )
```

Appendix D

cd and obj Files

This appendix contains the Extend Backus-Naur Form (EBNF) [41] grammars for the cd and obj files, which are used to describe the structure of objects manipulated as an activity is executed. The cd file basically describes a simple class diagram, including classifier names and attributes; values that are expected to exist are also included. The (optional) obj file describes the set of object instances that exist at the beginning of an activity execution. Examples of cd and obj files are included with the activity execution examples in Appendix B.

D.1 Conventions

The following conventions are used with our EBNF grammars:

- Tokens are enclosed in angle brackets, e.g., `<EOL>`.
- Non-terminals are not decorated, e.g., CdFile.

- Optional items are enclosed in parenthesis, followed by a question mark, e.g., (ActLine)?.
- Items repeating zero or more times are enclosed in parentheses, followed by an asterisk, e.g., (<IDENTIFIER>)*.
- Items repeating one or more times are enclosed in parentheses, followed by a plus sign, e.g., (Argument)+.
- Alternate choices are separated by a pipe, e.g, (ValueName | VariableName).
- Whitespace, specifically blank lines, is used to separate elements. For example, a class description is complete when a blank line is encountered.

D.2 EBNF Grammar for cd Files

D.2.1 Tokens

`" " | "\t" | "\f" | "\013"`	(Except for carriage return, whitespace ignored)
`"\n" | "\r\n"`	<EOL>
`"#..."`	(Java single-line comments ignored)[1]
`"type"`	<TYPE>
`"value"`	<VALUE>
`"class"`	<CLASS>
`"attr"`	<ATTR>
`"extends"`	<EXTENDS>
`"var"`	<VAR>
`"op"`	<OP>
`"act"`	<ACT>
`(["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[])+ |`	
`(["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[])`	

[1]The description of this token requires several lines in the JavaCC `jj` file. Essentially, single-line comments terminate with a carriage return. Any symbol is permissible within either type of comment.

```
  (["a"-"z","A"-"Z","0"-"9","_","-"]|"\\"~[])*
  (["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[]))      <IDENTIFIER>
 <EOF>                                          (End of file)
```

D.2.2 Grammar

The start symbol is CdFile.

ActLine ::=	`<ACT>` FileName (`"["` (Argument)+ `"]"`)? `<EOL>`
Argument ::=	VariableName `":"` ClassOrTypeName (`","`)?
AttrLine ::=	`<ATTR>` VariableName `":"` ClassOrTypeName `<EOL>`
CDFile ::=	(`<EOL>`)* (Type)* (`<EOL>`)* (Class)* (`<EOL>`)* ((FullValueLine `|` VarLine `|` StandaloneOpLine) (`<EOL>`)*)* (`<EOL>`)* `<EOF>`
Class ::=	ClassSig (ActLine)? (AttrLine)* (OpLine)* `<EOL>`
ClassOrTypeName ::=	`<IDENTIFIER>`
ClassSig ::=	ClassOrTypeName (`<EXTENDS>` ClassOrTypeName)? `<EOL>`
FileName ::=	`<IDENTIFIER>` `"."` `<ACT>`
FullValueLine ::=	`<VALUE>` ValueName `":"` ClassOrTypeName `<EOL>`
OperationName ::=	`<IDENTIFIER>`
OpLine::=	`<OP>` OperationName `"("` (Argument)* `")"` `":"` ClassOrTypeName FileName (`"["` (Argument)+ `"]"`)? `<EOL>`
StandaloneOpLine ::=	`<OP>` OperationName `"("` (Argument)* `")"` `":"` ClassOrTypeName FileName `<EOL>`

Type ::=	TypeSig (ValueLine)* <EOL>
TypeSig ::=	<TYPE> ClassOrTypeName <EOL>
ValueLine ::=	<VALUE> ValueName <EOL>
ValueName ::	<IDENTIFIER>
VariableName ::=	<IDENTIFIER>
VarLine ::=	<VAR> VariableName ":" ClassOrTypeName <EOL>

D.3 EBNF Grammar for obj Files

D.3.1 Tokens

" " | "\t" | "\f" | "\013"	(Except for carriage return, whitespace ignored)
"\n" | "\r\n"	<EOL>
"#..."	(Java single-line comments ignored)[1]
"obj"	<OBJ>
"attr"	<ATTR>
"collection"	<COLLECTION>
(["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[])+ | (["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[]) (["a"-"z","A"-"Z","0"-"9","_","-"]|"\\"~[])* (["a"-"z","A"-"Z","0"-"9","_"]|"\\"~[]))	<IDENTIFIER>
<EOF>	(End of file)

D.3.2 Grammar

The start symbol is ObjFile.

AttrLine ::=	<ATTR> VariableName "=" ValueName <EOL>

[1]The description of this token requires several lines in the JavaCC `jj` file. Essentially, single-line comments terminate with a carriage return. Any symbol is permissible within either type of comment.

ClassName ::=	`<IDENTIFIER>`
Collection ::=	CollectionSig (CollectionEntry)* `<EOL>`
CollectionEntry ::=	ValueName `<EOL>`
CollectionSig ::=	`<COLLECTION>` ValueName `<EOL>`
Object ::=	ObjectSig (AttrLine)* `<EOL>`
ObjectName ::=	`<IDENTIFIER>`
ObjSig ::=	`<OBJ>` ObjectName `":"` ClassName `<EOL>`
ObjFile ::=	(`<EOL>`)* (Object)* (`<EOL>`)* (Collection)* (`<EOL>`)* `<EOF>`
ValueName ::=	`<IDENTIFIER>`
VariableName ::=	`<IDENTIFIER>`

Index

Symbols

A

B

C

T

U

V

W