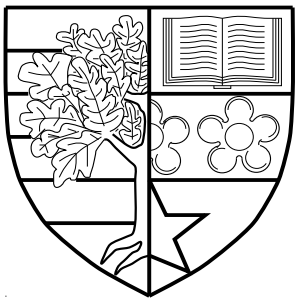
Action Semantics of Unified Modeling Language

by

Mikai Yang



Submitted for the Degree of

Doctor of Philosophy

on completion of research in the

Department of Computing Sciences

School of Mathematical and Computing Sciences

Heriot-Watt University

July 2009

Declaration

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, except where due acknowledgement is made, and not been submitted for any other degree.

Mikai Yang (Candidate)

Prof. Rob Pooley & Prof. Greg Michaelson (Supervisors)

Date

Abstract

The Unified Modeling Language or UML, as a visual and general purpose modeling language, has been around for more than a decade, gaining increasingly wide application and becoming the de-facto industrial standard for modeling software systems. However, the dynamic semantics of UML behaviours are only described in natural languages. Specification in natural languages inevitably involves vagueness, lacks reasonability and discourages mechanical language implementation. Such semi-formality of UML causes wide concern for researchers, including us.

The formal semantics of UML demands more readability and extensibility due to its fast evolution and a wider range of users. Therefore we adopt Action Semantics (AS), mainly created by Peter Mosses, to formalize the dynamic semantics of UML, because AS can satisfy these needs advantageously compared to other frameworks.

Instead of defining UML directly, we design an action language, called ALx, and use it as the intermediary between a typical executable UML and its action semantics. ALx is highly heterogeneous, combining the features of Object Oriented Programming Languages, Object Query Languages, Model Description Languages and more complex behaviours like state machines. Adopting AS to formalize such a heterogeneous language is in turn of significance in exploring the adequacy and applicability of AS.

In order to give assurance of the validity of the action semantics of ALx, a prototype ALx-to-Java translator is implemented, underpinned by our formal semantic description of the action language and using the Model Driven Approach (MDA). We argue that MDA is a feasible way of implementing this source-to-source language translator because the cornerstone of MDA, UML, is adequate to specify the static aspect of programming languages, and MDA provides executable transformation languages to model mapping rules between languages.

We also construct a translator using a commonly-used conventional approach, in

which a tool is employed to generate the lexical scanner and the parser, and then other components including the type checker, symbol table constructor, intermediate representation producer and code generator, are coded manually. Then we compare the conventional approach with the MDA. The result shows that MDA has advantages over the conventional method in the aspect of code quality but is inferior to the latter in terms of system performance.

Acknowledgements

I am indebted to a great number of people for their teaching, help, guidance, encouragement and inspiration throughout the works described in this thesis, including this document itself. Firstly, I feel incredibly fortunate to work with Professor Rob Pooley, my first supervisor. My sincere thanks go to him for his continuously inspiring me and granting me the freedom to explore new and varied (sometimes romantic-seeming) topics. His valuable insight into some problems we come up with and his timely encouragement are vital in forming the research topic, and his continuing patient and efficient mentorship are of great significance in boosting the proceeding of the research. In addition, this research could not be accomplished smoothly without his great efforts in securing a stable financial support.

Secondly, I am deeply grateful to Professor Greg Michaelson, my second supervisor, for his rich knowledge and experiences in the subject concerned, his systematic mentorship in the development of this search, assisting me in time management of the milestones, and his great consideration and patience exhibited when I was tangled with difficulties, in both research and personal issues. The seamless and pleasurable collaboration of the two supervisors is highly creditable, which established a favourable atmosphere for the whole process and does huge good to the accomplishment of this research.

Thirdly, I would like to express gratitude to Staff in the Department of Computer Science for amazing support, including faculty, staff and students. My special thanks go to Iain A. McCrone, who helped me a great deal in installing necessary software tools, especially in installing the Action Environment tool which is critical to test a part of the action semantics description. This work also benefited immensely from the ideas, comments, feedback, food for thoughts contributed by fellow PhD students and attendees of my presentations. Their names are not listed here but borne in my mind.

I hope that this doesn't compromise the genuineness of my thanks to them.

Finally, I do not know how I can thank my parents for their selfless and boundless love as well as their great support. They always talked to me with encouraging words that turned into the inexhaustible power driving me on all this way. What I can do is to dedicate this work to them to express my limitless gratitude to them.

Table of Contents

List of Figures

Chapter 1

Introduction and Motivation

1.1 Problem Statement

Problem 1: UML is semi-formal.

The Unified Modeling Language (UML) is a general-purpose graphical modeling language that is widely applied in system design and documentation. Previously, UML lacked sufficient expressivity in describing dynamic aspects of systems, such as method bodies and exit/entry actions of state machines, and as such has to resort to a plain natural language or an existing programming language as a complementary formalism. These two compromise methods have their drawbacks: 1) using natural languages causes ambiguities in the description and hampers rigorous model checking and early system simulation; 2) using programming languages usually involves unnecessary implementation-specific details and requires a related background of the user. To address this problem, UML Precise Action Semantics (UPAS) [60] has been incorporated into UML 2.0 [78, 79] to provide precise behavioural primitives such that large-scale realistic behaviours can be built systematically based on them. Consequently, UML has been made more expressive in defining the dynamic parts of the system, and thus becomes executable.

However, although the syntax and the static semantics of UML have been formally specified using the MetaObject Facility (MOF) [77] in a four-layer metamodeling framework facilitated by the Object Constraint Language (OCL), UPAS and the behaviours of UML such as state machines, activities and interactions, are only standardized in English. It is well-known that natural language inevitably involves vagueness and is

hard to reason about. So it is desired to provide a formal semantics for this aspect of UML.

Problem 2: is Action Semantics applicable to a very hybrid language?

Action Semantics (AS) is a hybrid semantic description framework incorporating the advantages of denotational semantics, structural operational semantics and algebraic specification. It defines as the major semantic entities a set of actions whose execution semantics are well-defined using structural operational semantics. To describe the semantics of a language, one only needs to be concerned with translating the constructs in this language to the appropriate actions or other semantic entities like yielders and data. The translations are expressed in semantic functions defined by semantic equations. Furthermore, AS provides some ready-to-use predefined data sorts so that users can easily import some of them in their description for efficiency. Flexibly, users are also allowed to define their own sorts depending on the languages being described. The notations of actions are carefully related to meaning-suggestive English words and phrases to achieve superior comprehensibility.

Compared to other semantic formalism such as denotational semantics and operational semantics, AS enjoys better readability and extensibility, and it has been successfully used to describe a wide variety of real programming languages such as standard ML, Pascal, Java, ADA and ANDF-SF [104, 68, 21, 65, 38]. However, the problem is that, to the best of our knowledge, it has never been applied to a hybrid language like an executable UML which basically comprise model descriptive constructs, common imperative constructs, object query constructs and some complex behaviours such as state machines and link navigations. Therefore it is of significance to explore Action Semantics in such a hybrid language to test its adequacy and applicability.

1.2 Research Overview

In this research we employ AS as the formalism for UML in the hope that we can cope with the two problems in one go: formalizing UML and exploring the expressivity of AS. The adoption of AS as the vehicle is also attributed to the formal semantics of

UML demanding more readability and extensibility because of its fast evolution and a wide range of users, while AS has required properties superior to other frameworks, including operational semantics [86] and denotational semantics [92].

Our approach to formalizing UML has a distinctive feature: instead of defining UML directly, we design an Action Language (AL) [59] and use it as the intermediary between UML and action semantics of UML. The designed AL, called ALx, is bigger than the existing ALs because aside from the part for the common functionality available in the currently-used ALs, it also incorporates a model-describing part that can be viewed as the textual counterpart of the graphical UML. On the one hand, the two parts are integrated together seamlessly to form a textual and computationally-complete modeling language; on the other hand, the former part itself can be embedded in graphical UML models to specify method bodies and activities in state machines. So we provide the formal semantics of UML by first composing the action semantics of the intermediary ALx, and then specifying a formal mapping model between UML and ALx.

To observe the behaviours of the action semantics of xUML, we also construct a source-to-source translator, called xUML-to-Java translator, which takes UML models as input and yield executable Java code. We first adopt MDA as the best potential approach to implement the translator based on the composed action semantics of UML. MDA has been widely employed in constructing business system but rarely used in language implementation, so this experiment is intended to serve two purposes: testing the action semantics of ALx and investigating the applicability of MDA to language implementation. Furthermore, we also use a conventional method to implement the same translator. Subsequently, a comparison is conducted between the MDA and the conventional approach to further look into the pros and cons of MDA over the conventional approach in language implementation.

1.3 Contributions

The contribution of this work can be summarised as follows:

- We harness the framework of Action Semantics to indirectly specify a self-defined xUML by formalizing its textual counterpart ALx. This attempt shows that AS is expressively adequate to formalize a heterogeneous language like ALx and the

yielded action semantics of the xUML is readable, modular and reusable.

This work has been published as a conference paper in SBLP 2008 [108], and accepted by the Journal of Universal Computer Science.

- To give assurance of the validity of the action semantics description, we implement a prototype ALx-to-Java translator, underpinned by our formal semantic description of the action language. In the process, we find that the action semantics of ALx clearly suggests implementation logic.

- We argue that MDA is in itself applicable to implementing the aimed source-to-source language translator and also show this in practice by adopting MDA to building the xUML-to-Java translator.

- In order to know the advantages of MDA in implementing the translator, we also use a conventional approach to build the translator and conduct a comparison between the approaches. The comparison results show that MDA has advantages of reducing developing cost over the conventional approach. Such merit is especially valuable in prototyping programming languages.

1.4 Road Map

The rest of the dissertation is organized as follows:

- Chapter 2 presents the background and evolution of UML, and the historical attempts to formalize them.

- Chapter 3 provides an introduction to Action Semantics, preceded by introducing Denotational Semantics and Operational Semantics which Action Semantics combines.

- Chapter 4 presents the Action Semantics of some typical constructs of ALx accompanied by informal explanation.

- Chapter 5 describes the overall architecture of the translator which is intended for testing the composed Action Semantics of ALx, and highlights a Java library which implements a major part of the action semantics of ALx.

- Chapter 6 explains why MDA is the best potential approach to implementing the translator and the feasibility of MDA for this purpose, and then gives the structure of an MDA-based implementation as well as implementation techniques involved.
- Chapter 7 describes a conventional approach to implementing the translator and presents the comparative results of the two approaches.
- Chapter 8 concludes this thesis and outlines directions for future work.

Chapter 2

Background of UML and Formalizing UML

A solid design is of paramount importance in the development of a software system, especially when the project involved is a large enterprise application. Modeling, which is a major step of designing software systems prior to coding, has been attracting intensive attention of both industry and academia.

People usually prefer visual models to textual ones as representations of their design ideas, and desire that the visual models are adequately abstract, containing only the relevant details necessary to help them in conceptualization, understanding and communication.

Visual models have been applied in many fields like mechanical engineering (mechanical drawing) and electronic engineering (electrical schematics). Over decades, researchers continue moving software development from an artistic manner to an engineering one. An important effort was seeking a graphical formalism to facilitate the design process. The Unified Modeling Language (UML) [78, 79] was such an outcome.

The Unified Modeling Language (UML), "a visual and general-purpose modeling language for specifying, constructing and documenting the artefacts of systems" [78], has gained increasingly wide applications and has already become the de-facto industrial standard in modeling of software system. This, however, by no means indicates that UML is fully matured, being able to satisfy users in every particular. Instead, researchers have found several deficiencies of UML regarding its expressivity, formality, long learning curve and applicability. Reasonably, researchers are concerned with its

computational-completeness, as gives rise to the Precise Action Semantics [61] and Executable UML(xUML) [59]; researchers are also concerned with the formality of UML, thus many attempts have been conducted to formalize its semantics, including the one described in this thesis.

This chapter starts with a quick introduction to UML and its derivative, xUML followed by the introduction of UML Precise Action Semantics (UPAS) and its implementation: action languages. Then, the attempts made to formalize UML are surveyed, categorized and analyzed. Finally present is one of the motivations to this research. To be clear, in this thesis, UML 2.0 refers to the version of UML specified in the UML 2.0 Specification [78, 79], UML 1.5 refers to a version of UML that is specified in UML 1.5 Specification [82]. The 'UML' is also used where we talk about a general concept of UML, abstracted from the various versions.

2.1 Introduction to UML and Executable UML

2.1.1 UML Overview

UML is a typically collective achievement, and came into being by unifying three major object-oriented modeling methods: Booch's method, Jacobson's OOSE (Object-Oriented Software Engineering) and Rumbaugh's OMT (Object Modeling Technique), as well as other important methods such as Fusion, Shlaer-Mellor, and Coad-Yourdon [17]. In November 1997, UML was accepted by the Object Management Group (OMG) as a standard modeling language. Since then, it has been playing an increasingly important role in software intensive systems and nowadays dominates object-oriented modeling. As it is said in [17] that UML is a small hill atop a large mountain of previous experience, UML is continuously contributed to and enriched by practitioners and researchers worldwide. Over a decade, maintained by an OMG Revision Task Force, UML has evolved across versions 1.3, 1.4 and 1.5. Today, the latest version is UML 2.0 [78, 79] which is a major revision of UML 1.x.

UML is a rich language, so we do not exhaustively present the features of UML. The key features of UML can be summarized as follows:

- UML is a general purpose rather than a domain specific modeling language. Furthermore, not only can it be applied to software systems, but also to non-software

ones.

- UML adopts the object-oriented paradigm to analyze and specify a system.
- All aspects of a system can be specified in UML using appropriate parts.
- UML is a visual, graphical modeling language aimed to be superior to a textual one in the sense of intuitive understanding [28].

However, UML is not perfect. Some negative aspects of UML, such as incompleteness in computation and lack of formality in semantics are addressed respectively in Section 2.1.2 and Section 2.1.3.

As mentioned earlier, UML enables system modeling in all aspects of the system of interest. UML follows the traditional way to distinguish structural aspects and behavioural aspects, and offers various diagrams to cover these two aspects. Usually, a UML model for a system consists of a set of sub-models or views each of which concentrates on a specific system aspect.

To model the structural aspects of the system, for example, UML provides the class diagram, which specifies the type level of the system, and the object diagram, which focuses on the instance level.

1. A typical class diagram is shown in Figure 2.1. Class diagrams model static aspects of the system via classifying concrete objects using the construct *Class* as well as their structural features: *attributes* and *associations*. The *associations* describe the possible relationships among objects and are further specialized to *Aggregation* and *Composition*: Aggregation is to specify the whole/part relationship between objects, and Composition is a strong form of aggregation where the part is only contained by at most one whole and its lifecycle is controlled by the whole in such a way that when the whole is destroyed all its parts are also destroyed [78]. Class diagrams can be viewed as a part of the type system because, akin to primitive type enumeration type, and collection type, class definition is essentially a complex type definition.

2. An object diagram (See Figure 2.2) is employed to give a visual snapshot into the running system at a point in time, to see the states, the collaboration and the relationships of a set of selected objects [17]. See Figure 2.2. An object

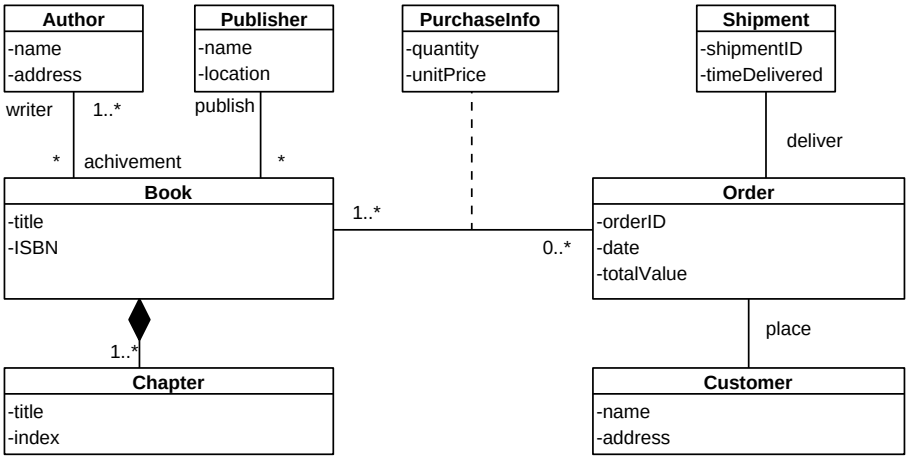


Figure 2.1: Example UML class diagram modeling static aspects of the system.

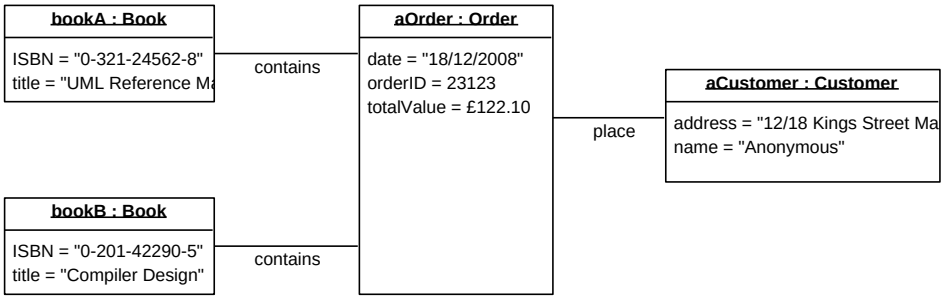


Figure 2.2: Example object diagram showing snapshots of the run-time system.

diagram contains primarily objects with attributes evaluated (slot), and links which connect objects and are defined as the instances of associations [78]. For each object, its name and type are also given. Notably, the object diagram reflects a system being frozen at a moment, which strongly implies that, despite the presence of links, there exist no messages passing between objects, and that when modeling system with object diagrams, the user must specify which frame this object diagram is intended for in the whole interaction storyboard of the system.

In addition to class diagrams and object diagrams, UML also provides other structural diagrams regarding implementation and deployment, such as component diagrams, which allow modeling a component-based system architecture, deployment diagrams, which, even further, incorporate hardware in an architectural level.

To model the behavioural aspects of the system of the system, UML offers mainly use case diagrams, state machine diagrams, sequence diagrams, collaboration diagrams, and activity diagrams [79].

1. *Use case diagrams* provide a global and coarse-grained view of the main function-

ality of the system. The functionalities in use case diagrams are restricted to the externally visible ones, and are only concerned with what functionality can be consumed in the system to the outwards, but not with how the functionality is implemented.

2. *State machine diagrams* or *state charts* depict the various states that an object may be in during its lifecycle, the transitions between those states, and behaviours in the states and in the transitions. A stateful object may receive events (signals, calls, timing event, etc) from itself or its context; the events may be simply ignored or trigger transitions of the state, which then possibly cause the execution of some behaviours such as transition actions, entry actions and exit actions. State machine diagrams provide insight into individual objects and hence are frequently referred to as *intra-object views* [79].

3. *Sequence diagrams* are the inter-object views of the system. In a sequence diagram, a selected set of objects is rendered and the interactions between them are also depicted by a sequence of events, which trigger the corresponding behaviours of the receiving objects. The behaviours are performed either synchronously or asynchronously depending on the kinds of events; for example, call events (invoking a method) cause the behaviours to be executed synchronously, while signal events cause asynchronously execution. Sequence diagrams place more emphasis on the time sequence of the interactions.

4. *Collaboration diagrams*, akin to sequence diagrams, are also a kind of inter-object view of the system. However, collaboration diagrams are more focused on the structural aspects of the participating objects and interactions, despite the fact that they reflect mainly the same information as sequence diagrams. Likewise, collaboration diagrams also present a selected set of objects, but the layout of interactions are structure-oriented, offering a clear view of how objects are interrelated and what interactions an object takes part in.

5. *Activity diagrams* are control-flow and data-flow based, primarily for coordinating behaviours residing in multiple objects, and their major graphical elements are nodes and directed edges. A node indicates an action, an activity, or a control such as decision and merge (for decision making and the merge of multiple possible

execution paths) , fork and join (for concurrency). An edge shows the activity has control-dependency or data-dependency on its preceding activities. That is, the performance of an action can be launched only when its preceding activity has completed in execution, or only when the required data become available.

To sum up, the behaviour-related diagrams in UML characterize the system in different ways: state machine diagrams are dedicated to inter-object behaviours; sequence diagrams and collaboration are mainly for realizing scenarios of use cases using different organizations of their constituents, the former emphasizing more on time sequence and the latter more on structure. Activity diagrams, however, are control flow and data flow oriented.

2.1.2 Executable UML

UML is a modeling language rather than an executable programming language. UML is not made executable in the first place because it is required to be adapted to various stages of modeling activities. Particularly, it must provide constructs to facilitate the design activities in the fairly early stages in the development lifecycle such as conceptualizing and sketching the system, when designers are primarily focused on what the system has to do rather than how that will be achieved. This separation of concerns has benefits in making designers concentrate on the system requirements to avoid being overwhelmed by too many implementation details, and enabling the reuse of design models for implementations on varying platforms. For these benefits, the constructs in UML must be adequately abstract but at the cost of computational completeness, which gives rise to the fact that UML is not executable.

However, UML is widely expected to be executable. Raising the level of abstraction is a major objective in software engineering. For example, programmers have moved from assembly languages to the higher-level languages such as C and Java for high efficiency. Today, researchers have a great hope that the abstraction level can be further raised to the model level. Unfortunately, the non-executability of UML conflicts with this expectation.

An executable UML is the result of attempting to make UML executable. See Figure 2.3. xUML is a subset of UML complemented by additional rigorously-defined constructs to specify those incomplete behaviours, such as operations lacking body and

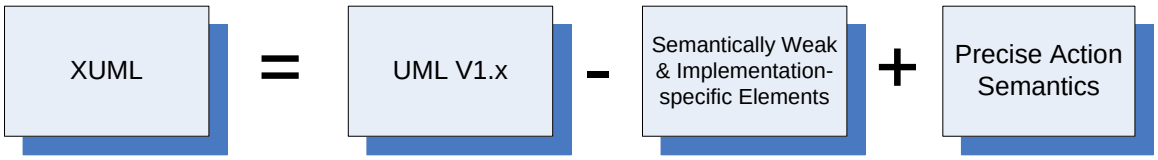


Figure 2.3: xUML: a rigorous subset of UML plus UMPAS

state machines lacking entry/exit activities. Additionally, ambiguous constructs are removed from UML. In other words, xUML is a subset of UML and intended to be a higher-level programming language than Java, C++, etc.

xUML has the following features. 1) xUML can be unambiguously and consistently interpreted by a human. 2) xUML can be executed by a machine, which also means xUML models can be simulated and validated in the earlier stages of system development. 3) xUML can be translated to a less abstract target language, enabling 100% code generation. Hopefully, xUML can be mapped to silicon.

Not being standardized, xUMLs vary depending on application domains, but, invariably, parallel to other programming languages like Java, SmallTalk and Ada, an xUML must provide the basic constructs to serve data declaration, data computation, execution sequence and concurrency. In addition, refinements must be made to UML so that it lacks ambiguities either in syntax or in semantics, and is independent of implementation technologies.

For example, the xUML proposed in [59] subsets UML in such a way. Class diagrams are employed for the declaration of data types. The operations on data, including those of built-in data types such as arithmetic computation and collection data structure handling, and also those of user-defined operators such as creating/deleting object, writing/reading attributes and navigating among objects, are specified in an action language. Class collaboration diagrams are used for declaring events. The role of control flow and concurrency is played by the confluence of sequence diagrams, state machines, the implicit event mechanism, and the operation calls in action language. The identified subset is then refined to remove ambiguities and implementation-specific details, like naming every association uniquely, abandoning composition and aggregation, restricting multiplicities on associations to 1 (exactly one), 0..*(zero-to-many), 0..1 (zero-to-one), 1..* (one-to-many), and adopting only highly abstract data type as pre-defined data types such as integer, real, boolean, string, date and timestamp. Simply speaking, xUML can be described by Figure 2.3 [43].

xUML tools have been around for a decade, for example iUML from Kennedy Carter [107], Kabira from Kabira Design Center [47] and BridgePoint Development Suite from Project Technology [87]. They are distinguished from each other because their underlying xUMLs, despite sharing similar concepts, are not completely the same either in syntax or in execution semantics, varying according to their targeted application domains.

2.1.3 UPAS & Action Languages

As mentioned, for turning a UML model into an xUML model, namely for making it executable, its behaviours are required to be defined in sufficient detail. For this purpose, the earlier versions of UML (the versions before UML 1.5) refer the user to either plain natural languages like English or existing implementation languages such as Java and C++. Both of the approaches seem quick but are not elegant. Using English inevitably gives rise to ambiguities, resulting in the possibility that the behaviours may not be consistently or precisely interpreted by human. Furthermore, despite its expressivity, English is not a machine-understandable language, which disables automated reasoning, early simulation and execution of the models, and code generation. Using an implementation language one has to assume that the readers of the model have knowledge of this language, and it also implies over-specification of models because implementation languages contain details unnecessary in the stage of modeling, compromising the expected enhancement of level of abstraction. Last but not least, such lack of a common formalism in describing behaviours hinders the compatibility and cooperation between tools.

As such, from UML 1.5, UPAS [4, 5] including action models and activity models are incorporated into UML so that the behaviours can be specified completely, formally, high abstractly and implementation independently. UPAS has become key to make UML executable.

The action model of UPAS defines a set of actions which are non-decomposable and fundamental units of behaviours. One action represent a single and primitive step in the execution, and all user-defined behaviours, including activities, state machines and interactions, are built upon these basic actions, and the effects of their execution, like writing attributes and triggering state transitions, are ultimately caused by the con-

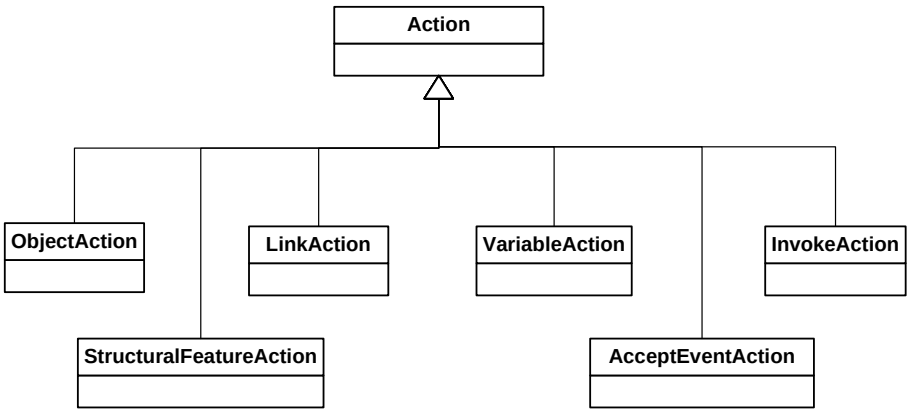


Figure 2.4: Overall classification of actions.

stituent actions. The defined actions in UML2.0 [79] includes essentially the following. See Figure 2.4 for an overall classification of actions.

Object Actions

Object Actions (See Figure 2.5) includes principally CreateObjectAction and DestroyObjectAction. Given a class, CreateObjectAction creates an object of this class and returns it as output without other effects, such as invoking a behaviour, generating an event or initializing attributes. DestroyObjectAction is the opposite: it destroys an object given as its input, similarly without other effects [79]. In addition, StartClassifierBehaviorAction starts the execution of the classifier behaviour of the given object if the object has one (Note that each object has at most one classifier behaviour which may be an activity, a state machine or an interaction). ReadIsClassifiedObjectAction determines whether the dynamically given object is an instance of the given classifier. ReclassifyObjectAction, as its name implies, is intended to modify the type aspect of a dynamically given object by adding the given new classifiers and removing the given previous classifiers. Note in UML, multi-classification of objects is supported.

Structural Feature Actions

Structural Feature Actions (See Figure 2.6) are used to read, write, and clear the values of the structural features of objects, such as attributes and association ends, fulfilled primarily by WriteStructuralFeatureAction, ReadStructuralFeatureAction and ClearStructuralFeature. For multi-valued structural features, AddStructuralValueAction and RemoveStructuralFeatureValueAction, sub-actions of WriteStructuralFeature-

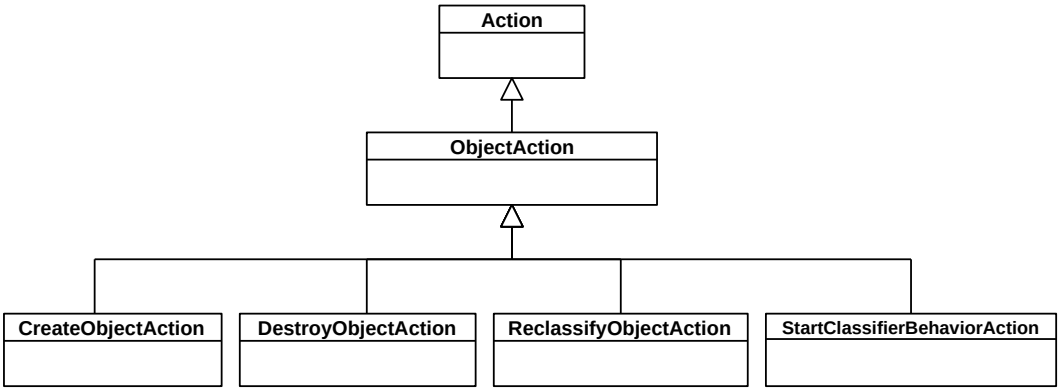


Figure 2.5: Classification of Object actions

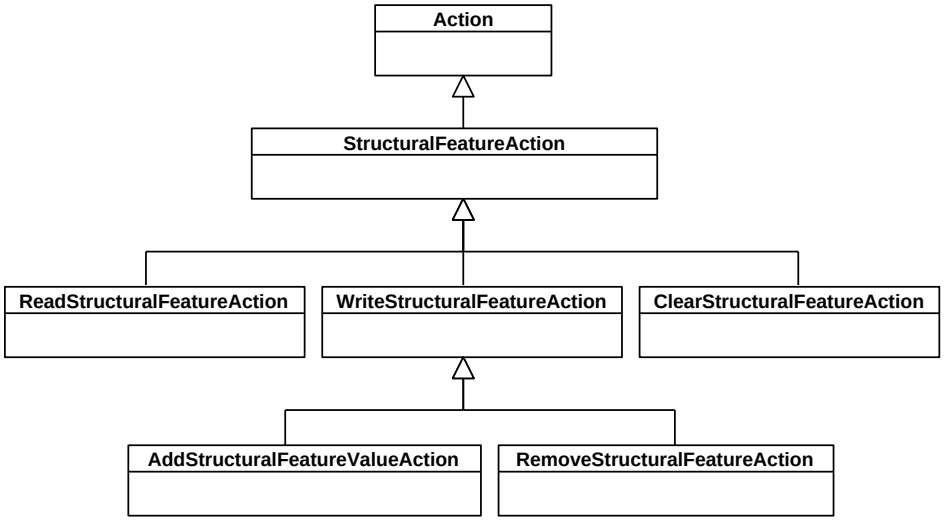


Figure 2.6: Classification of StructuralFeatureAction

Action, are provided to add or remove a member to or from the value collection.

Link Actions

Links are the instances of associations, and the communication bridges between objects: the run-time configuration of a system can be imagined as a community of objects that are connected by links, and their communications, such as data transfer and event passing, are conducted over the links, which implies that two objects without link connections cannot communicate.

CreateLinkAction (See Figure 2.7) creates a link of a given association across the given objects; DestroyLinkAction deletes a link of a given association across the given objects. Moreover, ReadLinkAction navigates an association from a specified source link end (an object) to zero, one or more target objects. The likelihood of returning multiple objects is due to the fact that one object is potentially linked with multiple

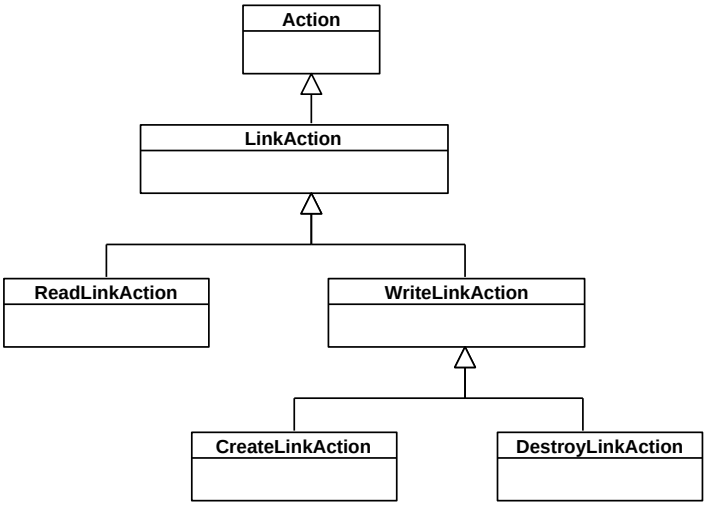


Figure 2.7: Classification of LinkAction

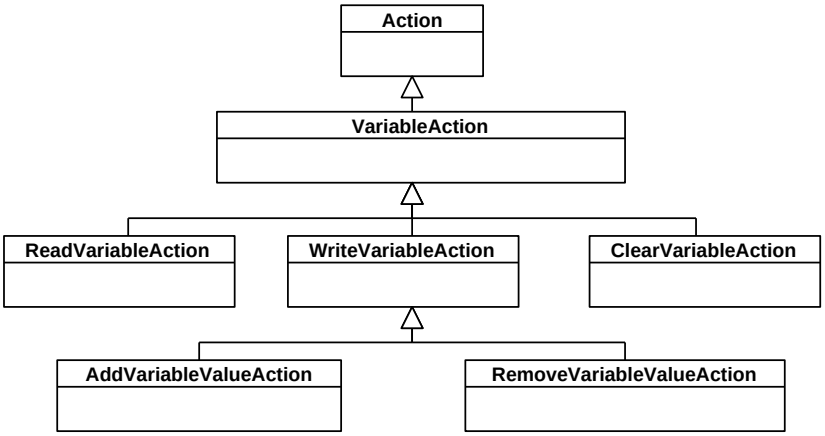


Figure 2.8: Classification of VariableAction

objects with one association. More capably, Link Actions can specify a qualifier so that solely the qualified target objects are manipulated.

Not only can Link Actions deal with links, but the Structural Feature Actions can also create, destroy or read links, because associations are a kind of structural feature as well. For simplicity, the overlap of semantics of different constructs are circumvented to the best in our research.

Variable Actions

Besides data being able to be passed between actions through data flow, variables enable passing data indirectly by storing values shared by the actions within a group. Variable can be read, written and cleared by ReadVariableAction, WriteVariableAction and ClearVariableAction. See Figure 2.8.

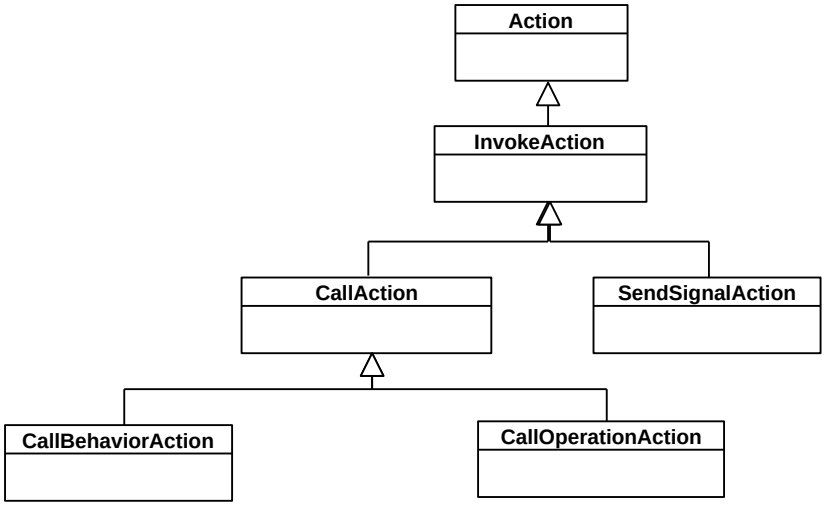


Figure 2.9: Classification of InvokeAction

Invocation Action

Three basic invocation actions (See Figure 2.9) are available: CallOperationAction, CallBehaviorAction and SendSignalAction. CallOperationAction sends a call event to the target actions, which then may trigger the execution of the specified behavioural feature (like operation) that, in turn, result in the invocation of the behaviour implementing the feature. CallBehaviorAction, however, invokes the specified behaviour directly without using a behavioural feature. SendSignalAction sends a signal event to the target, which then may respond to the event by firing a state transition or invoking an activity. The behaviours invoked by the call actions are executed either synchronously or asynchronously, depending on the call type; however, SendSignalAction causes the behaviours to be executed asynchronously

Accept Event Action

AcceptEventAction (See Figure 2.10) can be regarded as an element to enforce a join point in the concurrency. If the execution of a behaviour reaches a point that is an accept event action, then execution is blocked until the needed event occurs. CallEventAction specializes AcceptEventAction, but it waits for a call event rather than a signal event. CallEventAction may be followed by ReplyAction which is responsible for sending return values back to the caller.

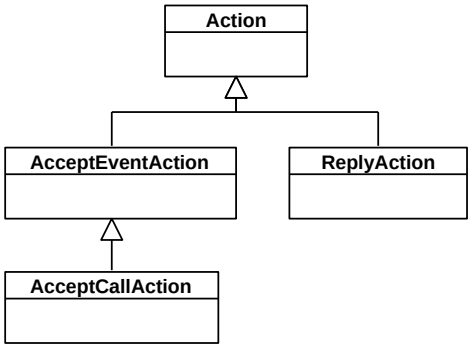


Figure 2.10: Classification of AcceptEventAction

Computation Action

UPAS in UML 2.0, unlike that in UML 1.5, has not specified two kinds of actions:

- computation actions, which mainly embody mathematics functions or string operations, and do not interact with instances.
- collection actions, which mainly operate on implementation-unspecific collection data structures (like Bag, Set and Sequence) to retrieve elements, join collections and iterate elements.

However, these actions are thought to be implicitly included in UML 2.0. OMG does not explicitly define them because, in our opinion, their semantics are common amongst programming languages, well-understood and precise.

2.1.3.1 Activity Model

As mentioned, each action represents merely an individual step in execution; they must be organized in a way to construct meaningful behaviours. In UML, the organization of actions is fulfilled by an activity model. The UML Specification [79] also states that actions must be directly contained in an activity. Activities are specified in the activity diagram, where actions are rendered as a kind of activity node, denoted by rounded rectangles with a name, and connected by directed edges with other nodes including actions, control nodes and object nodes. See Figure 2.11 for illustration. Control nodes controls the path and concurrency of execution and include mainly Decision Nodes, Merge Nodes, Fork Nodes and Join Nodes. Data nodes mean a temporary storage of data and are used to hold the data produced by actions until the following actions become active and consume it. Data can also come from parameter passing (activities,

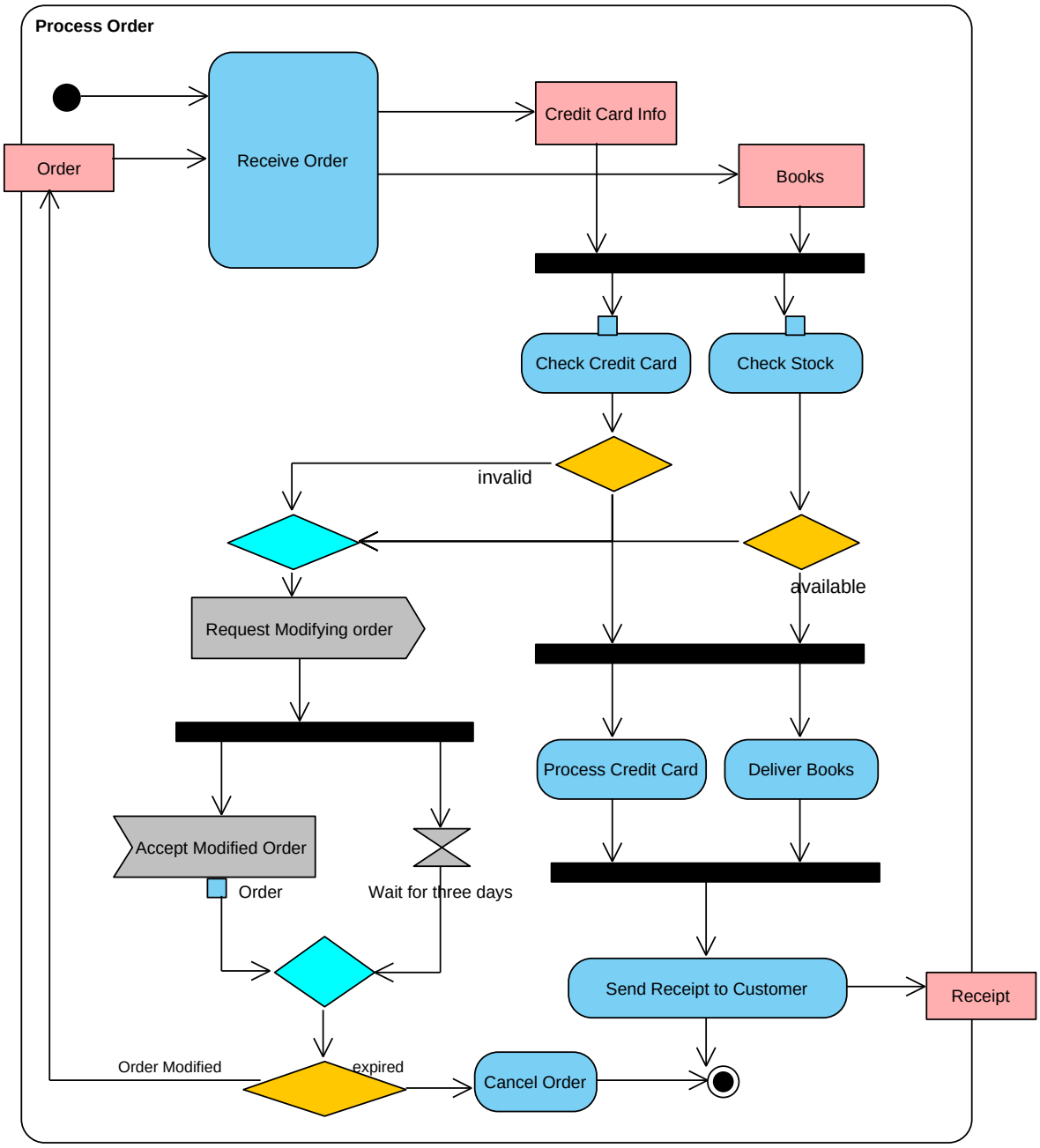


Figure 2.11: Example business process model for processing orders

like other behaviours, are parameterized), stored in data nodes, waiting to be consumed by the actions connected with these data nodes.

Directed edges represent control flows and data flows, which determines when actions start to execute. Control and data move one-way along the edges, and the following action can begin to be executed only when its preceding action release the control and all its required data become available from either preceding actions or parameters.

To summarize, the action model, activity model, and other behaviours like state machines, as well as type structures defined primarily in class diagrams, form a Turing-

Complete UML profile for computations.

2.1.3.2 Action Languages

It is imperative that a usable language involves two aspects: syntax and semantics, however, the concrete syntax of actions has not been defined though its abstract syntax and semantics are standardized in the UML specification [79]. Consequently, action languages [107, 87, 99] are needed to provide such concrete syntax. On the one hand, the semantics of action languages is required to conform to UPAS. On the other hand, action languages are required to be textual in that textual languages usually have higher coding efficiency than graphical ones, especially in the circumstances that xUML is expected to be a programming language.

A primitive construct in an action language may be a mapping to one action, or several for convenience. For example, creating an object may involve initializing attributes or creating other objects for compulsory associations, but the CreateObjectAction simply creates an object as required without any other effects, and further actions are required to initialize attributes and create objects for compulsory associations. Hence, an action language could define one object-creating construct as a shorthand for several actions.

Action languages also incorporate the constructs which correspond to the flow control mechanism provided in the activity model, such as decision making, path merge, forking and joining of concurrency. Usually, loop constructs like 'for' and 'while' are also provided for ease of coding [107]. More importantly, action languages rely on the type system of UML or xUML as a part of their type declarations.

A range of action languages have already been in use, such as Action Specification Language (ASL) [107], the BridgePoint Action Language [87], the Kabira Action Semantics (Kabira AS) [47], and the widely-used SDL [99] in telecommunication industry, Shlaer-Mellor's SMALL [59], TALL [59] and JAL [27]. These action languages vary in the fashion of syntax and semantics, for example, JAL is akin to Java in syntax, and TALL is a functional language. It is argued in [41] that the syntax of action language should be aligned with Object Constraint Language (OCL) because OCL has been standardized as a part of the UML specification.

2.1.3.3 AL vs OCL

Object Constraint Language(OCL) [76] is a formal declarative language standardized as a significant part of UML and mainly used to specify the constraints of a UML model (including the metamodel for UML itself). The instances of this model which conform to all the specified constraints are referred to as well-formed, otherwise as ill-formed.

OCL is mainly used as follows: 1) as a model query language to know about, for example, what class an object is, or which objects a class has; 2) to specify a condition for a type that all its instances must observe, namely a invariant of the type; 3) to specify pre-, post- and guard condition for operations or behaviours; 4) to specify derivation rules among model elements for making decisions or enforcing some relationships, for instance, to specify business rules; 5) to access structural features of an object, such as reading an attribute and association navigation.

There is no doubt that the emergence of OCL brings precision and rigour to UML. The static semantics of UML itself can be rigorously defined by OCL constraints. In addition, the preciseness of UML models can be enhanced by OCL because some UML constructs, such as the bodies of query operations, pre- and post-conditions and initial values of attributes, can be specified in OCL rather than informal natural languages.

However, OCL cannot replace action languages, even though a large part of OCL has semantic correspondence in ALs; for example, they both can read values of attributes and link ends, and have decision-making and looping constructs. OCL, as a pure specification language, has no side effect on system state like writing attributes of objects, creating/destroying objects and links, but ALs do. ALs usually further incorporate a reflection mechanism as OCL does. In addition, the type system of OCL is essentially equivalent to that of ALs because they both rely mainly on class diagrams for type declaration. For these reasons, ALs can replace OCL, at least in the sense of functionality, but not vice versa.

2.2 Formalizing UML

A specification of a computer language is required in order to achieve a common understanding of the language among the language designers, implementers and programmer. Generally, it involves two aspects: the specification of syntax and the specification of

semantics.

2.2.1 Abstract Syntax of UML

An abstract syntax is a model of the internal representation of programs in a computer language, and mainly concerned with the compositional structure of the phrases of a program. Instances of abstract syntaxes are produced as the major result of micro-syntax analysis in compile-time, are tree-structured and as such referred to as abstract syntax trees. A conventional and popular way to describe abstract syntaxes as context-free grammars is using notations akin to BNF or its variations such as EBNF; instead, OMG specifies the abstract syntax in the four-layer metamodeling framework [78], in which information is hierarchically divided into four layers, each representing a different level of abstraction. See Figure 2.12: from bottom to top, the four layers are the instance layer, modeling layer, metamodeling layer and meta-metamodeling layer, and the models in the layers apart from the instance (M0) layer are respectively called models (M1), meta-models (M2) and meta-metamodels (M3). Meta-metamodels describe the metamodels, in other words, the latter are instances of the former. This also applies to metamodels and models, plus model and instances. Interestingly, the meta-metamodel describes itself.

The descriptions of abstract syntax begin with defining a core reflecting the object-oriented modeling concepts: for example, objects are classifiable, have attributes and operations, have relationships with other objects, and can inherit features from ancestors. This core forms a fundamental part of UML and is specified in the UML infrastructure. Then this UML core can be used to describe the complex UML elements such as the state machine, the activity model and the action model. Therefore, UML can be thought as being defined by itself (by its own core). Even further, the UML core is imported and merged with MOF as its essential part to describe other metamodels such as CWM (Common Ware Model) [75].

Now we give an example to illustrate how the abstract syntax of UML is defined. See Figure 2.12. In the metamodel (M2) layer, there is a fraction of the UML metamodel, where the elements including Class, Association and Property are the instance of MOFClass which belongs to Meta-metamodel (M3) layer. In model (M1) layer, there is a model defined by the user whose syntax conforms to the UML metamodel, and the

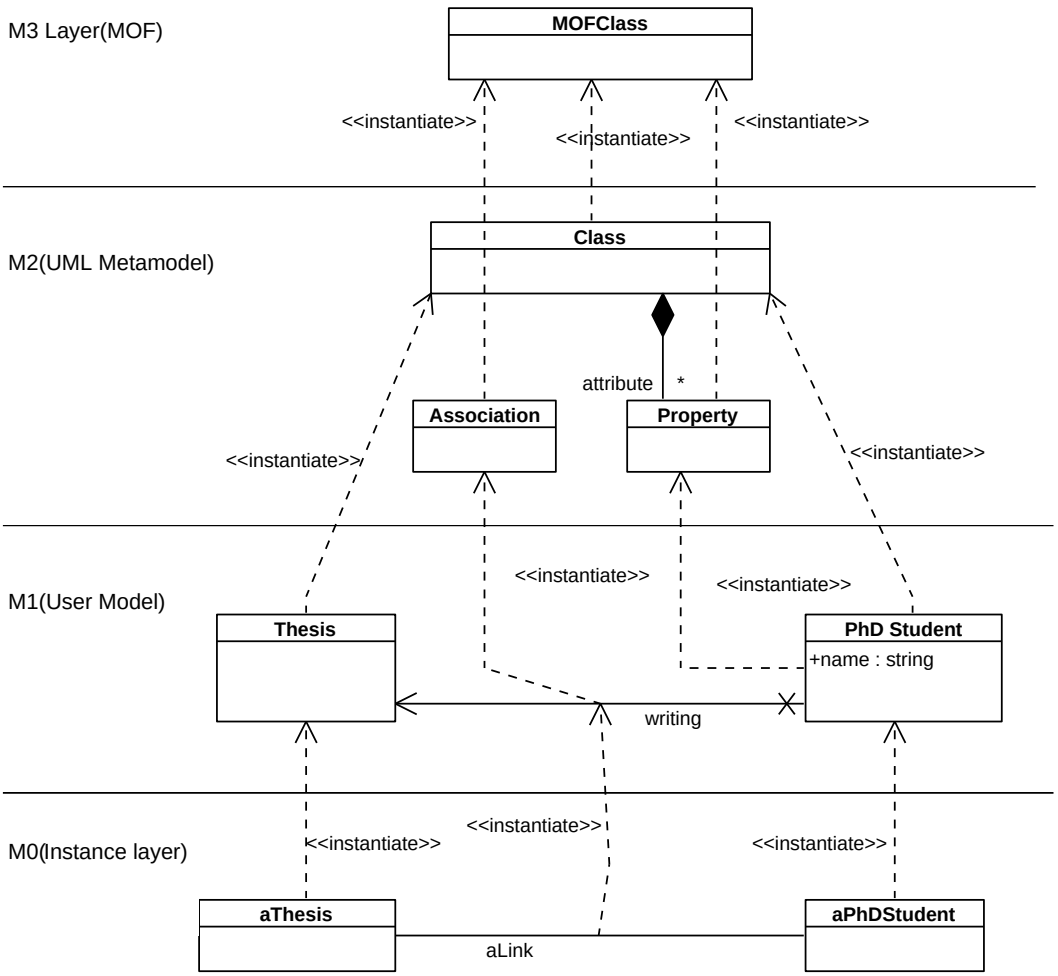


Figure 2.12: Illustration of the four-layer metamodeling architecture

elements in this layer are the instances of elements in the UML metamodel. The elements in the lowest layer (M0) are the runtime instances or some other real entities of a system. By this means, the abstract syntax of UML is specified in the UML metamodel using MOF, and a user model is checked against the UML metamodel for syntactic validity.

2.2.2 Contextual Constraints of UML

The syntactic validity of programs of a language depends on the compliance not only with the context-free grammar of the language but also with context-sensitive constraints. The latter, such as type rules, capture the context-sensitive aspect of the language, and are frequently called static semantics as they can be predicated and checked at compile time. For instance, in Ada [35], the parser would accept the statement which assigns an int value to a variable that is boolean-typed, whereas the type checker would reject it as a result of not meeting the contextual constraint that a variable can be only assigned with a value of the same type.

As mentioned, the UML metamodel has specified the context-free abstract syntax of UML in the four-layer metamodeling framework, while its context-sensitive constraints, which are referred to as well-formedness rules [78] are specified by a set of invariants [76] largely in OCL expressions accompanied by informal explanations. For instance, to prevent cyclic generalization, an OCL expression is specified in the context of Classifier (in M2 layer) like '*not self.allParents( ) –>includes(self)*', which means the parents of a class (in M1 layer) cannot include itself.

2.2.3 Semantics of UML (Related Work)

The run-time or dynamic semantics of UML is currently standardized in English in the UML Specification [78, 79]. Even though it is written with extreme care, it cannot be guaranteed to be entirely free of loopholes, contradictions, vagueness, and wording that doesn't express the writer's original intent. For example, it is not clear in UML whether a passive object (whose behaviours are invoked by other objects by CallOperationAction) can react to signal events. Since reaction to signals requires that the receiving object must reside in an active state of awaiting occurrence of a particular signal event, the receiving object must be an active one rather than passive. However,

the UML metamodel provides all objects, whether active or passive, with a mechanism (embodied in Reception: a meta-class in UML metamodel) for declaring what signals objects may react to and which behaviours should be invoked. This contradiction has been mentioned in [54] and the interpretation of this point by researchers differs from one to another.

The informality of the semantics of UML has aroused great concerns of researchers and UML practitioners, and the need for a precise semantics was discussed in [29, 32]. The benefits of formal semantics of UML can be summarized as follows.

- It allows subtle errors in the current and future versions of the UML standard to be detected, and suggestions for improvements to be made.
- It is critical in achieving common understanding among UML implementers, UML modelers and UML designers.
- It enables tool vendors to develop tools that offer more powerful and effective testing, analysis, and model transformation functionality and better support the exchange of modeling artefacts between different tools.
- It makes possible automatic code generation, model simulation, and validation and verification of models.

Numerous attempts have been conducted to provide formal semantics for UML. Their approaches can be categorized into the following six groups.

1. **Pure Set theory approach**. M. Richters and M. Gogolla [90] utilized set theory to formalize class diagrams and the operations available in OCL. In this approach, types are represented by sets, and the operations on types are defined by the mathematical functions over sets. This approach can be regarded as a denotational approach because UML elements are mapped to the mathematical notations, but usually merely concerned with the semantics of the static part of UML such as the class diagram and the OCL expressions.
2. **Meta-modeling approach**. This approach was only used to describe the static semantics of UML in a small subset of UML, which is parallel to OMG in specifying UML in a four-layer metamodeling framework. The pUML group originated this

approach and used it in the semantics of UML, where, essentially, an algebraic specification is used to describe legal snapshots of the system [29, 25, 11].

3. **Translation approach**. This approach is characterized by defining the translation from UML to traditional specification languages, such as Z, B, Object-Z, CASL, Petri-net. The semantics of UML is then represented by the target specification languages whose semantics have been formalized. W. McUmber and H. Cheng [58] proposed a general framework for this approach and argued the translation should be homomorphic mappings between metamodels so that the structural relationships between the elements in two different metamodels can be preserved. The translation of UML to the input languages of tools, such as theorem provers, code generators and model simulators, are also categorized into this approach. For example, M. Kyas and H. Fecher translated OCL combined with class diagrams into the input language of PVS [52].

   G. Reggio et. al [89] employed an extension of the algebraic language CASL to describe the semantics of individual diagrams, class diagrams and state machines, and then the semantics of the individual diagrams are integrated to give the overall semantics of UML.

   ASM (Abstract State Machine) is also popular in defining UML semantics [18, 31, 74]. E. Börger et al. [19] provide a relatively complete dynamic semantics of UML in terms of ASM which has been enriched by some new constructs specially for the characteristics of UML state machines. The model covers inter-object communication (event handling mechanism), the run-to-complete process of intra-object state transitions as well as flow control and data control, which together form the major dynamics of UML.

   Petri-nets are frequently used to model the semantics of activity diagram due to the similarities in semantics [97, 34, 96]. In this approach, activity nodes are mapped to places, and edges to transitions. For example, Harald Störrle and Jan Hendrik Hausmann [97] explained the formal semantics of activity diagrams by defining a mapping of the basic elements of activity diagrams to procedural Petri-nets and then investigating how strong the alignment of UML's activity diagrams to Petri-net is.

4. **Operational semantics approach**. This approach usually starts with defining the abstract syntax of a part of UML (State machines or Activity diagrams) in terms of mathematical concepts (largely in set theory) in a high level of abstraction. Then object configuration, state configuration and environments are also defined as mathematical data structures (such as tuple, set, queue, map etc), and also some auxiliary operations are defined as functions over this data structures. Finally, execution steps are defined using transition rules in first-order predicate logic. For example, W. Damm et al. [26] define a real-time-system-oriented subset krtUML of UML which is adequate to represent the behaviour-modeling mechanism of UML. Then, the dynamic semantics of krtUML is provided using a symbolic transition systems, where the state-space of the transition system is given by the valuation of a set of typed system variables, and initial states and the transition relations are defined by some first-order logic predicates.

5. **Virtual machine approach**. V. Vitolins and A. Kalnins's [103] semantics of UML activity diagrams is a representative of the virtual machine approach, where a virtual machine is designed by means of metamodel and takes a specific activity diagram as input. The execution steps are defined by a mix of pseudo-code and OCL expressions as pre- and post-conditions.

6. **Combined approach**. One may combine the approaches mentioned above to formalize the semantics of UML. An example of combined approach can be found in [55], where the author used set theory and first-order predicate logic to model the abstract syntax and static semantics of the sequence diagrams. As for the operational semantics of message execution, predicate rules are used for transitions of system state.

Other approaches to UML semantics also exist. For example, M. Enciso [91] uses temporal logic to represent dynamic behaviours of UML state machines; S. Kuske [51] described a UML state machine based on the theory of graph transformation; D. Harel and S. Maoz [39] proposes a modal semantics, Modal Sequence Diagrams (MSD), to address the definitions of *assert* and *negate* in sequence diagrams.

2.2.4 Limitations of Previous Attempts

The approaches mentioned surely do good for the formalization of UML semantics. However, they share the following drawbacks.

- Most approaches are focused on one or two diagrams. This is partially because one approach is very suitable for one part of UML but can not specify the semantics of other parts. Even if it can, it may be awkward. For example, Petri-nets are confined to modeling the activity diagrams of UML; Pure set theory is limited to specifying the semantics of static part of UML such as OCL. However, it is hoped that the dynamic semantics of UML may be specified in a universal approach.

- As mentioned, the formal semantics of UML must cater for diverse audience of language implementers, language designers and language users. This requires that the formal semantics of UML, on the one hand, should be sufficiently formal, and on the other hand should be readable by different kinds of users. Most current attempts have definitely added more or less formality to UML, however, they are usually based on abstract mathematical concepts, which decreases the readability of the semantics. This is further compromised by poor organization and lack of modularity in the descriptions. As a result, the current semantics of UML has practical problems.

- The present semantics of UML is rarely based on a mature semantics-describing framework. Indeed, some approaches themselves, relatively speaking, are not stable and are evolving dramatically, and some either lack theoretical foundation or are not well-proven in practical use. That is to say, the current UML semantics is founded on weak bases. However, a mature semantics-describing framework has a solid ground both in theory and in practice and usually has been applied to a range of languages. In addition to that, a mature basis means a significant community of users and has been taught to students. In conclusion, it is preferable that the semantics of UML is based on a mature semantics-describing framework.

In addition, D. Harel and B. Rumpe [40] also argued that there are misconceptions surrounding the the existing formal semantics of UML and pointed out what semantics actually is. These limitations of the previous attempts motivate us to describe the semantics of UML in a mature framework, Mosses' Action Semantics [23].

2.3 Summary

UML, a general-purpose visual modeling language, is intended to be used in all stages of the system development, and captures the static and dynamic aspects of the system using different views (or diagrams). The xUML, a rigorous subset of UML, is the outcome of the efforts of moving UML to a programming language with high level of abstraction. UPAS is incorporated into UML to define a set of actions which are the basic building blocks of behaviours, and action languages are created to provide syntax to UPAS.

The run-time semantics of UML is specified in plain natural languages, leading to inevitable loopholes, ambiguities, inconsistency and poor wording. Despite various efforts having been made to provide formal semantics to UML, none of them bases the semantics of UML on a mature semantics-describing framework, hence lacking readability, solid ground in theory and practice, and tool supports. This motivates us to describe the semantics of UML using Mosses' Action Semantics.

Chapter 3

Introduction to Action Semantics

Programming languages are consciously designed by computer scientists and targeted to a relatively small population of users, mostly programmers. Unlike natural languages, ambiguities in a programming language are not tolerable because its audience is computers rather than the highly-intelligent human beings. Thus, researchers agree on the necessity of formalizing the semantics of programming languages because a formal semantics can act as [92]:

- A formal standard for language implementation. The formal standard of semantics can help ensure that the language is implemented exactly the same on all machines, by different implementers.

- A precise and complete user documentation. A user of the language can refer to this formal semantics for exactly understanding some subtle parts of the language. The quality of a software product cannot be assured if its developers fail to understand the implementation language precisely.

- A tool for design and analysis. On the one hand, with the formal semantics definition, the designer of the language can study the pragmatics of the language in the early stages of language development even though there are few supporting tools and no user feedback. On the other hand, the designer can use the formal semantics to make reasoning to find out loopholes, inconsistencies and incompletenesses within the language specification.

- Input to a compiler generator. A compiler generator takes the formal semantics as input and produces an assured implementation of the language. Automatic

generation reduces the time of prototyping and sets the programmers free from tedious and error-prone coding.

For decades, people have explored various approaches to formalizing the semantics of a programming language, such as *denotational semantics* [98], *operational semantics* [73], and *axiomatic semantics* [73], and *Action Semantics* (AS) [65]. In this chapter, we will give a small language at the very beginning as our running example to introduce these approaches (except *axiomatic semantics* because it is not related to our research). Emphasis is placed on AS because it is the semantics-describing framework adopted in this research. Then, a comparison is made between UPAS (introduced in Chapter 2) and Mosses's Action Semantics, to clear up the confusion between them. Finally, we sum up this chapter by highlighting the major features of the introduced semantics and the justifications of why we choose action semantics as the vehicle to specify UML.

3.1 Introduction to Operational Semantics

In an operational semantics one describes the semantics of a language by specifying a *transition system* [86], defined as a tuple $< \Gamma, \longrightarrow >$. Γ is a set of elements: γ, called *configuration*, and $\longrightarrow$ is a binary relation of $\Gamma \times \Gamma$, called the transition relation. A particular $\gamma \longrightarrow \gamma\prime$ is an element of $\longrightarrow$ and means a transition from the configuration γ to $\gamma\prime$. Furthermore, the terminal configuration T can be specified in addition to the *transition system*$< \Gamma, \longrightarrow >$, and then a terminal transition system $< \Gamma, \longrightarrow, T >$ is obtained, where T is a set of terminal configurations and if a transition results in a terminal configuration, then the system would halt [86]. A terminal transition system is useful in specifying a finite automaton with a set of finite final states.

Configuration indicates what parts of the transition system (viewed as an abstract machine) may change during the execution of programs and can be thought of as the system state before or after a transition. A configuration normally contains two aspects: a *control part*, and one or more *data parts* [86]. The control part refers to the instructions or the part of a program that remain to be executed, and the data parts vary depending on the language described. For example, when one describes a simple declarative language consisting of binding but no commands, the data parts of configuration are just the binding environment. If the language is imperative, then the store

is also a part of the configuration. However, in order to make the operational semantics syntax-directed, the control part of the configuration is often ignored especially in formalizing high-level languages.

To illustrate operational semantics, we coin a simple imperative language, called IMP, as the running example. Its abstract syntax is shown in Syntax 3.1. IMP is a typ-

Syntax 3.1 Context-free grammar for IMP

S	::=	skip | $I := E$ | if E then S else S | while E do S | S; S
D	::=	$I :: T$
E	::=	N | I | $E\ Op\ E$
Op	::=	$+$ | - | $*$ | $\neq$ | $\vee$ | $\wedge$
N	::=	$(0\|1)^+$
T	::=	binary | boolean

ical imperative programming language, including statements (S), a variable declaration (D) and expressions (E). For simplicity, only two types are considered: binary number and boolean, and they are intuitively corresponding to the sets $\mathbb{N}$ ($\mathbb{N}$ = natural numbers $\sqcup$ 0) and $\mathbb{B}$ (truth values). An identifier (I) stands for a variable that corresponds to a single location in the store. The data that can be held in the storage are called *storables*, and those that can be bound to tokens are called *bindables*. For convenience, we define the set $VALUE$ as the union of $\mathbb{B}$ and $\mathbb{N}$, namely $VALUE = \mathbb{B} \sqcup \mathbb{N}$. In the case of IMP, values are storables, and only the locations of the storage are bindables.

The execution of an IMP program involves modification of the binding environment (caused by variable declarations) and the data store (caused by the execution of statements). So the configuration for IMP is defined as follows:

Definition 1 *Configure γ for IMP is a tuple $\langle ENV, STORE \rangle$*

- *where ENV is a set of functions(env: $I \longrightarrow LOCATION$), with the following two auxiliary operations on ENV.*
  - *bind: $ENV \times I \times LOCATION \longrightarrow ENV$, which adds a binding to the current environment and produce a new environment.*
  - *find: $ENV \times I \longrightarrow LOCATION$, which returns a location bound to the given identifier in the current environment.*
- *where STORE is a set of functions (store: $LOCATION \longrightarrow VALUE$). The LOCATION is a set whose elements are used to identify the cells in the storage; its*

VALUE can be a unique integer or a unique string, which depends on the situation and is of no significance in describing semantics. Three auxiliary operations are available over STORE:

- *allocate: STORE* $\longrightarrow$ *STORE* $\times$ *LOCATION, which allocates a cell in the current store resulting in a new store and return the location of the newly allocated cell.*
- *update :* $STORE \times LOCATION \times VALUE \longrightarrow STORE$*, which modifies a location in the store to return a new one.*
- *fetch :* $STORE \times LOCATION \longrightarrow VALUE$*, which obtains the value of a given location from a store.*

All the above auxiliary functions are only informally explained merely for saving space. The mentioned sets $\mathbb{B}$ and $\mathbb{N}$, and their operations, are intuitive and can be specified with ease, for example, by algebraic specification, so we do not define them here.

The semantics of declaration of a variable is given by the function

$$elaborate : I \times STORE \times ENV \rightarrow ENV$$

which is defined as as

$$elaborate[\![i : I \ :: \ Type]\!](o, \ v) = bind(v, \ second(allocate(o))).$$

Note that we use '$[\![\ ]\!]$' to enclose a syntactic argument, a terminal or a non-terminal, and the *second()* is assumed to be a predefined operation on binary tuples which retrieves the second element of a binary tuple. Furthermore, meta-variables are variables that range over non-terminals and are declared in the form of '$n : Nonterminal$' where 'n' is a meta-variable. In addition, the variables o stands for an element of $STORE$, and v for an element of ENV.

Then we specify the semantics of evaluation of expressions, whose effect is simply returning values to be consumed immediately by the enclosing statements, without modifying the configuration elements: the storage and the binding environment. The semantics of evaluating expressions is represented by the function $ev : STORE \times ENV \times E \rightarrow VALUE$, which is then defined in a syntax-directed manner (See Syntax 3.1).

OSD 3.1 Rules for defining the evaluation of IMP expressions.

$$o \in STORE, \quad v \in ENV, \quad i : I, \quad e_1 : E, \quad e_2 : E$$

$$ev[\![()]\!] = 0, \quad ev[\![0]\!] = 0, \quad ev[\![1]\!] = 1$$

$$ev[\![N]\!] = ev[\![(0 \mid 1)^*(0 \mid 1)]\!] = 2 \times ev[\![(0 \mid 1)^*]\!] + ev[\![(0 \mid 1)]\!]$$

$$ev[\![\ i : I]\!](o,\ v) = fetch(o, find(v,\ i))$$

$$\begin{array}{ll} ev[\![e_1\ op\ e_2]\!](o,\ v) = ev[\![e_1]\!](o,\ v) + ev[\![e_2]\!](o,\ v) & \text{if } op = + \\ ev[\![e_1\ op\ e_2]\!](o,\ v) = ev[\![e_1]\!](o,\ v) - ev[\![e_2]\!](o,\ v) & \text{if } op = - \\ \ldots & \end{array}$$

From OSD 3.1, it can be known that, despite a query into the storage and environment, the evaluation of expressions has effects neither on the storage nor on the binding environment. Unlike the evaluation of expressions, the execution of statements may have effects on the storage. The semantics of executing statements are represented using function $ex : S \times STORE \times ENV \longrightarrow STORE$, which is then defined by a set of transition rules in OSD 3.2.

It can be noticed that the transition rules in OSD 3.2 are only concerned with the final effect of phrases. The operational semantics of this kind is called *natural semantics* or *big-step semantics* [73]. Another kind of operational semantics, called *structural operational semantics* [86], are used to describe the *individual* steps of computations. For instance, the structural operational semantics of the statement '$S ::= S;\ S$' is like:

$$\frac{ex[\![s1]\!](o,\ v) \rightarrow\ o'}{ex[\![s_1 :\ S;\ s_2 :\ S]\!](o,\ v) \rightarrow\ ex[\![s_2]\!](o',\ v)}$$

To conclude, in an operational semantics, one is concerned both with effects or results of the computation (embodied in the change of configuration) and with how to execute programs (embodied in transition rules).

3.2 Introduction to Denotational Semantics

In *denotational semantics* [95, 98], one provides the meaning of a programming language in terms of mathematical objects, such as integers, truth values, tuples, and functions,

OSD 3.2 Rules for defining the execution of IMP statements.

$$o,\ o',\ o'' \in STORE,\quad v \in ENV,\quad a \in VALUE$$
$$i:\ I,\quad e:\ E,\quad s_1:\ S,\quad s_2:\ S$$

$$ex[\![skip]\!](o,\ v) \rightarrow\ o$$

$$\frac{ev[\![e]\!](o,\ v) \rightarrow\ a}{ex[\![i\ :=\ e]\!](o,\ v) \rightarrow\ update(o,\ find(v,\ i),\ a)}$$

$$\frac{ex[\![s_1]\!](o,\ v) \rightarrow\ o'}{ex[\![if\ e\ then\ s_1\ else\ s_2]\!](o,\ v) \rightarrow\ o'} \qquad \text{if } ev[\![e]\!](o,\ v) = TT$$

$$\frac{ex[\![s_2]\!](o,\ v) \rightarrow\ o'}{ex[\![if\ e\ then\ s_1\ else\ s_2]\!] \rightarrow\ o'} \qquad \text{if } ev[\![e]\!](o,\ v) = FF$$

$$\frac{ex[\![s]\!](o,\ v) \rightarrow\ o',\ ex[\![while\ e\ do\ s]\!](o',\ v) \rightarrow\ o''}{ex[\![while\ e\ do\ s]\!](o,\ v) \rightarrow\ o''} \qquad \text{if } ev[\![e]\!](o,\ v) = TT$$

$$ex[\![while\ e\ do\ s]\!](o,\ v) \rightarrow\ o \qquad \text{if } ev[\![e]\!](o,\ v) = FF$$

$$\frac{ex[\![s_1]\!](o,\ v) \rightarrow\ o',\ ex[\![s_2]\!](o',\ v) \rightarrow\ o''}{ex[\![s_1;\ s_2]\!](o,\ v) \rightarrow\ o''}$$

so denotational semantics was originally called *mathematical semantics*. These mathematical objects construct the semantic world for the language, meanwhile the syntactic world of the language is expressed in a variant of BNF or EBNF. Two major tasks in supplying denotational semantics for a language are, first, defining the semantic world, and second, specifying semantic functions which connect the two worlds through mapping the objects in the syntactic world to those in the semantic world.

Dana Scott's *Domain theory* [9] answers the question of how to create the semantic world by arguing that the semantic world is made up from *domains*. Semantic domains can be roughly regarded as the set-theoretical *sets* and are categorized as primitive domains and compound domains. The primitive domains are such like: **N** (the domain of natural numbers), **B** (the domain of truth values), **I** (the domain of integers), **R** (the domain of rational numbers), etc. It is straightforward that primitive domains are the semantic shadows of the primitive types in programming languages.

Compound domains are needed to model those composite data structures and complicated computations. Analogous to set construction of set theory, compound domains

are constructed from the simpler domains by the constructors: Cartesian product ($\times$), Union ($+$) and Function ($\longrightarrow$).

In *domain theory*, all domains may be extended with one additional element ($\perp$), called *undefinedness* or *bottom*. Such extended domains are referred to as *lifted domains* [92]. The introduction of undefinedness is owing to the fact that execution of programs is subject to *non-terminations* (infinite loops) and *abnormal terminations*. Thus the undefinedness should be present to represent such situations.

So far, we know that the primitive domains and the methods of constructing compound domains. Now we proceed to illustrate how to use the domains to denote the semantics of the toy imperative language IMP.

Likewise, mathematical objects are needed to denote the storage and the binding environment of IMP. For them, we reuse their definitions in Section 3.1. We also reuse the operational semantics of expressions and variable declarations. The reuse is safe because they are essentially *mathematically* defined based on set theory albeit in a different style. As such, we only need to specify the semantics of statements in IMP using denotational semantics.

The semantic function of executing statements is $ex : S \longrightarrow STORE \times ENV \longrightarrow STORE$. That is, the execution of a statement will use the given environment and store and then produce a new store. Its signature is similar to that in the operational semantics. Differently, in denotational semantics, it is semantic equations, rather than the first-order predicate logic rules, that are used to express semantic functions. See DSD 3.1 for the denotational semantics of IMP statements. Note that in the description we use an auxiliary function '*cond(_, _, _)*'. This function has three parameters and will select the second parameter if the first parameter is evaluated true, otherwise it will select the third parameter.

In DSD 3.1, the defined semantic equations map syntactic phrases into mathematical objects (functions), and the semantics of almost every composite syntactic phrase is composed from those of its immediate sub-phrases. Such compositionality is required by denotational semantics, which accounts for denotational semantics being called compositional semantics. However, the semantic function for the construct *while-statement* is recursively-defined; i.e., it is defined partially on itself rather than purely on the semantics of its immediate sub-phrases, which breaks the principle of compositionality. So, rigorously speaking, it is by no means fully defined.

DSD 3.1 Denotational semantics definition of IMP statements

Statement

$o \in \mathsf{STORE}$, $v \in \mathsf{ENV}$, i: I, e: E, s_1, s_2: S.

(1) $ex[\![\ \mathsf{skip}\ ]\!](o, v) = o.$

(2) $ex[\![i := e\ ]\!](o, v) = update(o, find(e, i), ev[\![\mathsf{e}]\!](o, v)).$

(3) $ex[\![\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2]\!](o, v) = cond(ev[\![e]\!](o, v), ex[\![s_1]\!](o, v), ex[\![s_2]\!](o, v)).$

(4) $ex[\![\mathsf{while}\ e\ \mathsf{do}\ s]\!](o, v) = cond(ev[\![\mathsf{e}]\!](o, v), ex[\![\mathsf{while}\ e\ \mathsf{do}\ s]\!](ex[\![s]\!](o, v), v), o).$

Domain theory provides a treatment of such recursively defined semantic functions so as to get the non-recursive definition. It is generally shown as follows. The following equation can be obtained from Equation (4) of DSD 3.1:

$$ex[\![\mathsf{while}\ e\ \mathsf{do}\ s]\!](v) = cond(ev[\![\mathsf{e}]\!](v), ex[\![\mathsf{while}\ e\ \mathsf{do}\ s]\!](ex[\![s]\!](v), v), id),$$

where id is the identitity function defined as

$$id : STORE \longrightarrow STORE$$

which satisfies $\forall o \in STORE.id(o) = o$. Then, '$ex[\![\text{while } e \text{ do } s]\!](v)$' is a fixed point of the function F defined as

$$F(g) = cond(ev[\![e]\!](v),\ g(ex[\![s]\!]),\ id),$$

where the function F is not recursively defined.

However, the ultimate goal is not to get the function F but to get one of its fixed point which genuinely denotes the semantics of the while-statement phrase. The further problem is that F may have no, or more than one fixed points. Thanks to domain theory, it has proved that at most one fixed point can be established as the desired semantic function. This fixed point is the least-defined one among all fixed points of F [73]. Domain theory has provided a solid solution as to how to get this least fixed point. Readers are referred to [9] for more details.

We simply use FIX, which is defined rigorously in domain theory, to denote the function that is able to produce the least-defined fixed point of the given function. So the semantic equation of the while-statement can be safely specified as:

$$ex[\![\textsf{while } e \textsf{ do } s]\!](o,\, v) = FIX(F)(o), \textsf{ where } F(g) = cond(\textsf{ev}[\![e]\!](v),\, g(\textsf{ex}[\![s]\!]),\, id).$$

Such description of IMP does not consider errors such as infinite loops and abnormal terminations. In denotational semantics, an error occurring at a sub-phrase is propagated to its immediate containing phrase, until the root of the whole program. This process is modelled by the confluence of the aforementioned *undefinedness* and the mechanism of strict functions which are characterized by

$$\text{if } f \text{ is a strict function in the domain } A \to\ B, \text{then } f(\perp) = \perp.$$

For example, in some languages, if an erroneous assignment like '$x := 3/0$' is possible, then we can define the execution of assignment as

$$ex(i : I := e : E) = \begin{cases} \ldots & \text{if } ev(e) \neq \perp \\ \perp & \text{if } ev(e) = \perp. \end{cases}$$

$$ev(e1 : E / e2 : E) = \begin{cases} ev(n1)/ev(n2) & \text{if } ev(n2) \neq 0 \\ \perp & \text{if } ev(n2) = 0 \end{cases}$$

According to this, then $ev(3/0) = \perp$, thus $ex(x := 3/0) = \perp$, and so on. The error is propagated to the top-most semantic level by this means.

To conclude, *denotational semantics* are aimed to specify the meanings of programming languages *purely* on a mathematical basis. A denotational semantics assigns semantics to every phrase—every expression, every statement, every declaration, etc. and it is *compositional* because the semantics of a phrase is composed by those of its immediate subordinates. So, the structure of a denotational semantics is parallel to the language's syntactic structure. Distinguished from operational semantics, *denotational semantics* is only concerned with the final computational results but little with computational steps.

3.3 Introduction to Action Semantics

Unlike informal semantics written in natural languages, the formal techniques, particularly the *operational semantics* and *denotational semantics* introduced in the earlier

sections, can be employed to provide accurate and unambiguous semantics for programming languages, which then serve as the basis for proving properties of programs as well as for fast language implementation.

Even though the formal techniques are strongly promoted, most programmers still prefer informal semantics to formal ones to understand programming languages, in that they find formal semantics notationally dense, cryptic and unintelligible. Furthermore, for language designers, a formal specification is difficult to create correctly, to modify, and to extend. Especially when the concerned language is large-scale, its formal definition becomes overwhelming both to the language designer and the language user, and thus remain mostly impractical in reality, particularly in industrial circumstances.

Action Semantics (AS), which addresses these criticisms of formal methods, was developed in the second half of the 1980's by Peter Mosses with the collaboration of David Watt [67]. Its major ambition is to make formal semantics easy to create and read, and then make formal techniques more applicable [65].

Currently, there are two versions of action notation: the original version (referred to as AN-1) and the newer version (known as AN-2). AN-1 is described in the Action Semantics book [65], and formally defined by providing a *structural operational semantics* for its kernel and some rules that allow the full AN-1 to be reduced to its kernel. AN-2 is the revised design of AN-1 and has been proposed at the AS 2000 workshop [53]. AN-1 is stable, so it is adopted in this thesis.

In this section, we introduce AS in a considerable detail. In order to make the introduction not overwhelming, we do not consider the concurrency mechanism in the action semantics. This introduction to AS is primarily based on [65], [106] and [95].

3.3.1 Action Machine

The framework of *Action Semantics* [65] has specified an abstract machine (*Action Machine* or AM) which can respond to a set of 'instructions', namely *actions*, the execution semantics of which are formally defined using *structural operational semantics*. So we consider that it is preferable to introduce action semantics from the perspective of *operational semantics*.

Generally, the *Action Machine* (AM) is such that it is in a particular state (or configuration) at any moment of runtime and its state evolves over time. The state or

configuration of AM is defined as a tuple:

$$state \ = \ < acting, \ store >$$

where

- the *acting* component is analogous to the remaining action to be performed in the AM. It is associated with two kinds of data: *transient data*, which is essentially a variably-sized tuple; *binding data*, which is map-structured and akin conceptually to the symbol table used in a language's compile time. So more clearly, the state can be written as

  $$state \ = \ < action, \ transient, \ binding, \ store > .$$

  The transient and binding data may be consumed or dynamically changed by the action being performed. The *acting* is nestable. That is, the structure of acting can be described in a BNF form as

  $$acting ::= action \ transients \ bindings \mid acting \ transients \ bindings.$$

  The nestability of the *acting* provides convenience for defining data flows using structural operational semantics.

- the *store* component is a memory abstraction, and it is conceptually capable of storing data unless the data are explicitly changed or destroyed. In addition, the store can act as a medium of communications amongst actions which are not structurally related. Unlike a real memory, the size of this store can vary infinitely.

An action performed in AM usually causes a sequence of state transitions: changing the store and meanwhile making the acting component simpler and simpler. Step by step, the initial acting part is diminished to one of the following *terminated* actings (omitting data and bindings):

- *Completed*, which indicates the performance is successfully accomplished.

- *Escaped*, which indicates that the performance is terminated somewhere, leaving the remaining action unperformed. This is parallel to exception handling in some

programming languages.

- *Failed*, which indicates the performance has either diverged or terminated abnormally. Being diverged means the performance of the action falls into an infinite loop; abnormal termination is analogous to the phenomenon that a running program encounters a vital dynamic error.

AS provides a rich range of constructive actions so that AM is a Turing-Complete machine.

3.3.2 Sorts and Algebraic Specification

The performance of actions may use and produce data. AS, as a complete framework, provides a way to specify this data, namely using *unified algebraic specification* [65]. The unified algebraic specification is an unorthodox algebraic specification which shares major concepts with the other algebraic specifications such as Algebraic Specification Formalism (ASF) described in [14]. Take one point as example, in all algebraic specifications, *sorts* are the major concept, and a sort can be simply viewed as a set-theoretical *set* equipped with some operations.

However the unified algebra is different from the traditional ones in various ways. Particularly, in traditional algebras, sorts and the contained elements (individuals) are treated separately and differently. For instance, merely individuals but not sorts can be applied to operations. However, the unified algebra handles elements and sorts uniformly. The following are the distinguishing features of unified algebra [64]:

- An *individual* is also a sort (singleton sort); there is no distinction made between a singleton and its only element. Both of them are treated as values and thus they both can be the arguments of operations. For example, 'e: S' is used to define 'e' is an individual of the sort **S**.

- The carrier of an algebra is a distributive lattice, where all the elements are sorts including singletons. The sorts are partially ordered by sort inclusion (denoted by $\leq$ or $\geq$). Joins and meets are obtained respectively by sort union (denoted by |) and sort difference (denoted by &). Most significantly, the empty sort or vacuous sort is incorporated as the bottom element to represent undefined results,

represented notationally by **nothing** in AS. In unified algebra, '$S_1 \leq S_2$' means that S_1 is a subsort of S_2.

- Operations on elements are subsort-preserved. that is, for each operation *op*, It holds that if $s_1 \leq s_2$, then $op(s_1) \leq op(s_2)$.

- The axioms used to specify unified algebras are quite general: Horn clauses, involving equality, sort inclusion, and classification of elements into sorts.

AS has defined some data sorts for common use, which are called by us *built-in data sorts* including some primitive ones such as **truth-value**, **integer**, **cell**, **natural**, and some composite data sorts such as **list**, **tree**, **map**. The reader is referred to [65] for a complete unified algebraic specification of these data sorts as well as their operations. In addition to the built-in data sorts, users can define their own sorts (*user-defined data sorts*) on demand. For instance, one can define the data sorts **class** and **object** to denote respectively classes and objects in an OOP language. All built-in data sorts and user-defined ones are sub-sorts of **data**.

To illustrate major concepts of the unified algebraic specification, we excerpt from [65] the specification of a built-in data type—**map**, shown in ASD 3.1.

In ASD 3.1, '**introduces**' is a keyword. Its contents are a number of sort names and operation names that this specification is going to define. Among them, the operation names are distinguished from sort names: operation names have place holders denoted by '_' , whereas sort names have none. Hence, in this example, map and range are two sorts; the others are operations. (The operations with place holders are often referred to as *term constructors*, and the sort names can be regarded as a special kind of operation, called *constants*.)

'**needs**' is also a keyword. Its contents are usually a number of module names that this specification relies on. As is an advantage, unified algebraic specification enables modularization to make a large specification well-organized. This example depends on the module 'Tuple/Basics' which has been defined somewhere. 'map $\leq$ component' indicate the sort **map** is a sub-sort of **component** which is defined elsewhere.

Clause 1 and **Clause 2** are both sort equations. Clause (1) defines the sort **map** through the two constructors: **disjoint-union** _ , and **map of** _ **to** _ . Clause

ASD 3.1 Excerption of the **map** specification of AS

Map

introduces: map , range , map of _ to _ , empty-map , disjoint-union _ , mapped-set _ .

needs: **Tuple/Basics**. map $\leq$ component .

(1) map = disjoint-union(map of element to range)* .

(2) range = nonmap-range | map (*disjoint*) .

(3) map of _ to _ :: element, range $\rightarrow$ map (*total*, *injective*) .

(4) empty-map : map .

(5) disjoint-union _ :: map* $\rightarrow$ map (*partial*, *associative*, *commutative*, *unit is* empty-map)

(6) mapped-set _ :: map-set (*total*) .

(7) disjoint-union ( ) = empty-map ;
disjoint-union (map of e:element to r: range) = map of e to r ;
intersection (mapped-set m_1, mapped-set m_2) is empty-set = true $\Rightarrow$
disjoint-union (m_1:map, m_2:map): map;
intersection (mapped-set m_1, mapped-set m_2) is empty-set = false $\Rightarrow$
disjoint-union (m_1:map, m_2:map) = nothing .

(8) mapped-set empty-map = empty-set ;
mapped-set map of e: element to r: range = set of e ;
mapped-set disjoint-union(m_1: map, m_2: map) = disjoint-union(mapped-set m_1, mapped-set m_2) .

2 defines the sort **range** through the sort union (denoted by '|'). The key word *disjoint* indicates that the sort **nonmap-range** and the sort **map** are disjoint, that is, they share no common elements. Aside from sort union, sort intersection (denoted by '&') is also supported in unified algebraic specification.

Clause 3 defines the signature of the operation **map of _ to _**, which is a constructor for maps with a single entry. The properties of this operation are denoted by the keywords: *total*, *injective*. The meanings of some important keywords of this kind are highlighted as follows:

- *total*, indicates that the operation is a total function. That is, if the arguments of this operation are individuals of the specified sorts, then the result is also an individual. To understand this, recall that, in unified algebra, not only individuals but sorts are legal arguments; individuals exclude vacuous sorts (**nothing**). Moreover, a total operation is also a strict operation.
- *injective*, indicates that the operation is a one-to-one mapping function.
- *partial*, indicates that the operation is a partial function. I.e., when the arguments are *individuals* (excluding **nothing**), the result may be **nothing**.

- *strict*, indicates that the operation is strict. That is, when any required argument is **nothing**, the result is definitely **nothing**.

Clause 4 asserts that **empty-map** is an individual of the sort **map**.

Clause 5 specifies the signature of a constructor of **map**, which means a multi-entried map is a sequence of single-entry maps, prefixed by 'disjoint-union'. Sequences are treated in action semantics as variable-sized tuple. The symbol '*' is an operation of the sort **tuple** which constructs a sequence of the given element.

Clause 6 specifies the signature of an operation which returns the set of keys of the given map.

Clauses 7 is several Horn clauses to define the operation **disjoint-union** _, which unions two maps. However if there exists any common key in the two sorts, the result is **nothing** indicating that the operation fails.

Clauses 8 defines the operation **mapped-set** _, which returns the keys of a map.

Note that in AS, as well as the abstract data types which are specified in terms of sorts using unified algebraic specification, the notations of the other semantic entities, including actions and yielders, are also specified in this way,. AS defines the sort **action** and the sort **yielder**, as well as a number of constructors to create a fully-fledged range of actions and yielders.

3.3.3 Facets of Actions

To know AS and AM, first of all, it is necessary to understand the behaviours of actions—the 'instructions' of the AM, namely to know how actions change the state of a running AM.

So far, we have mentioned three kinds of information available in AM: transients, bindings and the stable information stored in the global store. From the perspective of users, these three kinds of information have different life spans and purposes:

1. Transients: a variably-sized tuple of data (including empty tuples), short-lived, corresponding to the intermediate results of execution steps.

2. Bindings: bindings of tokens to data, parallel to symbol tables, spanning across a mid-term. They are essentially entries of a map from tokens to data.

3. Stable: the data stored in cells of the global store. The global store can also be thought of as infinitely-sized map from cell to data.

An action may process one or more kinds of this information simultaneously, while some actions have no effects on information at all but only play roles as control flow or data flow. It is also possible that some actions act as a control flow or data flow and meanwhile process some information. Because of this, an action is claimed to have one or more of the following facets:

1. the basic facet, which carries out the function of control flow or data flow.

2. the functional facet, which processes transient data.

3. the declarative facet, processing scoped binding information.

4. the imperative facet, manipulating the store, such as allocating a cell, storing a value in a cell.

Note that there is a communicative facet as well, however we don't cover it in this introduction to AS. The facets of an action are independent of each other as every facet solely do its duty without interfering with other facets. Bear in mind that an action performance may *complete* (terminate normally), *escape* (exceptional termination), *fail* (terminate abnormally), or *diverge* (not terminate at all).

3.3.4 Yielders

Before introducing actions, we consider first yielders, which are a special kind of action. They are analogous to expressions in programming languages. Yielders are evaluated to produce data based on the *current information* present at the current state, including the current transients, the current bindings and the current store. Evaluation of yielders causes no change to the current information even though using it. Now we proceed to illustrate some important yielders as follows.

1. Transients yielders

- the given _ : data → yielder

  This is a constructor to form specific yielders given a subsort of **data**. Consider the yielder **the given truth-value**, whose evaluation yields the transient data of the current information if the transient data is an individual in the sort of **truth-value**(true or false). Otherwise, the evaluation of this yielder produces **nothing**.

- the given _#_ : Datum, PositiveInteger → Yielder

  This kind of yielder when evaluated will yield the nth (specified in the second argument) item in the transients, provided that it agrees with the sort specified in the first argument. Suppose the current transients is (true, 1), then the yielder **the given truth-value#1** will yield true, while the yielder **the given integer#1** will yield **nothing**.

2. Bindings yielders

   - the _ bound to _ : data, token → yielder

     These yielders yield the object bound to a token in the current bindings. For instance, suppose the current bindings are $\{x \rightarrow\ 3,\ y \rightarrow\ true,\ \ldots\}$, then yielder **the integer bound to x** yields 3; if the token is not present in the current bindings or the bound object is not an integer, it will yield **nothing**.

3. Storable yielder

   - the _ stored in _ : data, yielder → yielder

     When a yielder of this kind is evaluated, the sub-yielder specified as the second argument is evaluated first to get a cell, then this yielder is evaluated to yield an object which is stored in the cell. Likewise, if the object doesn't agree with the sort specified as the first argument, this yielder yields **nothing**.

4. Data-operation yielders

   - data : yielder

     It should be borne in mind that data (either built-in defined or user-defined) which statically occur in actions are also yielders. When evaluated, they

yield themselves. For one example, 'true' and 'false' are individuals of the sort **truth-value**; if they occur in actions, they become yielders and can be evaluated to yield themselves.

- data-operation :: yielder, ... → yielder

  The operators on data, if applied to suitable arguments and appearing in an action, are also yielders. For example, 'not _' is an operation over the sort **true-value** and becomes a yielder when applied to 'true' or 'false' in an action.

3.3.5 Functional

Now we illustrate some important functional actions which only have the functional facets hence their performance merely has effects on the transient date. (Note, having no effects on transient data or bindings data means not copying them to the next state. However, having no effects on the store means no change made to the store, as the store which holds data permanently does not demand explicit copy.)

1. give _ :: yielder → action

   The action **give Y1**, when performed, will cause current transient data to be set to the data yielded by Y1. Suppose, currently, AM is in a state <'give not true', (3, false), {x → 3, y → true}, {cell0 → 1, ...}>. After the action is performed, the new state is resulted as <'completed', false, empty-map, {cell0 → 1, ...} >. It can be noticed that transient data is replaced from true to 3; the store remains unchanged. Notably, the bindings are not copied to the next state, resulting in an **empty-map**. This is illustrated in Figure 3.1 (a).

2. regive : action

   This action simply copies the transient data to the next state. See Figure 3.1 (b).

3. check _ :: yielder → action

   This action is not primitive but composite. We present it here because it is frequently used. The performance of the action **check Y** begins with evaluating the yielder **Y**. If the result of the evaluation is true, then the performance of this action completes, otherwise, it fails. In either case, the transient data is set to

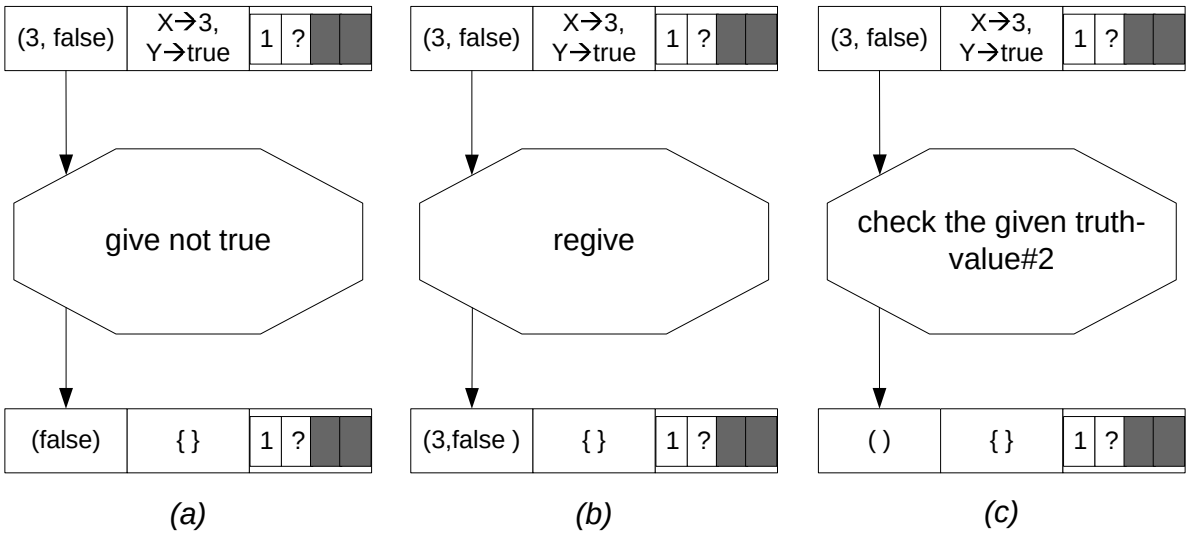


Figure 3.1: Performance of functional actions.

empty-tuple, and the binding information is set to empty-map. See Figure 3.1 (c).

3.3.6 Declarative

Declarative actions are concerned with bindings. The following actions are purely declarative, having no effects on transients or the store.

1. bind _ to _ :: yielder, yielder → action

   Actions of this kind are used to bind a token to an object. When an action of this kind is performed, the transient data is set to empty-tuple and the previous binding information is totally replaced by the new binding. See Figure 3.2 (a) for illustration.

2. rebind : action

   The action **rebind**, in contrast with the action **regive**, solely reproduces the current bindings without other effects, and thereby extends the scope of the bindings. See Figure 3.2 (b).

3. produce _ :: yielder → action

   In the action **produce Y**, Y yields a map which consists of bindings entries. When this action is performed, the yielded map will replace the current bindings. See Figure 3.2 (c).

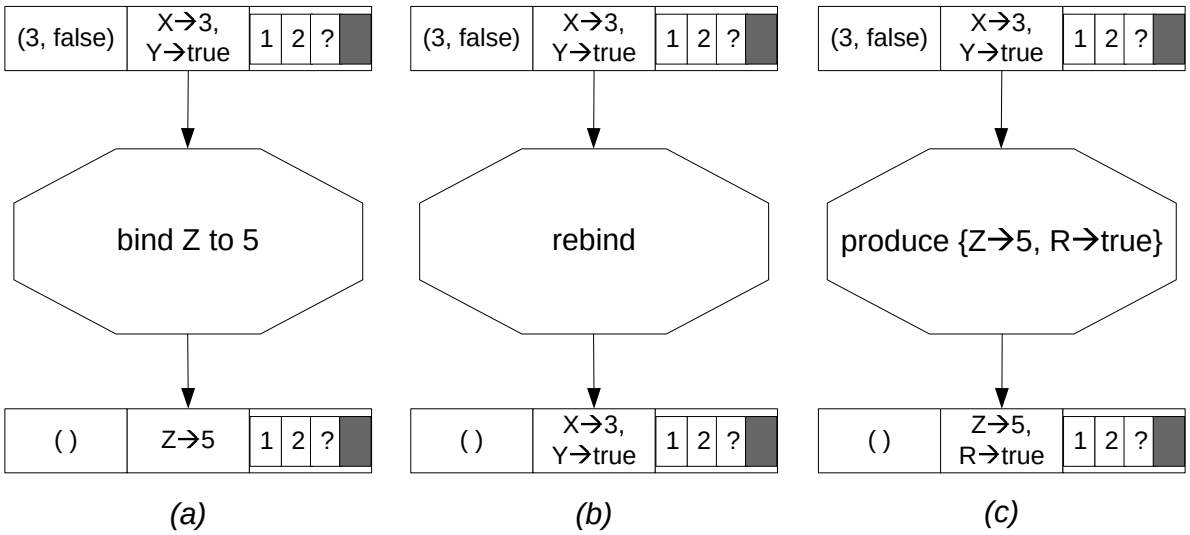


Figure 3.2: Performance of declarative actions.

3.3.7 Imperative

The imperative facet is concerned with the store, which contains an arbitrary number of cells. Each cell is in one of the three states [106]:

1. *defined*: a cell in this state implies that it contains a value.
2. *undefined*: the cells in this state are those that have been allocated but not yet employed to hold values.
3. *unused*: this means a cell has not been allocated.

So, it can be seen that a cell must be allocated before it can contain a value.

The following actions are commonly used to deal with the store.

1. store _ in _ :: yielder, yielder → action

   The action **store Y1 in Y2** puts the data yielded by Y1 into the cell yielded by Y2. Note that a cell can be regarded as a location, and is also a datum like an integer. Thus cells can be yielded, can be the transient data and can be bound to tokens. See Figure 3.3 (a) for illustration.

2. deallocate _ :: yielder → action

   The action **deallocate Y** changes the state of the cell yielded by Y to *unused*. See Figure 3.3 (b).

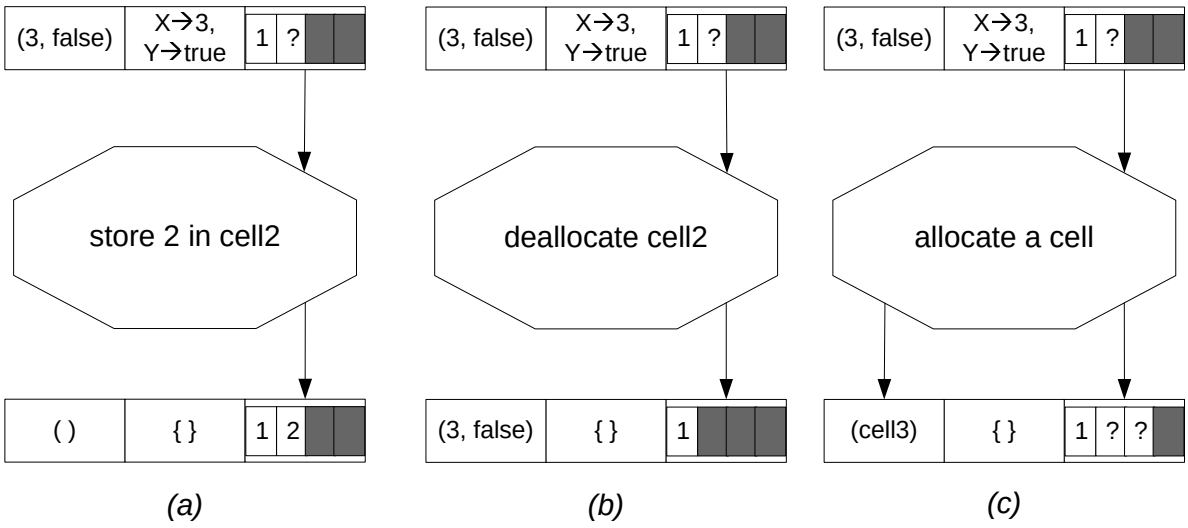


Figure 3.3: Performance of imperative actions

3. allocate a cell : action.

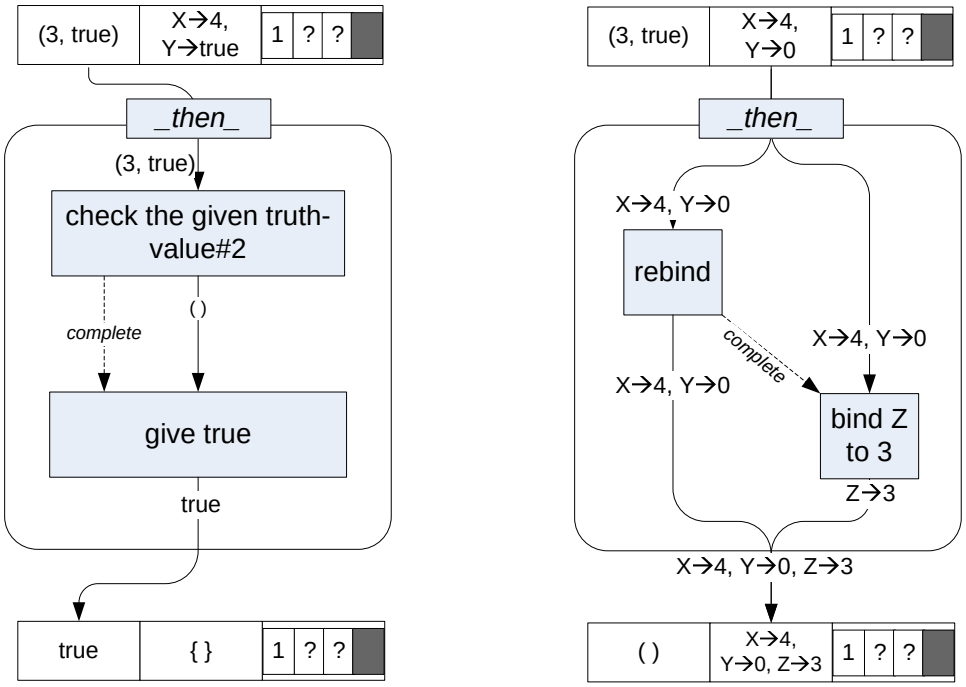
   This action is composite and imperative, and is very useful in specifying imperative languages. This action finds a unused cell in the store and changes its state to *undefined*. Note, in addition to changing the store, this action also produces the cell as the transient data. See Figure 3.3 (c).

3.3.8 Combinators

Composite actions are formed from simpler actions using infix actions or prefix actions which are also called combinators. In addition to the role of combining actions, they also play an important part in data flow (including transients flow and bindings flow) and control flow (including error propagation). Note that actions are not performed until both the control flow and data flow become available.

3.3.8.1 Functional Combinators

Functional combinators address several schemes of transient flow as well as control flow. They have the same policy on binding data flow: a functional combinator **A1** ***O*** **A2** copies its received binding data both to A1 and A2, and the output binding information is the result of merging those produced by A1 and by A2 (except for the action **A1 or A2**). If the merge fails, the whole action fails. (Note, **merge** means the disjoint-union of two maps. If the two maps have overlapping tokens, as specified in ASD 3.1, the



(a) transients data flow *(b) bindings data flow*

Figure 3.4: Data and control flow of '_ then _'

merge would return **nothing** which denotes the failure of the operation). Regarding transient flows of functional combinators, see the following.

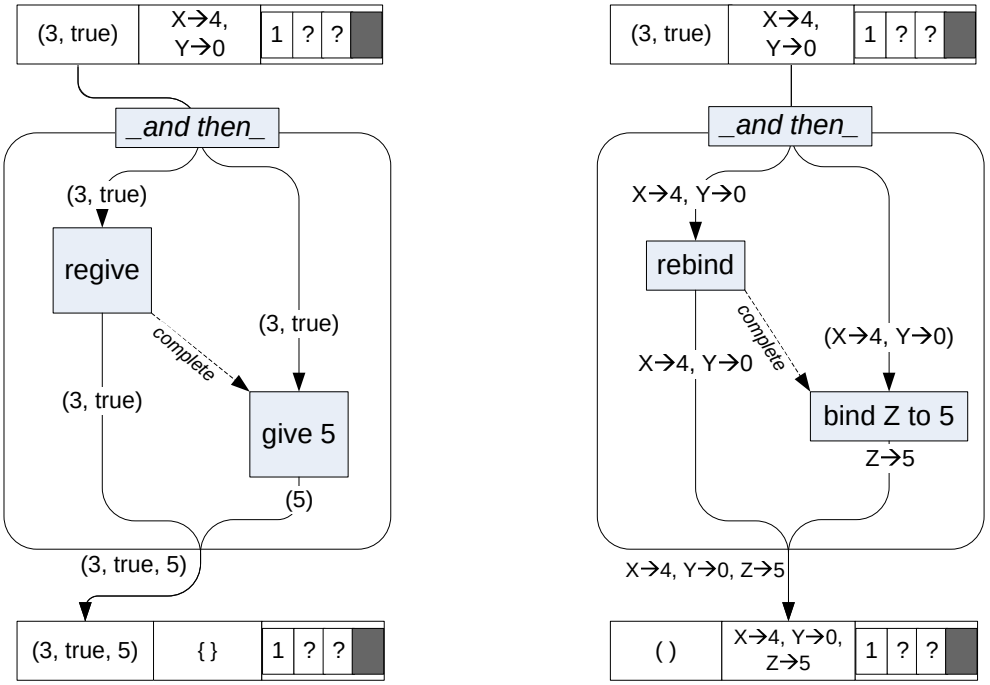
1. _ then _ :: action, action → action

   Control flow: the action **A1 then A2** enforces a control dependency of A2 on A1. That is, only when A1 completes can A2 be performed. If A1 fails, the whole action fails. See Figure 3.4 for an example of control flow; the control flow is denoted by dotted arrowed lines in the diagram.

   Transient data flow: the received transient data of the whole action is available for A1 only. If A1 completes and produces transients, then the produced transients become accessible to A2. The transients produced by the whole action are determined by A2. See Figure 3.4 (a) for the transient data flow of the action **(check the given truth-value#2) then (give true)**.

2. _ and then _ :: action, action → action

   Control flow: the action **A1 and then A2**, similar to **A1 then A2**, enforces a control dependency of A2 on A1. See Figure 3.5 for the control flow of the combinator _ **and then** _.



(a) transients data flow *(b) bindings data flow*

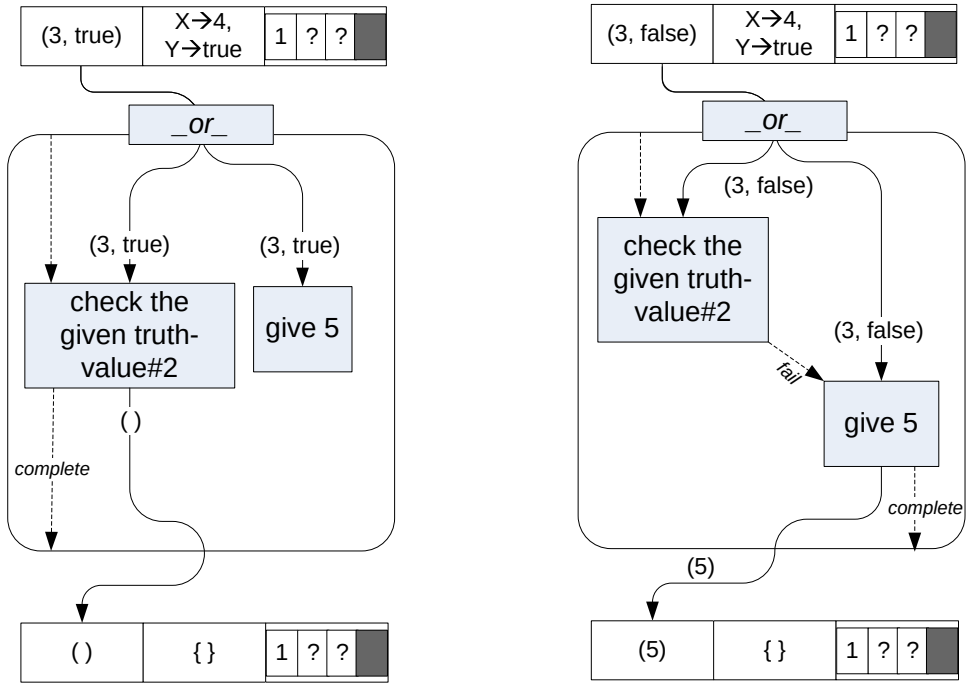
Figure 3.5: Data and control flow of '_ and then _'.

Transient data flow: Parallel to **A1 then A2**, the action **A1 and then A2** copies the received transients both to A1 and to A2. In addition, the transients produced by the whole action are the result of merging those produced by A1 and A2. See Figure 3.5 (a) for the transients data flow of the action **regive and then give 5**.

3. _ or _ :: action, action → action

   Control flow: the action **A1 or A2** represent some kind of non-determinism as this action either chooses A1 or A2 to perform. If the chosen sub-action completes, then it completes without performing the other action; otherwise, the action will try the other sub-action that is not chosen in the first attempt. The diagram (a) of Figure 3.6 shows that the first constituent action of the action **(check the given truth-value#2) or (give 5)** is chosen, and it completes without trying the other constituent action, whereas, in diagram (b) of Figure 3.6, the performance of the chosen constituent action fails, so it must try the other constituent action.

   Transient data flow: this action copies the received transients both to A1 and A2. The output transients are determined by the finally selected action.



(a) The chosen path completes. *(b) The chosen path fails.*

Figure 3.6: Non-deterministic choice of '_ or _'

Binding data flow: this action also copies the received bindings to both A1 and A2. Differently, the output bindings are determined by the finally chosen action.

4. _ and _ :: action, action → action

   Control flow: the action **A1 and A2** makes the steps of performances of A1 and A2 interleave in an arbitrary way. The completion of the whole action requires that both A1 and A2 complete. if either one fails, the whole action fails. See Figure 3.7 for the control flow of the action **regive and give 5** (diagram (a)) and the action **rebind and bind Z to 3**(diagram (b)).

   Transient data flow: this action copies the received transients to both A1 and A2. The output transient information is the merge of those produced by A1 and those produced by A2.

3.3.8.2 Delarative Combinators

In contrast to functional combinators, declarative combinators may differ in the policies on control and binding data flow. However they have the same policy on transient data flow. As far as transients flow is concerned, a declarative combinator **A1 *O* A2** copies

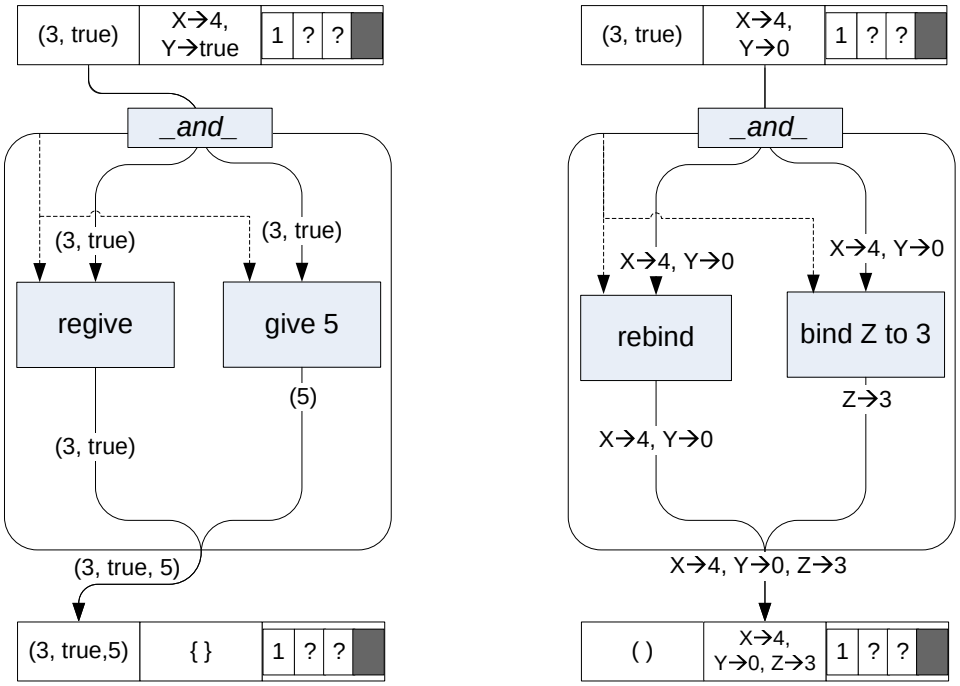


Figure 3.7: Data and control flow of '_ and _'

its received transients both to A1 and A2, and transients produced by the whole action result from the merge of those produced by A1 and by A2. Here, we highlight the following declarative combinators.

1. _ hence _ :: action, action → action

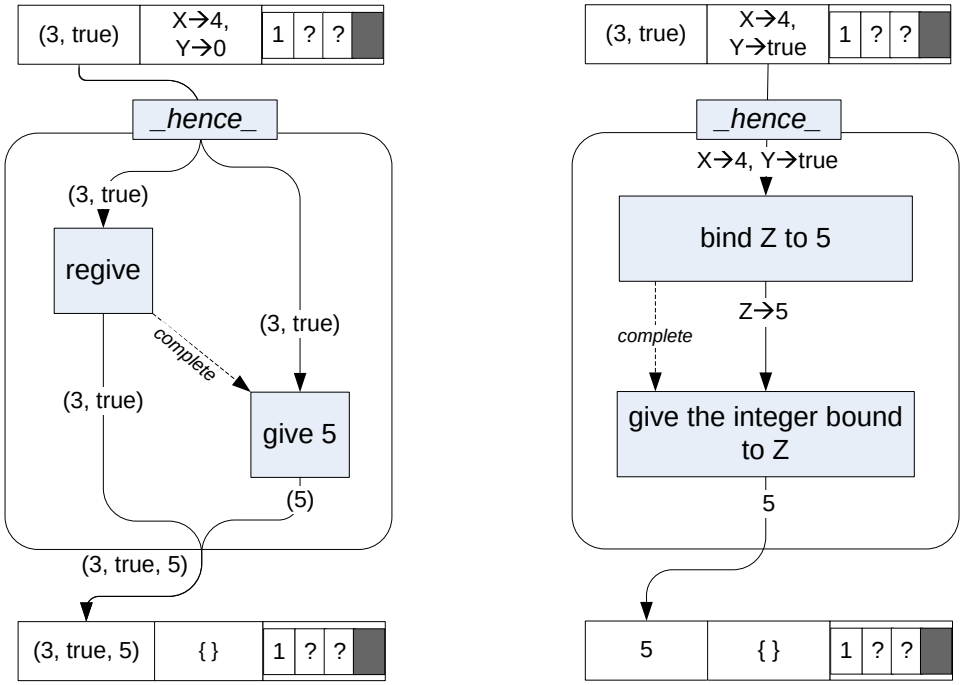
   Control flow: the action **A1 hence A2**, similar to the action **A1 then A2**, enforces a control dependency of A2 on A1. See Figure 3.8 for the control flow of the action **regive hence give 5** and the action **(bind Z to 5) hence (give the integer bound to Z)**, respectively shown in the diagrams (a) and (b).

   Binding data flow: A1 receives the bindings of the whole action. When A1 completes and produces bindings, the bindings then becomes available to A2. The output bindings of the whole action are solely determined by A2.

2. _ moreover _ :: action, action → action

   Control flow: similar to the action **A1 and A2**, the action **A1 moreover A2** allows A1 and A2 to be executed concurrently. When both of them complete, the whole action completes, otherwise, it fails.

   Binding data flow: this action copies the received bindings to both A and B. In



(a) transients data flow *(b) bindings data flow*

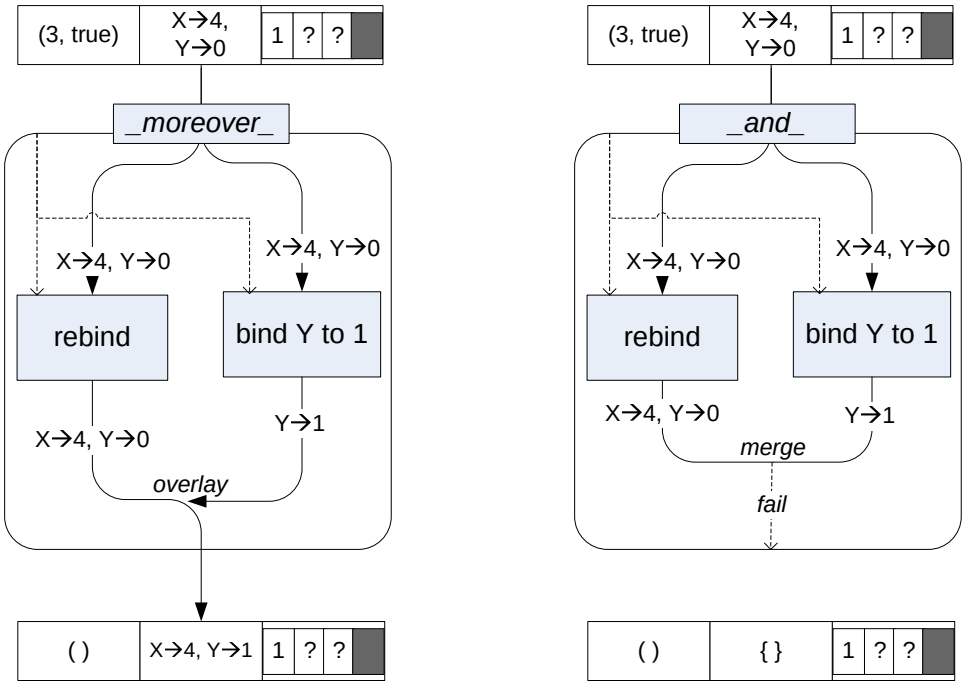
Figure 3.8: Data and control flow of '_ hence _'.

contrast to the action **A1 and A2**, the output bindings of the whole action is the result of **overlay** those produced by A with those produced by B. Note, the operation **overlay** and **disjoint-union** are both for combining maps. As far as two binding maps (M1, M2) are concerned, if there are no overlapping tokens, the two operations returns the same results. If overlapping tokens exist, disjoint (M1, M2) yields **nothing** representing failure; while overlay (M1, M2) yields M1, together with those bindings that M1 doesn't override. Figure 3.9 shows a comparison between two actions: **rebind moreover bind Y to 1** and **rebind and bind Y to 1**.

3. furthermore _ :: action → action

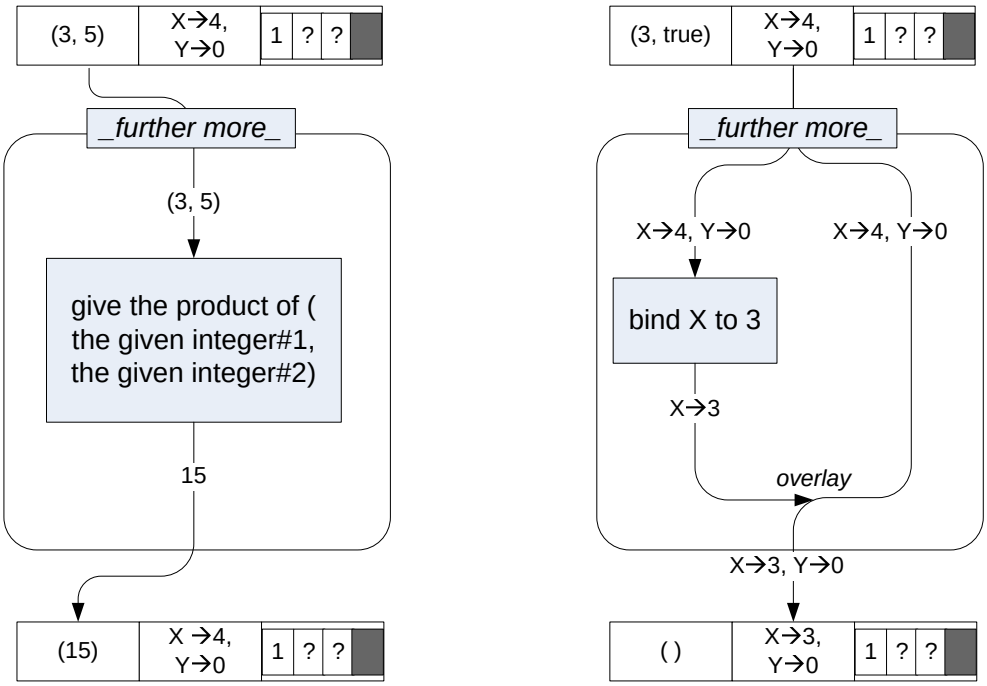
   The '**futhermore** _' is a prefix combinator, the shorthand for **rebind moreover** _ and very useful in describing semantics for procedures. Because it is prefix, it simply transfers the control to its only sub-action. The data flow is shown in Figure 3.10.

In this subsection, we illustrate some important actions that are extensively used in specifying action semantics for programming languages. However, this illustration is



*(a) _moreover_* *(b) _and_*

Figure 3.9: Difference between '_ and _' and '_ moreover _'



(b) transient data flow *(b) bindings data flow*

Figure 3.10: Data flow of '_ furthermore _'

not exhaustive and only intended to provide a general concept of action behaviours. As the text goes on, more and more actions are introduced where needed.

3.3.9 Action Semantics of IMP

As mentioned, AS is a framework for describing the semantics of programming languages. In this framework, one specifies the semantics of a programming language by mapping the semantics of the programming language into semantics of actions; hence, the resulting actions are called the denotation of this language from the viewpoint of denotational semantics. The translation is specified by a set of semantic functions. Generally speaking, every syntactic sort (non-terminal) has a corresponding semantic function to specify its semantics; each semantic function is defined by one or more equations, each of which is aimed at a possible form of the non-terminal. The equations are required to cover all the possible sub-sorts to assure completeness.

Syntax 3.2 Adjusted context-free grammar of IMP.

grammar:

(1) Statement = ⟦ ";" ⟧ | ⟦ Identifier "=" Expression ⟧ |
⟦ "if" Expression "then" Statement "else" Statement ⟧ |
⟦ "while" Expression "do" Statement ⟧ |
⟦ "{" statements "}" ⟧

(2) Statements = ⟨ Statement ⟨ ";" Statement ⟩* ⟩

(3) Declaration = ⟦ Type Identifier ";" ⟧

(4) Expression = Numeral | Identifier | ⟦Expression Infix-Op Expression⟧

(5) Infix-Op = "+" | "-" | "*" | "!=" | "or" | "and"

(6) Numeral = ⟦ digit+ ⟧

(7) Type = "num" | "boolean"

We still use the toy language IMP to illustrate how AS is employed to accomplish a semantic description. The abstract syntax of IMP needs to be re-formulated in AS style. So, we adjust the abstract syntax described in Syntax 3.1, particularly in the aspect of micro-syntax, to that shown in Syntax 3.2.

The new abstract syntax of IMP is described in a unified algebraic way. The left-hand side of a derivation rule introduces a new sort implicitly, and each production on

the right-hand side is represented also by a sort. Then the whole right-hand side represents a composite sort formed by the union of the sub-sorts representing productions. For this reason, each derivation rule is essentially a sort equation of unified algebra.

Note that AS defines a constructor (denoted by '⟦ ⟧') to construct tree sorts from the sub-tree sorts. For example, in Equation (4) in Syntax 3.2, Expression, Numeral, Identifier are all tree sorts. ⟦ Expression Infix-Op Expression⟧ constructs a new tree sort from the sub-tree sorts: Expression, Infix-Op and Expression. Equation (2) states that Statements are variable-sized tuples (tuples are also sorts in action semantics). To understand why Equation (5) is also an instance of a sort equation of unified algebra, recall that the individuals are treated as sorts (singletons) in unified algebra. The adequacy of unified algebra representing grammars were stated in [66].

Both the grammars and the semantic functions are represented in an algebraic way. Each semantic function is represented as an operation, usually from a syntactic sort to a semantic sort including actions, yielders and data (including the built-in and user-defined data sorts). Semantic equations are used to define the corresponding semantic functions and are essentially sort equations of unified algebraic specifications. Thus, all artefacts in action semantics are specified in unified algebra.

ASD 3.2 Module **Data** define sorts **value, storable and bindable**

Data

needs: [**Mosses 1992**] **/(Data Notation, Action Notation).**

- value = truth-value | number | cell
- storable = value .
- bindable = cell .

Before specifying the semantic functions for IMP, for convenience, we define a sort **value**, which is formed by the union of three subsorts: **truth-value**, **number** and **cell**. A value is usually the result of evaluating an expression. Moreover, AS requires to define the sort **storable** if the store is referenced. The definition of **storable** makes clear which data can be stored. Likewise, the sort **bindable** is defined to show which data can be bound. In the case of IMP, the definition of these sorts is shown in ASD 3.2

The semantic function for declaration is shown in ASD 3.3. It uses the current

ASD 3.3 Module **Declaration** defines the semantic function for Declaration
Declaration

needs: **Data**

introduces: elaborate _ .

- elaborate _ :: Declaration → action [binding | storing] [using current storage]

(1) elaborate ⟦ Type i: Identifier ";" ⟧ = allocate a cell then bind i to it.

---

storage because it allocates a cell and binds the identifier to the allocated cell. The **'it'** that occurs in the **'bind i to it'** is actually a pre-defined shorthand for **the given cell**. Note that the **action** [**completing** | **binding** | **storing**] constructs a sub-sort of the sort **action** by attaching effect descriptors, which means such kinds of actions will have effects on bindings and the store, and complete when performed. By the means of providing effect descriptors for actions as well as a set of inference rules of the effects, the consistency of an action semantics description (ASD) can be checked statically.

The action semantics of expressions are specified by the semantic function **evaluate** _ **:: Expression → action** [**giving a value**] shown in Figure 3.4, where the sort **action** [**giving a value**] is a sub-sort of the sort **action** which, when performed, will have effects on transient data: giving a value. Under some circumstances, it is useful for **action** [**giving integer**^{2}] to represent a kind of action which when performed will give a tuple of two integers.

The semantic function **the operation-result of** _ is intuitive, where the **sum of** _, **difference** _, etc, are pre-defined data operations.

The major task is to specify statements. See ASD 3.5 for their description. The **action**[**completing** | **diverging** | **storing**] means that the action denoting a statement may complete, diverge or change the store when performed. Note that **action**[**completing**] ≤**action**[**completing** | **diverging** | **storing**]. ASD 3.5 is explained as follows:

- Equation 1 means the execution of an empty statement ('skip') is equivalent to the meaning of the action **complete**, which is a well-defined basic action and has no effect on transients, bindings and store when performed.

- Equation 2 defines the semantics of assignment. The execution of an assignment will evaluate the identifier and the expression concurrently, resulting in a binary

ASD 3.4 Module **Expression** specifies semantic functions for Expression

Expression

needs: **Data** .

introduces: evaluate _ , the operation-result of _ , the value of _ .

- evaluate _ :: Expression → action [giving a value].

(1) evaluate n: Numeral = give the value of n.

(2) evaluate i: Identifier = give the cell bound to i.

(3) evaluate ⟦ e_1:Expression o: Infix-Op e_2:Expression ⟧ =
(evaluate e_1 and evaluate e_2) then give the operation-result of o.

- the operation-result of _ :: Infix-Op → yielder [of a value] [using the given value[2]]

(4) the operation-result of "+" =
sum of (the given number#1, the given number#2).

(5) the operation-result of "and" =
both of (the given truth-value#1, the given truth-value#2).

(6) . . .

- the value of _ :: Numeral → Number.

(7) the value of n: Numeral = number & decimal n.

---

tuple consisting of the resulting cell and the resulting value (Note, their order is significant). The tuple is available as transient information to the next action which stores the value in the cell.

- Equation 3 defines the semantics of conditional choice. It is worth highlighting that if evaluation of 'e' gives false, the '**check the given truth-value**' would fail, then incur the failure of the '**check the given truth-value and then execute** s_1'. As a result, this chosen path fails and then the other path is tried. Note, the choice of alternative path to perform is non-deterministic and dependent on the implementation of AM.

- Equation 4 defines the semantics of conditional iteration, where an important action is used: **unfolding A**. Normally, there is an occurrence of the dummy action **unfold** in A, which can be considered to represent the action **A**. When 'unfolding A' executes A and whenever it reaches **unfold**, it performs A instead.

ASD 3.5 Moduel **Statements** defines the semantic function for Statements
Statements

needs: **Expression**

introduces: execute _ .

- execute _ :: Statements → action [completing | diverging | storing].

(1) execute ⟦";"⟧ = complete .

(2) execute ⟦ i: Identifier "=" e: Expression ⟧ = (evaluate i and evaluate e)
then store the given value#2 to the given cell.

(3) execute ⟦ "if" e: Expression "then" s_1: Statement "else" s_2:Statement ⟧ =
evaluate e then
| | check the given-truth-value and then execute s_1
or
| | check not the given truth-value and then execute s_2.

(4) execute ⟦ "while" e: Expression "do" s: Statement ⟧ =
unfolding
| evaluate e then
| | | check the given truth-value and then execute s
| | | and then unfold
| | or
| | | check not the given truth-value and then complete

(5) execute ⟦ "{" s: Statements "}" ⟧ = execute s.

(6) execute ⟨ s_1:Statement ";" s_2:Statements ⟩ = execute s_1 and then execute s_2.

- Equation 5 specifies block statements.
- Equation 6 specifies the semantics of a sequence of statements. It performs the statements one by one: only when the preceding statement is performed completely, can the next statements be performed.

It seems strange that we solely provide a semantic function for the syntactic sort **Statements**, not for **Statement**. This is because the sort **Statement** is a sub-sort of **Statements**, in that, in unified algebra, no distinction is made between a 1-nary tuple and the single element in this tuple. Hence, the semantics for **Statement** has been already defined via the semantic function for **Statements**.

3.3.10 Abstraction

So far, we have illustrated a variety of actions which can be utilized to describe some important constructs of programming languages, such as expressions, variable declarations and commands. However, we did not mention how to specify abstractions in AS such as functions and procedures which are popular in many programming languages.

In programming languages, the body of *function* is an expression that will be evaluated whenever the function is called, while the body of a *procedure* is a command (mostly, a sequence of commands) that will be executed when the procedure is invoked. In AS, whether for expressions or commands, their meanings are denoted by actions. However, the action representing a function body or procedure body is not executed where it occurs, instead, it is encapsulated, bound to an identifier (the function or procedure name) and then executed by an explicit call to it elsewhere. AS provides the following facilities to fulfil this process.

1. abstraction of _ :: action → abstraction

   This is a constructor for constructing an abstraction from an action. The term **abstraction of A** incorporates A, which usually is the denotation of a function body or a procedure body.

2. bind _ to _ :: yielder, yielder → action

   This action has been mentioned previously and can be used to bind an abstraction to an identifier.

3. enact _ :: yielder → action

   The action **enact Y** causes the performance of the action incorporated in the abstraction yielded by Y.

The action **enact abstraction of A** causes A to be performed in an environment where both the transient and binding information are empty. However, in a practical programming language, a function or a procedure is performed in the environment either at declaration time or at invocation time, rather than a simple empty one. The former case is called static binding, and the later called dynamic binding. To achieve static and dynamic binding, action semantics provides:

$$closure\ of\ \_ :: yielder \rightarrow\ yielder$$

For example, the yielder **closure of abstraction of A** when evaluated attaches the current bindings to the abstraction **abstraction of A**. In fact, this evaluation often takes place in two kinds of time. If it takes place at declaration time, the static binding is achieved; if it takes place at the invocation time, the dynamic binding is achieved. To better understand this, **closure of abstraction A** can be viewed as **the abstraction of produce M hence A** where M is a binding map determined by the current bindings when this abstraction is evaluated.

The term **application of _ to _** is akin to **closure of _** , but it is used for attaching some transient data to an abstraction in the declaration time. For instance, the term **application of 3 to abstraction of A** attaches 3 to the abstraction. If one needs to attach the current transient (in the invocation time) to the abstraction, just use **application of the given value to the abstraction of A**.

To show this concept and a parameter passing mechanism, we extend IMP as in ASD 3.6 to incorporate procedural abstraction. (For simplicity, only one formal parameter is considered in a procedure. This can be easily extended to allow multiple formal parameters.)

ASD 3.6 Exentension is made to IMP grammars to incorporate abstractions

(1) Statement = ... | ⟦ "call" Identifier "(" Expression ")" ⟧

(2) Declaration = ... | ⟦ "procedure" Identifier "(" Formal-Parameter ")" "{" Statements "}" ⟧

(3) Formal-Parameter = ⟦ Type Identifier ⟧.

So the action semantics for IMP is accordingly extended as shown in ASD 3.7, and it is explained as follows:

- Equation (2) defines the declaration of procedures where 'the closure of ...' occurs in the declaration time to achieve the static binding. As such, the statements in the procedure body are executed in the binding environment as a result of overlaying the environment in the declaration time with the bindings produced by the formal parameter.

- Equation (3) reflects the copying mechanism of parameter passing. The passed value is stored in a newly-allocated cell, which then bound to an identifier for later use.

ASD 3.7 Extensions are made to the action semantics of IMP.

- elaborate _ :: Declaration → action [binding | storing] [using current storage]

(1) elaborate ⟦ Type i:Identifier ";" ⟧ = allocate a cell then bind i to it .

(2) elaborate ⟦"procedure" i:Identifier "(" fp: Formal-Parameters ")"
"{" s:Statements "}" ⟧ =
bind i to the closure of the abstraction of
| furthermore
| | respectively formally bind fp
| | hence
| | execute s

- formally bind _ :: Formal-Parameter → action [binding | storing] [using the given value | current storage].

(3) formally bind ⟦ t:Type i:Identifier ⟧ =
allocate a cell and give the given value
then (store the given value#2 in the given cell#1 and bind i to the given cell#1)

- execute _ :: Statements → action [completing | diverging | storing].

...

(4) execute ⟦"call" i:Identifier "(" e:Expression ")" ⟧ =
evaluate e then enact the application of the given value to the abstraction bound to i.

- Equation (4) defines the semantics of calling a procedure. First, the actual parameter (actually an expression) is evaluated giving a value. Then this value is attached to the abstraction representing the procedure. Note the act of attaching the value happens at invocation time (rigorously, just before the abstraction is invoked).

3.3.11 UPAS versus AS

In Chapter 2, we introduced UPAS, which is a part of the UML specification and intended to provide precise semantics for the minimum behavioural units of UML. It is very easy to confuse UPAS with Mosses's AS. To clarify them, a comparison is made as follows.

AS shares some features with UPAS. 1) They both take actions as fined-grained and fundamental semantic entities, and complex semantics is established based on them. 2) Both of them provide some ready-to-use notations to denote control flow and data flow. However, UPAS adopts an activity model for this, but AS uses combinator actions. 3)

Both semantics are compositional, although UPAS is not explicitly so.

Despite these similarities, UPAS and AS differ in the following aspects.

AS is a general-purpose framework being able to describe the semantics of a large range of languages; however UPAS is not a general framework and it is only aimed to provide a more precise semantic basis for an action language for UML. This is because 1) UPAS only defines the semantic entities (actions), but provides no mapping mechanism from syntactic objects to semantic objects, which is imperative for a framework. 2) UPAS is intended for modeling languages, so its actions are larger-grained than those of AS for high-level abstractions. Hence, these actions are not fundamental enough to describe some low-level languages. For example, UPAS lacks the actions dealing with bindings, cell allocations, and so on.

In AS, the semantic entities, namely actions, yielders and data, are formally defined in other semantic description frameworks, such as algebraic specification and structural operational semantics. The semantic entities in UPAS are solely defined in precise English but not formally in a well-established semantic framework as AS.

We quote from Mosses [67] to conclude the comparison:

> The UML Action Semantics is to some extent similar in spirit to the original Action Semantics framework, although there are major technical differences (p69).

3.4 Conclusion and Discussion

This chapter introduces two traditional frameworks, *operational semantics* and *denotational semantics*, which are closely relevant to action semantics. Operational semantics specify semantics of a programming language by defining a transition machine, and denotation semantics translate syntactic entities into mathematical objects.

AS combines features of operational semantics, denotational semantics and algebraic specification. AS uses *actions* rather than cryptic mathematical objects as the denotations to gain better modularity and intelligibility. AS employs *structural operational semantics* to provide formality to action denotations. AS adopts algebraic specification as formalism to provide itself with a very flexible type system that allows new types of data (sorts) to be user-definable. Due to these facts, action semantics is said to be a

hybrid framework.

We are determined to select AS to describe the formal semantics of UML for reasons from two sides: the AS side and the UML side.

- the AS side: 1) AS was developed for comprehensibility, modularity and practicability to overcome the mentioned drawbacks of traditional formal techniques. 2) AS is a fully-fledged formal framework which has been applied to various diversified languages and supported by several tools. 3) AS takes the advantages of denotational semantics, operational semantics and algebraic semantics, but hides user-unfriendly details of these frameworks. 4) AS itself is extensible in that users can define their own actions and their own abstract data types.

- the UML side: 1) compared to other computer languages, UML is intended for more general users, including users who have no strong background in Computer Science. Thus its formal semantics demands more understandability. 2) UML is made up of various diagrams, some of which can exist independently for special purposes. UML is still evolving, possibly unifying more modelling techniques with its diagrams. As such, the formal semantics of UML entails high modularity and superior extensibility. 3) As mentioned, UPAS, a major part of UML dynamics, shares some concepts with AS.

In addition, AS is a mature semantics-describing framework, which has been successfully used to describe a diversified variety of real programming languages such as standard ML [104], Pascal [68], Java [21], ADA [65] and ANDF-SF[38]. In addition, various prototype compiler generators based on action semantics [20, 84, 70, 83] and AS-aware tools like ASD [102] have been developed.

Chapter 4

Formalizing UML with Action Semantics

We propose a new approach to formalizing UML, which is distinguished by selecting Action Semantics (AS) as the vehicle. We first work out a toy executable subset of UML, which is referred to as xUML in this thesis, to represent typically the full version of UML, and then specify xUML indirectly by formalizing its textual correspondent, an extended Action Language (ALx), using AS. This is a translation approach that comprises two sub-translations: from xUML to ALx and from ALx to AS semantic entities.

In the current chapter, we first describe the general idea of this approach, followed by the introduction of xUML and ALx. Then, we provide the action semantics for some important and unique constructs of ALx, accompanied by informal explanations. Finally, we discuss some limitations of such an attempt at formalizing UML.

4.1 Our Approach to Formalizing UML

The overview of our approach to formalizing UML is illustrated in Figure 4.1. Firstly, xUML, an executable subset of UML, is mapped to its textual counterpart, ALx; this translation is guided by formal translation rules in terms of mapping models. Secondly, the syntactic sets of ALx are translated into AS semantic entities; both the syntactic sets of ALx and the translation rules are defined using the AS framework.

Our approach has a distinctive feature: instead of defining UML directly, we design

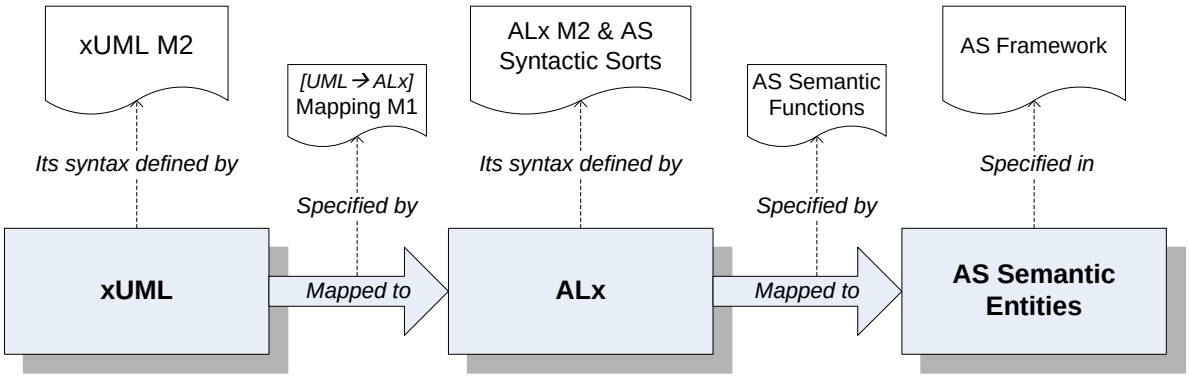


Figure 4.1: Overview of our approach to formalizing xUML

a textual programming language, ALx, and use it as the intermediary between UML and its action semantics. We do not formalize UML directly for the following reasons.

- Since action semantics are syntax-directed and compositional, we must specify the abstract syntax *tree* of UML before composing its action semantics. However, the abstract syntax of UML is not tree-structured, at least not intuitively, in that it is formalized in an object-oriented four-layer metamodeling architecture, namely using graph-like class diagrams.

- UML, as a modeling language intended for early stages in system development and a broad spectrum of different application domains, unavoidably includes some ambiguous and execution-unrelated constructs. So we need to remove these semantics-weak constructs and confine our attention to an executable subset of UML.

Consequently, we need a textual programming language corresponding to a rigorous and executable subset of UML as an interface between xUML and its action semantics. This textual correspondence is required to genuinely embody the major dynamic semantics of UML. We consider that an AL is the best candidate because it definitely incorporates most major dynamic semantics of UML; after all, ALs are created to provide concrete syntax for the basic behavioural units of UML—actions.

Hence, the complete formal description of xUML is constituted by the following artefacts:

- The syntax of xUML, which is defined by UML itself, called xUML metamodels.

- The syntax of of ALx. The abstract syntax of ALx is defined both in UML and in AS.

- The mapping rules between xUML and ALx, which occurs as mapping models from xUML to ALx.
- The action semantics of ALx, which is specified using AS framework.

4.2 xUML

For simplicity, our customized UML, xUML, is an unambiguous small subset of UML, which is enhanced by ALx to specify method bodies and activities in state machines. From the perspective of users, xUML offers three types of graphical diagrams: class diagrams, class collaboration diagrams and state charts. Those diagrams that are common in a fully-fledged UML, such as use cases, object interaction diagrams and sequential diagrams, are not investigated in the current research.

Class diagrams. Class diagrams of xUML, akin to those of a full UML, are employed to model the static aspect of the system using the object-oriented concept. The current xUML supports two kinds of relation: *generalization* and *association*.

The xUML class diagrams are distinguished from those of the full UML versions in the following aspects. 1) Each association must be named uniquely. This is compulsory in xUML because the link-navigation construct of ALx needs to reference associations by name. 2) Method bodies are defined in ALx to achieve computational completeness, which is reflected in the xUML metamodel by reusing the definition of *Block-Statement* in ALx to specify xUML operations.

The type system of xUML categorizes types as built-in types and user-defined types. The built-in types include primitive types and commonly-used generic types; classes defined by users in class diagrams are a major kind of user-defined type. In addition, user-defined types also allow users to define their own enumeration type, etc. The type system of xUML is a static type system. It should be noted that the type system of xUML is aligned to that of ALx so that ALx code can be integrated into xUML diagrams seamlessly and logically.

Class collaboration diagrams. These diagrams play two roles, declaring events, which will be referenced later in state charts, and giving a visual view of the collaboration of the classes in the system. In xUML, there are two types of events, *call events* and *signal events*. The former represent occurrences of the calls of methods, and the latter

are originated from the construct 'event-generation' in ALx. They are distinguished on the surface in this way: call events are denoted by solid arrowed lines, whereas signal events are denoted by arrowed broken lines. The necessity of declaring events arises from the fact that they will be referenced later in state charts.

State charts. A state chart is normally used to depict all possible states that objects of a class may reside in during its life cycle, thus a state chart is always associated with a class as the complementary description of dynamic behaviours. State transitions are trigged by events, either call events or signal events, and subsequently cause the execution of the exit action of the source state and the entry action of the target states, In xUML, we do not consider nested, pseudo, history states, and so on.

A complete xUML metamodel is defined using UML and provided in Appendix D.

4.3 ALx

It is necessary to create a new AL for our purpose of formalizing UML. So far, several ALs have existed for years, such as the Action Specification Language (ASL) [107], the BridgePoint Action Language [87], and the widely-used SDL [13] in the telecommunications industry. Although these ALs, generally speaking, have reflected the actions of UPAS in some ways, they each are not complete programming languages, rather they are more like scripting languages intended for being embedded in UML models in that they lack model description constructs to represent UML diagrams such as class diagrams and state charts. Therefore, we cannot simply reuse one of them as the intermediary between UML and its action semantics, and need to extend a current action language to be used as the counterpart of xUML.

The newly-created extended action language, called ALx, consists of two parts: a part for the common functionality available in the present ALs, and a model-describing part that can be viewed as the textual counterpart of the graphical xUML. See Figure 4.2 for illustration. The two parts, on the one hand, are integrated together seamlessly to form a textual and computationally-complete modeling language. On the other hand, the former part itself can be embedded in graphical UML models to specify method bodies and activities in state machines.

The model-describing part of ALx contains constructs that are textually and intuitively mapped to graphical elements in xUML, including the class diagram, class

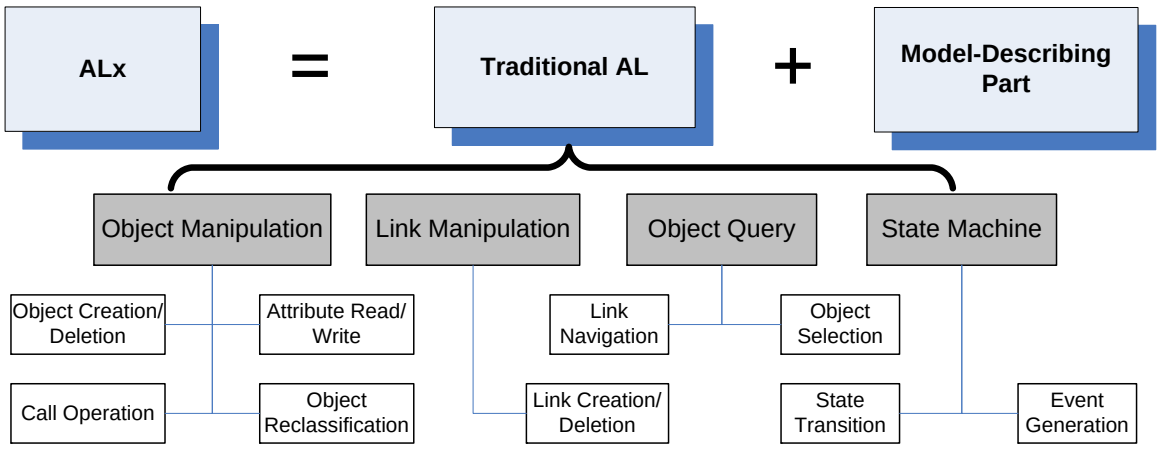


Figure 4.2: Constitution of ALx and its major constructs.

Category	Constructs	Syntax
Object Manipulation	Object-Creation	"create-object" <*object-reference*> "of" <*class*> *"("* *actual-parameters* *")"*
	Object-Deletion	"delete-object" <*object-reference*>
	Read-Attribute	<*object-reference*> "." <*attribute*>
	Write-Attribute	<*object-reference*> "." <*attribute*> "=" <*expression*>
	Call-Operation	<*object-reference*> "." <*method*> "(" *actual-parameters* ")"
	Object-Reclassification	"reclassify" <*object-reference*> <*class*> *"->"* <*class*> ;
Link Manipulation	Link-Creation	"link" <*object-reference* > "->" <*object-reference* > "(" <*relation* > ")" "link" <*object-reference* > "->" <*object-reference* > "(" <*relation* > ")"
	Link-Deletion	"unlink" <object-reference> "->" <object-reference> "("relation")";
Object Query	Link-Navigation	<object-reference> "=" <object-reference> "->" <relation> "(" <condition> ")" <collection-reference> "=" <object-reference> "->*" <relation> "("<condition ")"
	Object-Selection	"select-one" <*object-reference* > "of" <class> "(" <condition> ")" "select-many" <*collection-reference* > "of" <class> "(" <condition> ")"
State Machine	State-Transition	<*object-reference*> ">>" <*state*>
	Event-Generation	"*send-event* " <event> "->" <*object-reference* >

Figure 4.3: Typical set of ALx constructs

collaboration diagram and state chart of xUML. The detailed description of this part is ignored because the mapping between the two is very straightforward (Appendix J contains example excerpts from ATL files). In regards to the other part, the major constructs which are imported from conventional ALx, generally fall into the following four categories (See Figure 4.3 for details):

- Object-manipulating constructs. This includes those for creating/deleting objects, reading/writing attributes of objects, invoking operations of objects and reclassifying objects to a new generalization hierarchy.

- Link-manipulating constructs. This contains constructs for creating links and deleting links.

- Object-query constructs. This is intended to implement the object-query mechanism of ALx. It is self-evident that Object Selection shall be classified as this kind. Link Navigation is classified also as this kind because it also allows users to retrieve one or multiple existing objects in the run-time environment just as Object Selection does.

- State-machine-related constructs. Two constructs are involved in this kind. State Transition is aimed to trigger state transition via no occurrence of event, and it is generally used only in the internal activities of state machines. Event Generation is intended to issue events to state machines.

Note that the constructs illustrated here are not exhaustive.

4.4 xUML-to-ALx Mapping Models

In Section 4.1, we mentioned that the xUML-to-ALx mapping model is also an integral part of the specification of xUML. The fact is that the model-describing part of ALx is specially designed to be a textual correspondence of xUML, which implies that the mapping between the two is extremely intuitive. In other words, the metamodels of xUML and ALx are almost identical except for some treatments of microsyntax for avoiding naming collision or compliance with naming conventions. As a result, it is not necessary to explain the xUML-to-ALx mapping models here.

4.5 Running Example

In order to let the reader have some sense of xUML and ALx, we envisage a simple elevator serving building sites, and then employ xUML to model it in graphical notations. The resultant models are subsequently represented textually using ALx and ultimately translated into Java code. To simulate this model, a use scenario of this elevator (a sequence of operations on the lift) is singled out and coded in ALx.

The elevator is a fictitious basic machine for lifting passengers, most likely building staff, upward and downward. It is operated by passengers onboard using a mounted controller with four buttons. A door-switching button is pressed to either open or shut the door, depending on the present state of the door, closed or open. Two moving buttons are available for moving the elevator: one for moving the elevator up, called the moving-up button; the other one for moving the elevator down, called the moving-down button. The last button is used to stop the moving lift. A typical scenario of using this lifting machine can be described by the following steps:

1. A passenger enters this lift and shuts the door by pressing the door-switching button.

2. Consequently, the passenger chooses one of the moving-up/down buttons to move the elevator in the direction desired.

3. When the passenger reaches the destination floor, he/she presses the stop button to make the elevator stop and then steps off the elevator.

The passenger may change the moving direction of the elevator and needs to press the stop button first and then press the reverse moving button.

Now we model the system using xUML. First, a class diagram, shown in Figure 4.4, is authored to model the static aspect of the system. This class diagram is not intended to be the best modeling practice but for the simplicity of illustration.

The most notable characteristics in the class diagram is that every association is uniquely named; for instance, the association between the class 'Controller' and the class 'Elevator' is named 'R4'. Imposing unique names on associations makes associations identifiable by name, which is essential to the mechanism of link navigation. In contrast, naming associations is optional in UML specifications. In addition, the strong

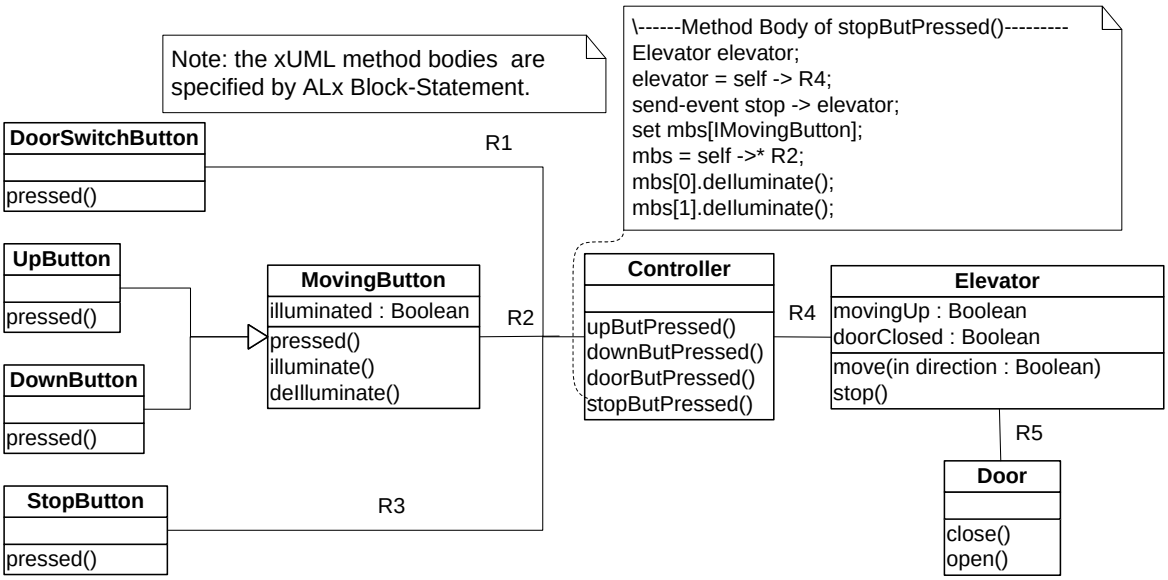


Figure 4.4: Class diagram of the elevating system.

associations, such as compositions and aggregations, are not syntactically distinguishable from normal associations in xUML, and they are treated equally at the semantic level as well. As to the class diagram shown in Figure 4.4, the association 'R5' is better modeled as a composition in that the door is an integral part of the lift both logically and physically. However, xUML is unable to represent this relationship, after all it is a toy executable UML. One can extend it to obtain more modeling capacity. Furthermore, xUML lacks constructs to specify the multiplicities and the names of association ends.

The elevator is the central object in the system and has various operational states, so a state chart, shown in Figure 4.5, is composed to specify the behaviours of the class 'Elevator', virtually a singleton class. The states of the lift are described as follows:

- StoppedWithDoorOpened. In this state, the elevator is parked with its door opened. When the lift enters this state, it calls the method 'open' of the class 'door' to open the door. This state can only transit to the State 'StoppedWithDoorClosed', and it has an exit activity to be invoked when moving out of this state for closing the door. Note that this state is the initial state of the elevator.

- StoppedWithDoorClosed. In this state, the elevator is stopped in a position with the door closed.

- MovingUp. This state indicates that the elevator is moving up. The entry activity

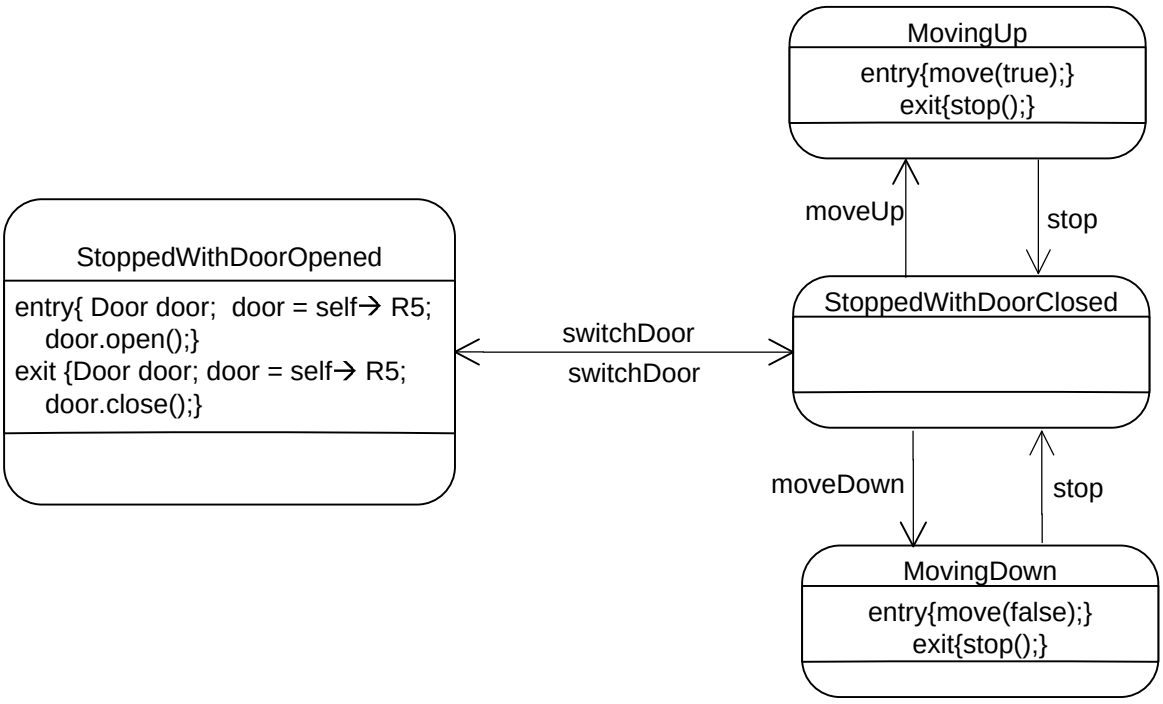


Figure 4.5: State chart of Elevator.

of this state is a call to the lift to start its motor to move itself upward, while the exit activity of this state is a call to stop it moving.

- MovingDown. This state is the opposite of 'MovingUp'.

In the state chart, state transitions are depicted by arrowed lines adjacent to which the names of events triggering the state transitions are specified. The following important implications can be understood from the transitions present in the state chart.

- The door of the elevator must be closed before it can be moved up or down.
- If the elevator is in motion, e.g., in a state of MovingUp or MovingDown, the door is bound to be closed.
- If the elevator is required to move in an opposite direction, it must be stopped first.

The last diagram is a class collaboration diagram intended to model interactions by means of specifying the events that may occur between the objects. See Figure 4.6 for illustration. The interactions between entities residing in the running system are carried out via the occurrence of events. Two kinds of events are considered in xUML, *signal events* and *call events*. See Figure 4.6: the *call events* are graphically expressed

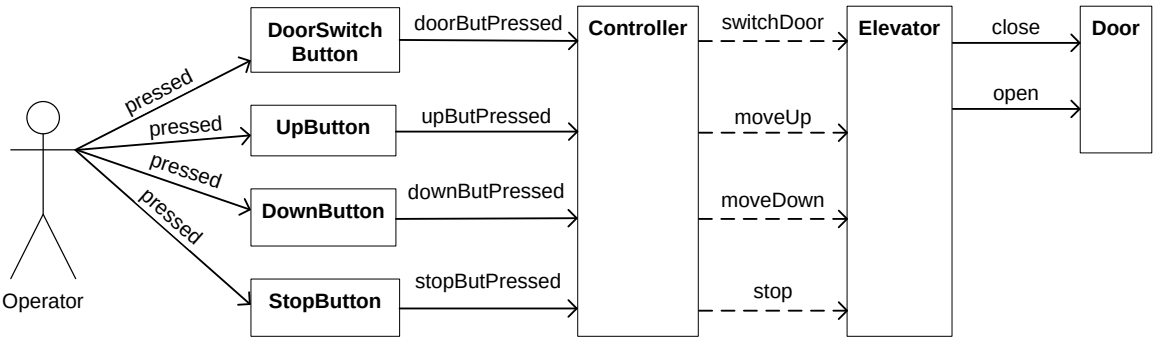


Figure 4.6: Class collaboration diagram of the elevating system.

by solid arrowed lines, whereas the *signal events* are expressed by broken arrowed lines instead.

Semantically, the occurrence of call events would invoke the corresponding operations. Call events modeled in class collaboration diagrams are solely for the purpose of illustration and conceptualization. However, the signal events occurring in the collaboration diagram are intended to declare signal events so as to be referenced in state charts. Currently, events are simply modeled as labels of type *string*, compared to the those specified in UML, where events are also a kind of classifier and thus may have attributes, operations and other facilities.

As mentioned, the graphical xUML model can be represented textually using ALx. See Figure 4.7 for the ALx code of the elevating system.

4.6 Related Action Semantics

AS has been applied to model a diversified range of realistic languages, such as Java [105, 21], standard ML [104] and Pascal [68]. Most relevant to our research is [105], in which Watt utilized AS to describe the semantics of JOOS, a subset of Java concerning the main concepts, such as classes, fields, inheritance, dynamic method selection and object constructors.

We have the following thoughts about the semantics of JOOS. On the one hand, the action semantics of JOOS has demonstrated that AS is capable of describing the major semantics of an OOPL, however it has not been demonstrated that AS is expressive or elegant sufficient to describe the semantics of a higher-level descriptive language like an OQL, or the semantics of complex behaviours such as state machines. This becomes one of our departure points of our work of applying AS to ALx.

```
class Controller {
   void upButPressed(){
      Elevator elevator;
      elevator = self → R4;
      send-event moveUp to elevator;
   }
   void downButPressed(){
      Elevator elevator;
      elevator = self → R4;
      send-event moveDown →  elevator;
   }
   void doorButPressed(){
      Elevator elevator;
      elevator = self → R4;
      send-event switchDoor →  elevator;
   }
   void stopButPressed(){
      Elevator elevator;
      elevator = self →  R4;
      send-event stop →  elevator;
      set [MovingButton] mbs;
      mbs = self ->* R2;
      mbs[0].delluminate();
      mbs[1].delluminate();
   }
}

class Elevator{
   boolean movingUp;
   boolean doorClosed;
   void move (boolean direction){
      if ( direction == true )
      prints("The elevator is moving up");
      if ( direction == false }
      prints("The elevator is moving down");
   }
   void stop(){
      prints{"Elevator is stopped."};
   }
}

class Door{
   void open(){
      prints{"The door is called to open"};
   }
   void close(){
      prints{"The door is called to close"};
   }
}

class MovingButton{
   boolean illuminated;
   void pressed(){}
   void delIluminate(){
      self.illuminated = false;
   }
   void illuminate(){
      self.illuminated = true;
      prints("A MovingButton illuminated.");
}

class UpButton extends MovingButton{
   void pressed(){
      Controller controller;
      controller = self → R2;
      controller.upButPressed();
      illuminate();
   }
}
```

```
class DownButton extends
MovingButton{
   void pressed(){
      Controller controller;
      controller = self → R2;
      controller.downButPressed();
      self.illuminated = true;
   }
}
class DoorSwitchButton{
   void pressed(){
      Controller controller;
      controller = self → R1;
      controller.doorButtonPressed();
   }
}

class StopButton{
   void pressed(){
      Controller controller;
      controller = self → R3;
      controller.stopButPressed();
   }
}

relation R1 DoorSwitchButton →
Controller;
relation R2 MovingButton →
Controller;
relation R3 StopButton → Controller;
relation R4 Controller → Elevator;
relation R5 Elevator → Door;

event switchDoor Controller →
Elevator;
event moveUp Controller → Elevator;
event moveDown Controller →
Elevator;
event stop Controller → Elevator;

state_machine_of Elevator{
   state StoppedWithDoorOpened{
      entry{ Door door;
            door = self→ R5;
            door.open();
      }
      exit { Door door;
             door = self→ R5;
            door.close();
      }
   }
   state StoppedWithDoorClosed{
      entry {}
      exit {}
   }
   state MovingUp{
      entry{ move(true); }
      exit{ stop(); }
   }
   state MovingDown{
      entry{ move{false}; }
      exit {stop(); }
   }

 initial_state: DoorOpened
```

```
   transition_table{
      StoppedWithDoorOpened, switchDoor,
      StoppedWithDoorClosed;
      StoppedWithDoorClosed, moveUp,
      MovingUp;
      StoppedWithDoorClosed,
      moveDown, MovingDown;
      StoppedWithDoorClosed, switchDoor,
      StoppedWithDoorOpened;
      MovingUp, Stop,
      StoppedWithDoorClosed;
      MovingDown, Stop
      StoppedWithDoorClosed;
   }
}
Main(){
   DoorSwitchButton doorSwitchButton:
   UpButton upButton
   DownButton downButon;
   StopButton stopButton;
   Controller controller;
   Elevator elevator;
   Door door;

   create-object doorSwitchButton of
   DoorSwitchButton;
   create-object upButton of UpButton;
   create-object downButton of DownButton;
   create-object stopButton of  StopButton;
   create-object controller of Controller;
   create-object elevator of Elevator;
   create-object door of Door;

   link doorSwitchButton →  controller (R1);
   link upButon →  controller (R2);
   link downButton →  controller (R2);
   link stopButton →  controller (R3);
   link controller →  elevator (R4);
   link elevator →  door (R5);

   // A staff enters the elevator
   // press the door-swiching button to
   // close the door
   doorSwitchButton.pressed();
   // then press the UpButton to move up.
   upButton.pressed();
   // after a while, he wants to move down back
   // for some reason.
   // So, he stop the elevator in the mid-way.
   stopButton.pressed();
   // Then, he presses the DownButton to
   move downwards.
   downButton.pressed();
   // after a while, he gets to the original place,
   // so he stops the elevator.
   stopButton.pressed();
   // Then, he presses the button to open the
   // door and get out.
   doorSwitchButton.pressed();
}
```

Figure 4.7: ALx code representing the visual model.

On the other hand, we can re-use, extend or modify many parts of the action semantics of JOOS in our action semantics for ALx because the two languages share much in the semantics relevant to basic expressions, imperative commands and object-oriented constructs. In this way, we can take full advantages of AS modularity and extensibility to save effort in our semantics description. This is also an opportunity to explore and check these benefits of AS in practice. As a result, we are able to focus on the semantics of those constructs that are greatly different from those of JOOS, such as class declaration and object creation/destruction, and absent in ALx, such as object query, state transition and link traversal.

To sum up, our action semantics of ALx is based on Watt's action semantics for JOOS. The reader is recommended to refer to the action semantics of JOOS when reading the semantics of ALx for better understanding.

4.7 Action Semantics of ALx

In this section, we highlight the action semantic description of some typical ALx constructs. For a complete description, the reader is referred to Appendix A, B and C.

4.7.1 Class and Class Declaration

To model ALx classes, a user-defined and composite sort **class** is specified as follows:

class = class of (class-token, type-bindings, method-bindings,
constructor, state-machine$^?$, class$^?$).

which indicates that a class is constructed from various components as follows:

- a **class-token**, which corresponds to the simple name of the class.
- a **type-bindings**, a map from token to type, where the token corresponds to the name of a field and the type to the declared type of this field.
- a **method-bindings**, a map from token to method, where a method is an abstraction encapsulating an action denoting the semantics of the method body.
- a **constructor**, a special method to be invoked during object creation.
- an optional **state-machine**, representing the state-machine behaviour of the class.

- an optional **class** is the direct super class of this class.

The sort **class** is equipped with operations to access the components of classes, including **method-bindings** _ , **class-token** _ , **type-bindings** _ , **constructor** _ , **state-machine** _ , **superclass** _ and **superclasses** _ . Their uses are straightforward: for example, the operation **method-bindings** _ is for obtaining the method-bindings component of the given class. Among them, the operation **type-bindings** _ is worth highlighting, because it returns the programmer-defined type-bindings of the given class, plus a special type-binding in which the token is "_ LinkRecord" and the type is **set**. This implies that every object of every class has an implicitly-defined field, the name of which is specially designed to be unique in the scope of an object. This treatment is used to record the links associated with the object.

The semantics of a class declaration of JOOS is that a class is constructed and then bound to a class-token which corresponds to the class name. This forms an entry of the bindings map—the scoped information. Thus, the class can be obtained based on its name. However, a class declaration of ALx will additionally allocate a cell specialized to store a list intended to memorize all objects of this class. Such lists are referred to as *object lists*. Initially, at runtime, when a class is created, the object list of this class is empty. Each time an object of this class is created, the newly created object is added to the list. Furthermore, the object list of a class is accessible because the cell holding its object list is bound to a token obtained by the operation **object-list-token** _ and specific to the class name. This semantics is described in ASD 4.1.

ASD 4.1 Class Declaration

- elaborate _ :: Class-Declaration → action [binding | storing][using current bindings | current storage].

(1) elaborate ⟦"class" I_1: Identifier "extends" I_2: Identifier "{"
F: Field-Declaration* C: Constructor-Declaration? M: Method-Declaration*
S: State-Machine-Declaration "}"⟧ =
| recursively bind the class-token of I_1 to
| the class of (the type-bindings of F, the method-bindings of M,
| the constructor of C , the state-machine of S, the class bound to the class-token of I_2).
and
| allocate a cell then
| | store an empty-list in it and
| | bind the object-list-token of the class-token of I_1 to it.

4.7.2 Objects

The sort **object** is defined as follows to model objects.

object = object of (class, variable-bindings, identity)

This means, an object consists of three components: 1) a **class**, which classifies this object. 2) a **variable-bindings**, essentially a map from field names to cells which hold values of the corresponding fields. 3) an **identity**, uniquely identifying the object, which is actually a cell allocated when the object is initialized.

Likewise, the sort **object** also provides operations to access the components of the specified object, such as **class** _ , **field-variable-bindings** _ and **identity** _ .

ASD 4.2 Field Initialization

- allocate an object of _ :: yielder [of a class] → action [storing | giving an object] [using current storage | current bindings]

(1) allocate an object of c: yielder [of a class] =
    instantiate the field-type-bindings of c and allocate an identity and initialize state of c
    then
    | give the object of (the class yielded by c, disjoint-union (the given variable-bindings#1,
    | the given variable-bindings#3), the given identity #2)

- instantiate _ :: yielder [of type-bindings] → action [storing | giving variable-bindings] [using current storage].

(2) instantiate t: yielder [of type-bindings] =
    | check (t is the empty-map) and then give the empty-map
    or
    | give t and choose a token [in the mapped-set of t] then
    | | instantiate (the given type-bindings #1 omitting the set of the given token #2) and
    | | give the given token #2 and allocate a variable initialized to the default-value of
    | | the type yielded by (the given type-bindings #1 at the given token #2)
    | then give the disjoint-union of (the given variable-bindings #1,
    | the map of the given token#2 to the given variable #3).

An object initialization, the major process in creating an object, takes the following procedure. Its action semantics is illustrated in ASD 4.2.

1. Instantiate the object's fields.
   (a) Obtain the field-type bindings of its class (the class is known).
   (b) Allocate a cell for each field.
   (c) Store the default value into the allocated cell based on the type of the field.
2. Allocate a cell to be the identity of the object being initialized.

3. Set the current state of the object to the initial state if its class has a state-machine behaviour using the auxiliary function **initialize state of** _ .

4. Construct the object using the components produced by the previous steps.

In the procedure, steps 1 to 3 can be carried out concurrently. This semantics is similar to the corresponding semantics of JOOS. Note that the special field "_ LinkRecord", mentioned in Subsection 4.7.1, is also initialized, along with other programmer-defined fields, to an empty set, being prepared to store the associated links. Furthermore, in ASD 4.2, the auxiliary function **initialize state of** _ is used to allocate a cell to store the current state of the object if this object has a state-machine behaviour. Likewise, this cell is bound to a unique token for later access.

4.7.3 Object Query

The object query mechanism enables retrieving objects of a given class from the runtime environment, usually based on a condition. The resulting object or objects are assigned to an object reference or are put into a variable of **set** type. The object query is implemented by two kinds of constructs, Object-Selection and Link-Navigation, the semantics of which are given as follows.

Object-Selection

To accomplish this object query mechanism, we have deliberately used, as mentioned in Section 4.7.1, a special cell for a class to hold its object list so as to keep a record of all its objects, including the objects of all its direct and indirect sub classes. Apart from that, the following two post-conditions of object creation and object deletion should be enforced.

- Whenever an object is created, it is put into the object list of its class and its super classes. See ASD 4.3. We use an auxiliary function with the following signature

  $$\mathsf{recursively\ add\ \_\ to\ \_} :: \mathsf{object}, \mathsf{class} \rightarrow \mathsf{action}$$

  to recursively add the newly created object to object lists of its corresponding class and **all superclasses**.

ASD 4.3 Object Creation

- execute _ :: Object-Creation → action [storing | diverging | escaping | binding] [using current bindings | current storage]

(1) execute ⟦"create-object" I_1: Identifier "of" I_2: Identifier "(" A: Arguments ")"⟧ =
| allocate an object of the class bound to the class-token of I_2 and respectively evaluate A
then
| enact the application of the constructor of the class bound to I_2 to
| the given (object, value*) and bind I_1 to the given object#1 and
| recursively add the given object#1 to class (the given object#1).

- recursively add _ to _ :: object, class → action [storing | diverging] [using current bindings | current storage]

(2) recursively add O: object to C: class =
| | give the object-list stored in the cell bound to
| | the object-list-token of (class-token C) then
| | | store concatenation (the given object-list, the list of O)
| | | to the cell bound to the object-list token of (class-token C)
| and give (superclass C)
then
| | check (the given tuple is()) and then complete
| or
| | check (not(the given tuple is()) and then recursively add O to the given class

---

- Whenever an object is deleted, it is removed from the object list of its class and its super classes. The formal description of object deletion is omitted here to save space.

Now that all objects of a class have been recorded, object selection is a matter of iterating over the objects collection, and picking out the objects which satisfy the specified condition. ASD 4.4 shows the semantics of a form of Object-Query (*select-many*) which is intended to return multiple objects. Note that, in ASD 4.4, the object being visited in each iteration is bound to the token 'selected' for immediate use in evaluating conditions.

Link Navigation

The sort **link** is defined to model links, which are instances of relations, as follows:

link = link of(relation, (object, object), identity)

This implies that a link contains its classifying relation, the connected two objects and its identity. To be simple, the sort **relation** is defined as follows:

relation = relation of (relation-token, class, class)

ASD 4.4 Object Query

- execute _ :: Object-Selection → action [storing | diverging]
  [using current bindings | current storage]

(1) execute ⟦ "select-many" I_1: Identifier "of" I_2: Identifier "(" E: Expression ")" ⟧ =
  | select instances in (the object-list stored in the cell bound to
  | the object-list-token of the class-token of I_2) satisfying E
  then
  | store the given set to the variable bound to I_1.

- select instances in _ satisfying _ :: object-list, Expression → action [giving a set | diverging] [using current bindings | current storage]

(2) select instances in I: object-list satisfying E: Expression =
  | check (I is empty-list) and then give empty-set
  or
  | check (not (I is empty-list)) and then
  | | give (head I) then bind "selected" to the given object
  | | thence
  | | | | evaluate E and select instances in (tail I) satisfying E and
  | | | | give the given object
  | | | then
  | | | | check(the given truth-value#1 is true) and then
  | | | | give disjoint-union(set of(the given object#3), the given set#2)
  | | | or
  | | | | check(not(the given truth-value#1 is true) and give the given set#2.

where the **relation-token** corresponds to the relation name; the two **classes** are ones that participate in the relation. We do not consider multiplicities of associations because multiplicities are more related to static semantics. Parallel to classes, which are produced in class declarations, relations are generated in relation declarations. Both class declarations and relation declarations of ALx, the model description parts of ALx, can completely represent textually a rigorous UML class diagram, the primary static aspect of the system.

The link navigating mechanism of ALx enables travelling from one object to another across a link and can be considered as a special kind of object query. Its fulfilment necessitates that each object records all the links connected to it. For this purpose, we have intentionally incorporated a field ("_ LinkRecord") in every object, as mentioned in Subsection 4.7.1. So, whenever a link is created, it is definitely added to both fields of the two linked objects. The formal semantics for link creation is shown in ASD 4.5. When the link is destroyed, it is removed from the fields. The formal semantics for link deletion is not illustrated here.

See ASD 4.6. The object selection based on link traversal involves the following major steps:

ASD 4.5 Link Creation

- execute _ :: Link-Creation → action [storing | diverging] [using current bindings | current storage]

(1) execute ⟦ "link" I_1: Identifier "→ " I_2: Identifier "(" I_3: Identifier ")" ⟧ =

```
| allocate a cell then
|     | give the link of ( I3, (the object stored in the cell bound to I1,
|     | the object stored in the cell bound to I2), the given cell)
|   then
|     | add the given link to the object stored in the cell bound to I1 and
|     | add the given link to the object stored in the cell bound to I2.
```

- add _ to _ :: link,object → action [storing | diverging] [using current bindings | current storage]

(2) add L:link to O : Object = give the field-variable-bindings of O
then give (the given variable-bindings at "_ LinkRecord")
then store disjoint-union of (the set stored in the given variable,
the set of L) in the given variable.

ASD 4.6 Link Navigation

- execute _ :: Link-Navigation → action [storing | diverging] [using current bindings | current storage]

(1) execute ⟦ I_1: Identifier "=" I_2: Identifier "→ *" I_3: Identifier ⟧ =

```
| give ( the object stored in the variable bound to I2) and give the relation bound to I3
then (regive and get the links from the given object#1)
then
| exhaust the linked objects of the given object#1
|       from the given set#3 related by the given relation#2
then store the given set to the variable bound to I1.
```

- exhaust the linked objects of _ from _ related by _ :: object, set, relation → action [giving a set | diverging]

(2) exhaust the linked objects of o: object from s: set related by r: relation =

```
| check (s is empty-set) and then give empty-set
or
| check (not (s is empty-set)) and then
|   | choose a link [in s] then
|   |   | exhaust the linked object of o from the intersection of (s, the set of the given link)
|   |   | and give the given link
|   | then
|   |   |   | check (the given link#2 is an instance of r) and then
|   |   |   | give disjoint-union (the set of the object
|   |   |   | linked with o by the given link#2, the given set#1)
|   |   | or
|   |   |   | check (not(the given link is an instance of r) and then give the given set#1.
```

---

1. Give all links of the given object. This is carried out by the auxiliary function

   get the links from _ :: object → action

   which returns values of the aforementioned field "_ LinkRecord" of the given object. Its formal definition is not shown here.

2. For each link, see whether it is an instance of the given relation (using a defined operation _ **is an instance of** _).

   - If so, retrieve the object connected with the given object by this link and put it into a set. This object retrieval is accomplished by the following operation

     the object linked with _ by _ :: object, link → object

   - If not, go to the next link.

   Note that this step is implemented in a recursively defined auxiliary function.

3. Bind the resulting set to a variable in the storage.

4.7.4 State Machine

To represent state machines, various sorts are defined, in particular **state-machine**, **state** and **transition-table**. Their definitions are given in ASD 4.7 and are self-explanatory. We highlight that the sort **transition-table** is actually a map that implements transition functions of state transitions, and the entry-action of a state is an abstraction encapsulating an action that is performed when the object moves into this state while the exit-action is performed as the object exits this state.

ASD 4.7 Sorts for modeling state machines

- state-machine = state-machine of (initial-state-token, transition-table, state-bindings)
- state = state of (state-token, entry-action$^{?}$, exit-action$^{?}$)
- transition-table = map [(state-token, event-token) to state-token]

According to UML 2.0, a state machine has an event pool which holds incoming events until they are dispatched; and event occurrence processing is the major behaviour of a state machine and is based on the run-to-completion assumption, interpreted as run-to-completion processing. Run-to-completion means that an event occurrence can only be processed if the processing of the previous event occurrence is fully completed. As for ALx, the semantics of this process is implemented in the construct "Event-Generation", which sends an event to an object with a behaviour of the state machine which then may trigger a state transition. ASD 4.8 shows the formal semantics of processing events, where various self-explanatory operations are defined to make the

ASD 4.8 State Machine

- execute _ :: Event-Generation → action [storing | diverging] [using current bindings | current storage]

(1) execute ⟦ "send-event" I_1: Identifier "→ " I_2: Identifier ⟧ =
give the object stored in the cell bound to I_2 then
| get the current state of the given object and regive then
| enact the application of the exit-action of the given state#1 to the given object#2
and then
| get the destination state of the given object when the event-token of I_1 and regive then
| | set the current state of the given object#2 to the given state#1 and then
| | enact the application of the entry-action of the given state#1 to the given object#2

semantic description concise. Among them, **get the destination state of _ when _** is an operation for searching the transition table and returning a destination state when an event occurs. Informally speaking, when an event happens, the exit-action, a method abstraction, of the current state is enacted and executed. Subsequently, the transition table is consulted for the target state, and the current state of the object is changed to this state. Finally the entry-action of the target state is executed. In ASD 4.8, we assume that all incoming events will definitely cause state changes, and the events that do not have effect on state machines are filtered out via static check of the program. This static check of filtering events can be also specified using AS but is ignored here to simplify the ASD.

4.8 Conclusion and Discussion

This chapter proposes a new approach to formalizing UML and presents the ASD of ALx. We do not explore describing the concurrency of UML, e.g., asynchronous calls to behaviours, co-existence of multiple *active* objects [93] each of which has its own thread, and asynchronous signal response, using communicative actions. This is owing to the following facts:

1. Each agent of AN-1, the abstraction of real computational processors, has its own local store, and no common store is provided to be readily shared by agents. It is feasible, but not trivial, to simulate a common store using an auxiliary agent that reacts to messages about allocating, changing, and inspecting its local store.

2. If we use AN-1 to cover the concurrency of UML ignoring the difficulties in modeling the common store, we consider that the most feasible solution is that each

object, whether *active* or *passive* [93], is allocated with one agent, and interactions between objects are modelled by message exchange between agents. This solution implies that the interactions between agents may be asynchronous, or synchronous. However, using AN-1 to model synchronous communication is not straightforward and needs to resort to auxiliary agents, due to AN-1 adopting the single asynchronous notion of communications.

As such, AN-1 is not suitable, or at least not elegant, for describing some notions such as light-weight processes and threads, which probably share stores and necessitate synchronous communications. In fact, this limitation of AN-1 was realized by Mosses at the beginning [65]. The newly developed AN-2 will make life easier in coping with concurrency since AN-2 allows agents to share and have global access to the storage. So, a major future work is to cover the concurrency of UML using AN-2.

At present, AS is not well supported by a suitable CASE Tool. Therefore, the static checks of the ASD of ALx is primarily by hand and not computerized, inevitably missing some errors, and it is very difficult to test the specified dynamic semantics without a proper action code interpreter. So, in order to bolster our belief in the correctness of the ASD of ALx, in the next chapter, we propose a prototype ALx-to-Java translator, underpinned by our formal semantic description of ALx, in a hope that we could observe the behaviours of xUML models through running the generated Java code.

Chapter 5

xUML-to-Java Translation

The current chapter discusses the conceptual design of an xUML-to-Java translator and then presents an implementation-neutral architecture, in which the working process of the translator consists of two sequential sub-translations: from xUML to ALx and from the ALx to Java, and a pre-defined Java library is proposed to simplify the translation. The Java code produced in each translation instance is only a part of the final Java system and must be combined with the library to form a complete system that is semantically equivalent to the original xUML model.

This chapter begins by describing the motivation of developing the xUML-to-Java translator. Then we present the architecture of the translator and explain its two-translation process. Subsequently, we put the emphasis of this chapter on the Java library.

5.1 Motivation behind Building the xUML-to-Java Translator

In Chapter 4, we formalized xUML indirectly by providing an action semantics for an intermediary action language, ALx. Our subsequent task is to check the validity of the action semantics description. It is expected that there exists an AS environment that is AS-conscious, capable of validating the syntax and static semantics of action semantics descriptions, and versatile enough to generate a runnable interpreter for a language given the ASD. In our situation, we can employ such an AS environment, if it

exists, to generate an xUML interpreter, and then feed a set of typical xUML programs to the interpreter, and observe the interpretation results against those expected. By this means, the task of experimenting on ALx can be carried out in a relatively short time and at a limited cost. Therefore, the existence of such an ideal AS tool is the key to this idea. We have conducted a thorough survey on a variety of existing AS tools in hope that we can seek a suitable one. We sincerely appreciate the tool builders for their great contribution. We think, however, that none of the tools is able to test the ASD of ALx in the expected manner.

The description of various AS tools and the reasons for excluding them are described on a per-tool basis as follows:

ASD Tools. The ASD Tools [102], the Action Semantics Description Tools, was developed by Mosses in collaboration with Arie Van Deursen, in the mid-1990s. It supports syntax-directed and textual editing, checking and interpretation of action semantics descriptions. The functionalities included significantly enhanced accuracy and productivity when writing and maintaining large specifications and are theoretically useful for students learning about the AS framework. The ASD Tools was implemented based on the ASF+SDF Meta-Environment and supported AN-1 only because when it was built, the action notation had not yet been updated to AN-2.

Even though we made it run after many attempts, ASD Tools has become obsolete owing to the lack of maintenance and updating. It is no longer workable with the current version of the Meta-Environment and the surrounding software components.

Action Environment [100]. This prototype tool, built upon Meta-Environment, provides support for a variant of action semantics characterized by using the Action Semantic Description Formalism (ASDF), which was designed specifically for providing reusable action semantics descriptions of individual language constructs.

The Action Environment can perform type checking on ASDs via checking if the actions defined in the right hand side of a semantic equation conform to the signature of the semantic function. Besides, it also provides an action interpreter, which can interpret an action semantically corresponding to a piece of code. The

interpretation result is an indication of how it terminated (normally, abruptly, or failing), the transient data it produced (if any), and the effects that the performed action has had on storage.

It appears that the Action Environment is the one that we are hunting for. The problem is that it has not been released. Another minor reason is that it does not use the original action semantic formalism but adopts a constructive action semantics, namely ASDF.

Actress system [20]. Developed by Watt's group at Glasgow University, the Actress system is aimed to interpreting action notation and compiling it into C. The interpreter is able to handle only a part of the standard actions. As such, it is vetoed.

OASIS [83]. OASIS, developed by Peter Ørbæk around 1994, is an action-semantics-based compiler generation, able to generating optimized compilers in SPARC assembler code. While, it can generate compilers for procedural, functional languages and object oriented languages only, it is unknown whether OASIS can apply to ALx which additionally incorporates features of OQLs. This problem is complicated by the fact that it is difficult for us to test the OASIS system because it is based on outdated software and a hardware platform we do not have access to. Furthermore, OASIS uses another AN based on a restricted version of the original AN. Thus, we have to forsake the idea of employing OASIS as the vehicle to test the ASD of ALx.

Recife Action Tools. Recife Action Tools, the product of the project RAT [7], contains various tools, amongst which Abaco System and Ani are the most outstanding.

Abaco [71], short for Algebraic Based Action COmpiler, is a tool set intended to help the implementation of action semantics descriptions of programming languages, based on interpreting the programming language description as a special case of order sorted algebras specifications. The system is composed of 1) a unified algebraic compiler, which translates programming language descriptions into executable programs that are able to recognize programs as specified in the source specification and produce the corresponding program actions. 2) an action pro-

cessor, which can execute the produced program actions. Similar to the Action Environment, the execution results comprise the status of termination, resultant transients and effects on the global store.

Ani or Action Notation Interpreter [69], also a RAT product, is an interpreter for actions: given an action, Ani will perform it. The outcome of the performance can be visualized through a convenient output. Notably, the current version of Ani is written in Java and has been ported to a web page as an applet, making it easily accessible. Ani is especially helpful for beginners of AS: they can compose actions, simple or composite, perform them in Ani, and then observe the output to grasp some sense of how actions behave.

So far, we have found that Recife Action Tools is the one most suiting our needs. 1) It is easily accessible: it can be easily downloaded from a well-maintained web page, and it is written in Java and thus has excellent portability, requiring not much leading time to make itself run. 2) It has the merit of relatively high running speed compared to the Action Environment and ASD Tools. This is because the former is built directly on the Java language platform, while the latter two rely on Meta-Environment, an application-level platform. 3) It is sufficient in terms of functionality: it can not only perform static checking on action semantics description but also perform actions producing a satisfying visual output.

Despite these advantages of the Recife Action Tools, in the end we still had to forsake it because we found that it is overwhelmingly time-consuming even to manipulate a small fraction of a description to pass the syntactic check. The major reason is poor error reporting.

In addition to these AS tools, there exists some other work on AS-based compiler generation. For example, in 1993, Bondorf and Palsberg [15] used the Similix system to obtain an action compiler by partial evaluation of an action interpreter. In a paper [16], Doh also proposed using partial evaluation for action transformation. Partial evaluation is employed in both works; however, to the best of our knowledge, it is quite difficult to apply to large-scale programming languages.

Now that none of the existing tools satisfies our need of testing the ALx action semantics, we discuss a new solution. The general idea of this approach is: we build a translator, which can transform xUML models into Java code, run the produced Java

code in an active JVM [57] and then observe the behaviours of the system. If the system behaves as required and as expected according to the formal specification of the xUML, it is implied that the ASD is correctly composed. It is should noted that this conclusion can be only reached if it is the case that the translation is action-semantics-based, which means that the produced Java code must be semantically equivalent to the original xUML model.

It should be emphasized that building the translation can further confirm that the AS framework has the merit of suggesting language implementation.

Some criteria are used for building the translator. Firstly, the development is required to be cost-effective, taking a reasonable amount of time and effort. Therefore, this translator is aimed to be a prototype instead of a fully-fledged system; the usability and computational efficiency of the final system are not the major concern. Secondly, code generation should be made full use of; the proportion of code generation to human coding in the development should be considerably greater. The advantages of exploring code generation are three-fold: it can lessen the development time; it can reduce human coding to avoid human errors; as desired, the code generation is a formal method because its core, the translation rules, is specified rigorously. Thirdly but crucially, the accuracy of the translator should be assured. That is, we must conform to the ASD of xUML to implement the translator so that the semantics of the produced code is semantically equivalent to the input xUML model. Fourthly, to confirm the implementation suggestivity of AS, if the ASD of ALx indicates a way of implementing a piece of semantics, we should follow this way to implement it.

5.2 Conceptual Design

The major functionality of this translator is to translate the given input xUML models into Java code in a semantics-preserving manner. Initially, the translator was intended to be incorporated into a prototype xUML tool which we were keen to build. So, for the following reasons, we decompose the overall translation process into two sequential stages (*dual-translation architecture*)—the translation from xUML models to ALx code and the one from ALx code to Java programs, rather than translating xUML models directly into Java code (*single-translation architecture*). Notably, the two sub-translations are designed to be capable of working independently.

- The dual-translation architecture can accommodate the following two groups of users of the xUML tool simultaneously: a majority of users have a preference for using visual diagrams to model the system, while some users are still keen to use textual modeling languages, like ALx, to construct xUML models. The former group of users usually draw diagrams in the xUML tool and invoke the translator to produce the Java code from the composed xUML diagrams. In contrast, the latter group of users model the system in the textual ALx and only need the translation from ALx to Java to produce Java code. Hence, the single-translation architecture cannot satisfy the latter case.

- The dual-translation architecture makes it possible that the first translation (from xUML to ALx) can be used as a model serialization mechanism for the graphical xUML models. This means, the in-memory xUML models can be persisted in the form of ALx code by being translated into ALx textual models.

- The dual-translation architecture is structurally loyal to our approach of formalizing xUML. As stated earlier, we do not specify the semantics of xUML directly but specify the intermediary ALx instead when formalizing xUML. Since developing the translator should follow the formal language specification with high fidelity, aligning the architecture of the translator to the semantic specification makes the point more convincing that AS is of strong suggestivity in language implementation.

The overall working process of this translator is illustrated in Figure 5.1. We use a typical scenario of using this translator to explain this architecture, shown as follows:

- **Composing an input xUML model**. Usually, the user authors a graphical xUML model using an xUML CASE tool and then serializes it into a textual format, like an XMI file [80]. In the phase of conceptual design, we are not concerned too much with the specific formats of the persistent xUML models.

- **Performing the xUML-to-ALx translation**. A particular xUML model is fed to the translator and the first sub-translation is started. The translator is fully aware of the structure of the persisted input model and can convert it into an in-memory ALx model. Additionally, a built-in code generator can be called to

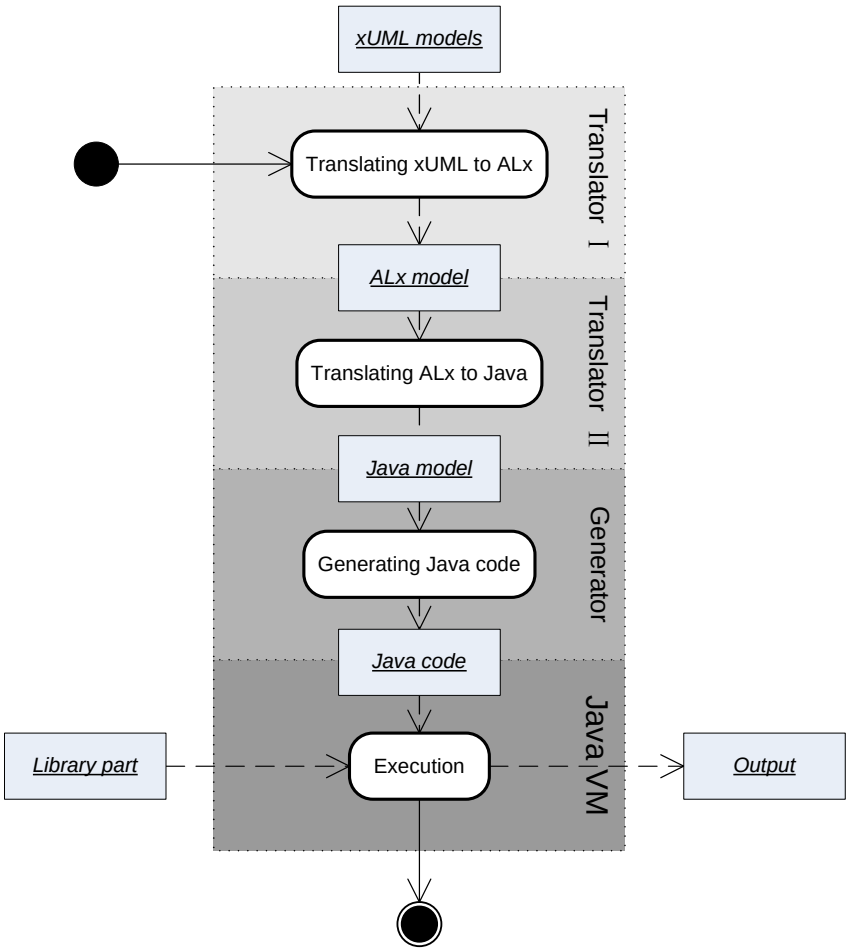


Figure 5.1: Architecture of the xUML-to-Java translator.

generate a textual copy of this in-memory ALx model if required, which is not shown in the architecture diagram. However, this code generation is normally avoided due to slowing down the whole process noticeably.

- **Performing the ALx-to-Java translation**. Through this translation, the in-memory ALx model is converted into the Java model that is also in memory. As mentioned, a group of system modelers may prefer to compose models directly in ALx instead of using graphical xUML notations. In such cases, the first translation is ignored. This requires that the translator can parse ALx code in addition to being able to parse the persistent form of xUML.

- **Generating Java code**. Since the in-memory Java model has been obtained from the previous stages, the remaining activity is to generate the code from the Java model. This is accomplished by a code-generator.

- **Executing the Java code**. In fact, executing the resulting Java is not a job of the

translator. We show it here for a complete scenario of model simulation. In this final stage, a JVM is instantiated, and commanded to load and run the resultant Java programmes. Note that in addition to loading the Java code produced by the preceding translations, a pre-compiled Java library designed by us is also loaded. We explain what the library is and why we need it next.

It is worth highlighting that if unnecessary, in theory, generating the Java code can be omitted as long as the in-memory Java model can be made consumable by the JVM directly, namely if the format of the model is made recognizable by the JVM. This treatment has the advantage of saving much time of writing texts to a persistent disk and reading them back to memory, and thus can speed up the whole process.

From the above, the final Java program of the modelled system is made up of two parts: one is the Java code yielded by the translation; the other is a pre-compiled library, which is intended to implement a major part of the xUML semantics. By so doing, the implementation of the xUML semantics is partially allocated to the library, which has two merits of: 1) greatly reducing the computation in the translation, and 2) making the definitions of the translation simpler. For convenience, we will use the *generated part* to refer to the Java code produced by the translation, and the *library part* to the library we compose.

Therefore, we focus on two tasks. For the generated part, we define the translation rules involved in the ALx-to-Java translation, namely the ALx-to-Java mapping rules, which are provided in AS-styled formalism in Appendix H. However we do not consider the implementation details in the stage of conceptual design. For the library part, we describe how the Java code implements a part of the ALx semantics.

5.3 Implementing ALx Semantics in Java

According to the architecture of the translator, the dynamic semantics of ALx are either implemented in the generated part or in the library part, or distributed in both. Assigning a part of the dynamic semantics to the library part can result in a simpler translator and a more concise generated part, and reduce the computational cost in translation.

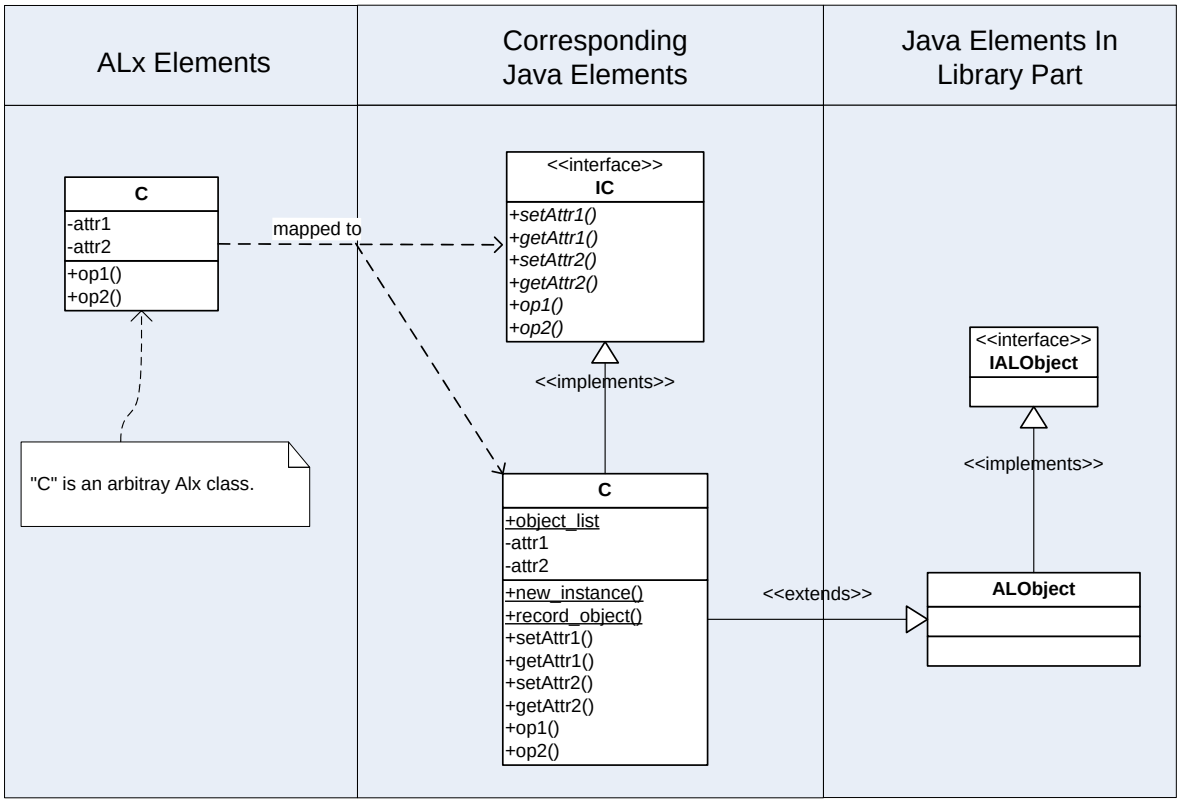


Figure 5.2: Translation overview of an arbitrary ALx class.

In this section, we proceed to detail how the generated part, coupled with the library part, implements the semantics of the ALx. In the meantime, we show that the implementation is based on the action semantics of ALx.

5.3.1 Implementing Object Query

Recording Objects of a Class

As an important part of ALx, the object query mechanism enables a query into the run-time environment to retrieve one or more object references of the given type. Usually, the returned objects are limited to those that meet the given condition. As specified in the ASD of ALx, the object query mechanism requires that in each class declaration, a special cell must be allocated in the common store to hold the object identity list of the class being declared.

Prior to explaining how to implement object identity lists, we list informally the relevant ALx-to-Java mapping rules:

- Each ALx class is mapped to a Java interface and a Java class; the Java class implements the Java interface and extends a library class called 'ALObject' (A

library class is a class defined in the library part).

- Each ALx attribute is mapped to a Java attribute.
- Each ALx method is mapped to a Java method.

These mapping rules are illustrated using Figure 5.2.

To implement the object identity list of an ALx class, a *class field* (or a *static field* [36]), called 'objectList' with type Java *list*, is incorporated in the corresponding Java class to record the references of all objects of this class. This class field is produced mechanically along with its containing Java class, and is initialized to an empty list (the default value) when the class is loaded into JVM.

The ALx semantics requires keeping the object identity list up-to-date, so the following two aspects must be assured:

- Whenever an object is created, its reference should be put into the object lists of all its ancestors.
- Whenever an object is deleted, its reference should be removed from the object lists of all its ancestors.

In Java code, the first aspect is accomplished using a method called **newinstance**, which is included in all generated Java classes that correspond to ALx classes. In the body of the method **newinstance**, there is a call to a static helper method, called 'recordObject', which is implemented as Code 5.1 shows. By this means, the task of recording object references is delegated to this helper method when creating objects. It is required that every object is created using **newinstance**. That means there exists no other way of creating objects. This requirement can be satisfied on the ground that all Object-Creation constructs are translated to Java code by the mapping rule:

translate ⟦ "create-object" I: Identifier "of" C: Identifer ⟧ = C "." "newinstance()".

Another ground is that the Java code is produced in a computerised manner, immune from human coding errors.

Note that the implementation of the helper method 'recordObject' is different if the class has a super class, shown in Code 5.2, where a line (Line 3) is added for the purpose of putting the reference of the newly-created object into the super class's object identity

Code 5.1 Implementation of **recordObject** of a class without super classes

```
...
public static void recordObject(ALObject alo){
  objectList.addObject(alo);
}
...
```

Code 5.2 Implementation of **recordObject** of a class with super classes

```
...
public static void recordObject(ALObject alo){
  objectList.addObject(alo);
  C0.recordObject(alo);  // C0 is the super class.
}
...
```

list. Evidently, this process will be recursive to a super class that has no super class any more. The translator is capable of coping with such implementational variation owing to its awareness of whether an ALx class has a super class or not at parse-time.

The second aspect regarding destroying objects is a direct opposite of the first one. Its fufillment resembles the first one conceptually, and thus is not detailed here.

Retrieving Object References of a Class

Now that the references of all objects have been recorded in object lists, an object query is merely a matter of obtaining the access to the proper object list and retrieving the qualified object references. In executing an occurrence of an object-query construct, the system will iterate each object in the object list, and only the references of those qualified objects are returned. Here we highlight only the translation rule for the conditional *select-one* in Rule 5.1, and *select-many* in Rule 5.2. (Note: in the rules, the new-line symbols and the translation of identifiers are omitted for simplicity.)

Rule 5.1 Translation rule for the select-one object query

translate ⟦ "select-one" I: Identifier "of" C: Identifer "(" E: Expression ")" ⟧ =
⟦ "Iterator⟨ALObject⟩ iterator = " C "." "objectList.iterator(); "
"while(iterator.hasNext()){"
"ALObject selected = iterator.next();"
"if(" translate E ") {" I "= selected; break;}"
"}" ⟧

Rule 5.2 Translation rule for the select-many object query

translate ⟦ "select-many" I: Identifier "of" C: Identifer "(" E: Expression ")" ⟧ =
⟦ "Iterator⟨ ALObject ⟩ iterator = " C "." "objectList.iterator(); "
"while(iterator.hasNext()){"
"ALObject selected = iterator.next();"
"if(" translate E ") {" I "." "add(selected); }"
"}" ⟧

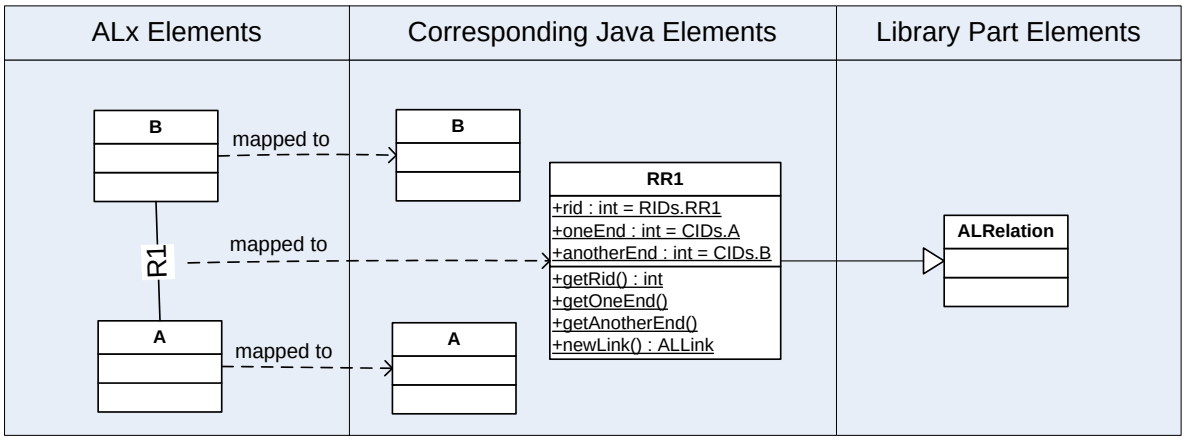


Figure 5.3: Translation overview of an arbitrary ALx relation.

5.3.2 Implementing Relations and Links

Relations are graphical artefacts in xUML models, and the relations present in ALx code are the textual counterparts of those in xUML models. In both cases, relations are model-level (M1) constructs. Links are defined to be instances of relations and thus are instance-level (M0) artefacts inhabiting the run-time environment. Being instances, like objects, links have their classifiers—relations, and can be also created and destroyed. This subsection presents the simulation of relations and links using Java.

Simulating Relations

According to the ASD of ALx, relations are specified as follows:

relation = relation of (relation-token, class, class).

This implies that a relation is composed of three components: a relation-token (i.e., the name of the relation), and two associated classes. Compared to the fully-fledged relations in the UML Specification, the relations defined in our case are quite simplified: we do not consider such concepts as association roles, association classes, navigational direction and multiplicity due to them being related to static semantics rather than run-time concerns.

See Figure 5.3 for an example. Each relation is represented by a Java class, which encompasses three static fields:

- *rid*, the unique identifier of the relation represented, which is allocated by the translator in a systematic manner.
- *oneEnd*, the value of which is essentially the 'cid' (an identifier for a class) of one of the associated classes.
- *otherEnd*. Similarly, the value of this field is the 'cid' of the other associated class.

The three fields are used to model the three components of a relation. They are made *static* for the convenience that there is no need to create an object of the relation-representing class to access these fields. In addition, because the definition of a relation cannot be changed on the fly, the values of the three fields are not changeable once initialized.

Most importantly, in each relation-representing Java class, a static method, called **newLink**, is incorporated, which is used to create links. The method body of **newLink** is implemented as Code 5.3 shows.

Code 5.3 Method of **ALLink** of relation-defining Java classes.

```
...                                                          1
public static ALLink newLink(ALObject o1, ALObject o2){     2
  ALLink temp = new ALLink(rid, o1, o2);                     3
  o1.addLink(temp);                                          4
  o2.addLink(temp);                                          5
  return temp;                                               6
}                                                            7
...                                                          8
```

Simulating Links

The run-time environment of ALx can be envisaged as a space with a network of objects, which are connected by *links*. Communications and interactions between objects, such as sending messages, signals and call to operations, are a vital part of system dynamics, and rely on *links* as the communication channels. In the case of ALx, a link consists of four ordered components: the classifying relation, two connected objects and a unique ID. In Java, we employ objects of the library class **ALLink** to simulate links, which

defines the common characteristics of all links. The code of this class is shown in Code 5.4.

Code 5.4 Library class **ALLink** is employed to simulate links.

```
package library;                                                    1
                                                                    2
public class ALLink {                                               3
  private int rid;                                                  4
  private ALObject oneEnd;                                          5
  private ALObject otherEnd;                                        6
                                                                    7
  public ALLink(int rid, ALObject oneEnd,                           8
    ALObject otherEnd){                                             9
      this.rid = rid;                                               10
      this.oneEnd = oneEnd;                                         11
      this.otherEnd = otherEnd;                                     12
  }                                                                 13
                                                                    14
  public int getItsRelation(){                                      15
      return rid;                                                   16
  }                                                                 17
                                                                    18
  public ALObject getOtherEnd(ALObject o){                          19
      if ( o == oneEnd)return otherEnd;                             20
      if ( o == otherEnd) return oneEnd;                            21
      return null;                                                  22
  }                                                                 23
                                                                    24
  public void destroy(){                                            25
      oneEnd.removeLink(this);                                      26
      otherEnd.removeLink(this);                                    27
  }                                                                 28
}                                                                   29
```

Code 5.4 explains itself. In **ALLink**, three attributes (*rid*, *oneEnd* and *otherEnd*) are used to represent the classifying relation and the two linked objects. The identities of links are not expicitly simulated by attributes because their role can be acted by the references of the link-simulating Java objects. Furthermore, the method **getOtherEnd** (See Line 19-22) is the Java counterpart of the auxiliary operation **the object other than _ of _** of the sort **link**, which is defined in the ASD of ALx and aimed to obtain the other linked object. Parallel to objects, links are created by the Link-Creation constructs, which are universally mapped to the invocation of the method **newLink** declared in relation-simulating classes.

5.3.3 Implementing Link Navigation

Link navigation, a kind of object query, enables travelling from one object along links to other reachable objects, probably via some intermediary objects. To implement link navigation, a field (named *linkList*) is embedded in the library class **ALObject**. See Line 2-3 in Code 5.5. The value of this field is a Java *list* dedicated to retaining all the references of the links which are connected to the object that owns this field. In all cases, a link connects two objects simultaneously, so the reference of a link must be present in both *link lists* of the connected objects. In addition to the field, some helper methods are defined to facilitate link navigations; their uses are suggested by name. We highlight a link-navigation-related part of **ALObject** in Code 5.5.

Code 5.5 Code fragment of the library class (ALObject) for link traversal.

```
public abstract class ALObject {                                          1
  protected LinkedList<ALLink> linkList = new LinkedList<ALLink>();       2
  ...                                                                     3
  public ALLink getLink(int rid){...}                                     4
  public ALObject getLinkedObject(int rid){...}                           5
  public void addLink(ALLink link){...}                                   6
  public void removeLink(ALLink link){...}                                7
  public LinkedList<ALLink> getLinks(int rid){...}                        8
  public LinkedList<ALObject> getLinkedObjects(int rid){...}              9
}                                                                         10
```

Parallel to object lists, link lists are also committed to be up-to-date. This requires that whenever a link is created, its reference should be added to both link lists of the two connected objects; whenever a link is deleted, its reference should be removed from the two link lists accordingly. The former is enforced by the member method **newLink** of relation-simulating Java classes (See Code 5.3 for details). The latter is done in the method **destroy** of the class **ALLink** (See Line 25-28 of Code 5.4). As far as the translation rules are concerned, every occurrence of Link-Creation in ALx code would result in an invocation of the method **newLink**, and Link-Deletion would result in invocation of the method **destroy**. The updating of link lists can be guaranteed as Java code for creating and destroying links is definitely machine-generated per the translation rules.

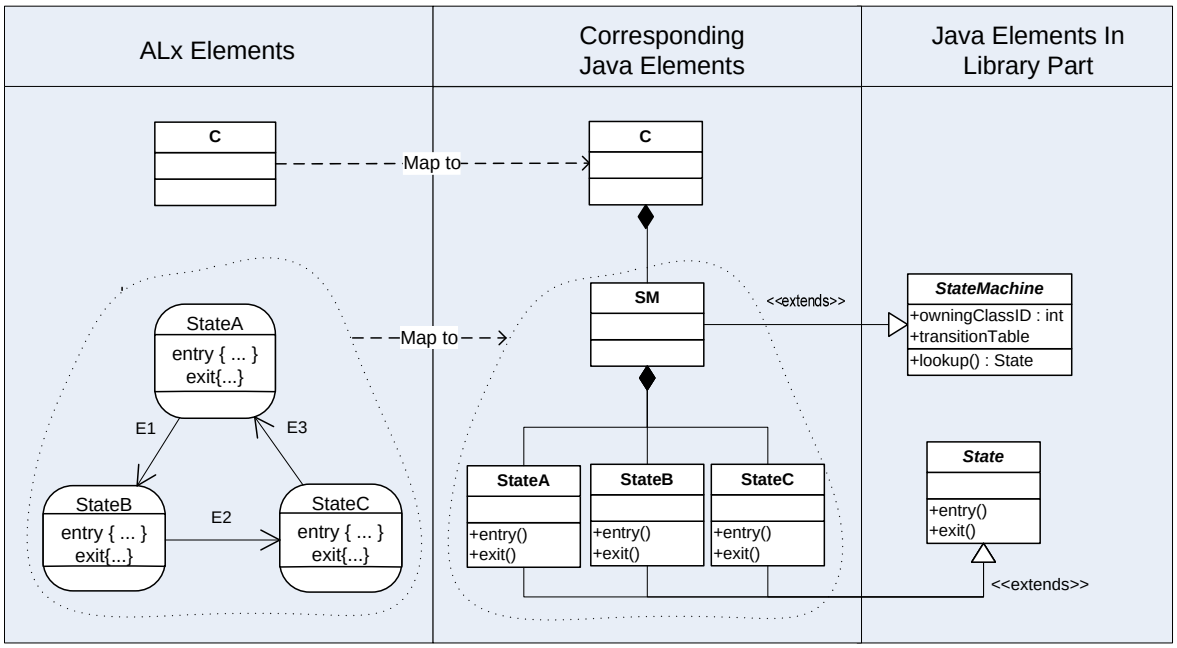


Figure 5.4: Translation overview of ALx state machines.

5.3.4 Implementing State Machines

Simulating State Machines

A state machine description belongs exclusively to a single class, so the relationship between a state machine and its owning class is compositional or part-whole. We consider that state machines are also a kind of instance. Hence, when a stateful object is created, an instance of the specified state machine should also be created and linked to this object.

We use Figure 5.4 to illustrate mappings of ALx state machines to Java classes that represent them. In this figure, on the ALx code side, 'C' is supposed to be a stateful class which incorporates a state machine as one of its behaviour. The state machine consists of various states: StateA, StateB and StateC. On the side of Java code, 'SM' is a Java class extending a library class called 'StateMachine', and it is composed by the Java class 'C'. Three Java classes (StateA, StateB and StateC), children of the library class 'State', are employed to represent the three states respectively, and are integral parts of 'SM'.

The mechanism of Java member classes is well adequate to represent the compositional relationship between a stateful class and its state machine, as well as the one between the state machine and its states. The reasons are two-fold:

- Objects of a Java member class always exist in the contexts of their containing objects. Precisely, an instance of a member class is always associated with an instance of the enclosing class. As such, an object of the member class can only be created after its enclosing object has been created. This concept is closely parallel to the compositional relation between stateful objects and their state machines.

- The code of a member class has full access to all the fields and methods, both static and non-static, of its enclosing class. This enables the entry/exit method of the member classes to invoke the member methods or access the member fields of the containing class.

Code 5.6 Code skeleton of the class 'StateMachine'

```
public abstract class StateMachine {                                    1
  public int ownedClassID;                                              2
  public HashMap<StateEventPair, State> transitionTable                 3
    = new HashMap<StateEventPair, State>();                             4
  public void addEntry(State sState, int ssid,                          5
    State tState){                                                      6
    StateEventPair sep =                                                7
      new StateEventPair(sState, ssid);                                 8
    transitionTable.put(sep, tState);                                   9
  }                                                                     10
  public int getOwnedClass(){                                           11
    return ownedClassID;                                                12
  }                                                                     13
  public void setOwnedClass(int cid){                                   14
    this.ownedClass = cid;                                              15
  }                                                                     16
  public State lookup(State state, int eid){                            17
    StateEventPair sep = new StateEventPair(state, eid);                18
    return  transitionTable.get(sep);                                   19
  }                                                                     20
}                                                                       21
```

The library class **StateMachine** is an abstract class to be extended by 'SM', where the common features of state machines are specified. See Code 5.6 for the skeleton code of this class. Every state machine has a transition table, each entry of which is made up of three elements: source state, event and destination state. Despite being called a table, the transition table is implemented using the date structure 'HashMap', the keys of which are the pairs of source states and event IDs, implemented as the objects of **StateEventPair**, and the value of which are the destination state.

In **StateMachine**, two assistant methods (**addEntry** and **lookup**) are provided for constructing and querying into the transition table. The method **addEntry** is defined to add an entry to the transition table, and it is usually called multiple times to construct a fully-fledged transition table when a state machine is created, namely when an object of 'SM' is instantiated. The code for constructing transition tables is implemented in the constructor of 'SM', automatically generated by the translator and varied depending on the state machine being represented. See Code 5.6 for details. The method **lookup** is provided for searching the transition table against the pair of source state and occurred event for the corresponding destination state.

A skeleton code of the 'Elevator' of the running example mentioned in Chapter 4 is given in Code 5.7 to further illustrate how Java member classes are used to represent state machines.

Simulating States

ALx states are one-to-one mapped to Java states, which are essentially Java classes extending the library class **State** and have two member methods: **entry** and **exit**. The mapping is intuitive, and thus it is not detailed here. It was mentioned that we use the mechanism of Java member classes to represent the compositionality between a stateful class and its state machine 'SM'. It can be also known from Figure 5.4 that such composition also exists between the state machine and its states. Thanks to Java allowing indefinite nesting of member classes, the compositionality between the state machine and its states is also represented by the mechanism of Java member classes. As mentioned, member classes are allowed to access the attributes and methods of the enclosing classes. This convenience allows the entry/exit method to manipulate the object's attributes and call methods.

Implementing Run-to-complete Process

The dynamic semantics of state machines is mainly embodied in state transitions, which are modelled by *run-to-completion* cycle that is essentially a procedure of handling events. A *run-to-completion* process involves the following steps:

1. Dispatching an event in the event queue. (Note that the object puts incoming events in the event queue first.)

Code 5.7 Java code for 'Elevator' of the running example.

```
public class Elevator extends ALObject                                    1
                      implements IElevator {                              2
  public Elevator(){                                                      3
    sm = new SM();                                                        4
    currentState =                                                        5
      ((Elevator.SM)sm).StoppedWithDoorOpened;                            6
  }                                                                       7
  public void stop() {                                                    8
    System.out.println(``The elevator is stopped.'');                     9
  }                                                                       10
    /* Member class 'SM' is for                                           11
    representing the state machine. */                                    12
  class SM extends StateMachine {                                         13
    public State StoppedWithDoorOpened                                    14
      = new StoppedWithDoorOpened();                                      15
    public State StoppedWithDoorClosed                                    16
      = new StoppedWithDoorClosed();                                      17
    ...                                                                   18
    public SM(){                                                          19
      // Construct the transition table.                                  20
      addEntry(StoppedWithDoorOpened,                                     21
               IDs.ESwitchDoor, StoppedWithDoorClosed);                   22
      addEntry(StoppedWithDoorClosed,                                     23
               IDs.EMoveUp, MovingUp);                                    24
       ...                                                                25
    }                                                                     26
    // Java member classes representing states.                           27
    class StoppedWithDoorOpened extends State{                            28
      public void entry ()  {...}                                         29
      public void exit()    {...}                                         30
    }                                                                     31
    class StoppedWithDoorClosed extends State{                            32
      public void entry() {...}                                           33
      public void exit()  {...}                                           34
    }                                                                     35
     ...                                                                  36
  }                                                                       37
}                                                                         38
```

2. Triggering a state transition if the dispatched event is not filtered off by a condition.

3. The object moves out of the current state. Just before this happens, the exit activity is called into execution until completed.

4. The object moves into the target state, and the entry activity of this target state is invoked and executed to finish.

This process is emphasized by the words *run-to-complete* as the next event cannot be dispatched to be processed if the process of the preceding event is not completed.

In fact, the run-to-complete process specified in the standard UML specification is far more complicated. To be simple, ALx adopts a state machine simplified both in static structure and in dynamic semantics, where not considered are parallelism, multi-threading and asynchrony of all behaviours, including that of operation invocation of classes and that of state machines. This is to say, the execution of the whole program takes place in a single main thread; at any moment, only one behaviour is performed, and it is either a method or a run-to-complete process. Therefore, the event queue of an object, which enables the concurrency of accepting events and handling events, could be ignored, so could the event-dispatching mechanism. As such, any event targeting an object will be tackled as soon as it occurs.

Code 5.8 Code skeleton of the method **stateTransimitted**

```
...
public void stateTransmitted(int eid){
  // check whether this is a stateful object.
  if (this.currentState == null) return;
  if (this.sm == null) return;
  //else, this is a stateful object.
  this.currentState.exit();
  // transit to another state
  // find the target state
  State tState= this.sm.lookup(this.currentState, eid);
  // execute the entry method in the target state.
  tState.entry();
  // Current state is set to target state.
  this.currentState = tState;
}
...
```

Now we proceed to illustrate how to implement a state transition in Java. The translation rule for Event-Generation is shown as follows:

$$\begin{aligned}&\mathsf{translate}\ [\![\text{“send-event”}\ \ I_1\text{: Identifier}\ \text{“}\rightarrow\text{”}\ \ I_2\text{: Identifier}]\!] =\\ &\qquad [\![\ I_2\ \text{“.”}\ \ \text{“stateTransmitted”}\ \ \text{“(”}\ \text{“EIDs”}\ \text{“.”}\ I_1\ \ \text{“)”}\ ]\!]\end{aligned}$$

This rule indicates that an event generation corresponds to an invocation of the method **stateTransmitted()** within the context of the target object. The method **stateTransmitted()** is defined in the class **ALObject**. See Code 5.8 for the implementation of this method. First, this method checks whether the target object is a stateful object

by checking if it has an associated state machine. If not, it returns. If it is, the method would invoke the exit activity of this object. Then, it consults the transition table for the destination state. Subsequently, the entry activity of the destination state is called. Finally, it updates the current state of this object to the destination state.

5.4 Discussion and Conclusion

Because there is no AS tool capable of validating the ALx ASD, and versatile enough to generate a runnable interpreter for ALx, we turn to implementing an xUML-to-Java translator following the ASD of ALx to give some assurance to the ALx ASD. The implementation-neutral architecture of the translator is proposed in this chapter, which is characterized by a two-translation process and allocating a significant part of the dynamic semantics implementation to a Java library. In addition, the implementation of some featured semantics of xUML or ALx, such as object-query mechanism, relation and link mechanism, state machines and the run-to-complete process of state machines, are detailed in this chapter.

We do not investigate multi-inheritance of classes because Java does not inherently support it. However, this does not mean that implementing multi-inheritance in Java is impossible. As far as we know, *design pattern* [33] can be applied to tackle this problem; for instance, in [88], the author proposes a pattern to avoid multi-inheritance in Java implementation. However, such implementation would make inheritance hierarchy and class structure of system extremely complicated and thus undermine our philosophy of mapping ALx to Java in a straightforward manner.

Chapter 6

Model-Oriented xUML-to-Java Translation

In this chapter, we introduce some key features of MDA, followed by justification of employing MDA as the approach to implement the xUML-to-Java translator. Subsequently, we investigate the applicability of UML, one pillar of MDA, to specify the static aspects of programming languages. Finally, we introduce the Eclipse MDA environment and describe how to use it to implement the conceptual design proposed in Chapter 5.

6.1 Key Features of MDA

Object-Oriented Programming (*OOP*), the most popular programming paradigm now, has advantages over structured programming both in modeling and coding the target systems. However it still focuses overly on the coding level. The necessity of raising the abstraction level has been widely realized by the community of software engineering. The Model Driven Architecture (MDA) [62, 49] is the product of this concern and is a framework underlying a new paradigm for software development, specified in the MDA Guide [62] in late 2001 by *Object Management Group* (*OMG*).

The emergence of MDA causes an important paradigm shift in software system construction — the move from object and component technology to model technology. The major activities in MDA are concentrated on modeling the system at different abstraction levels, and achieving the final system through model transformation.

A typical scenario of system development in MDA begins with gathering a set

of requirements. Developers then compose a system model which satisfies those requirements. This initial model captures the requirements at a fairly high abstraction level, without committing to a specific technology platform, and is called a platform-independent-model (PIM) [62, 49].

Then the PIM is transformed into a platform-specific model (PSM) [62, 49], guided by some well-defined rules which are usually specific to the selected implementation platform. The transformation lowers the level of abstraction by introducing implementation-specific elements into PIM; it is usually facilitated by a model-driven development environment and may be manual, semi-manual or automated. Note that the term 'platform' in the scope of MDA refers not only to a certain operating system but also to a language-based platform such as Java or C++, J2EE or .NET, HTML or XML, IBM DB or Microsoft SQL database, EJB [63] or Corba [72], etc.

Subsequently, a code model is usually produced. Although the resultant PSM is closer to the final code, it is still too abstract for compilation in a given language. For instance, it is preferable that a Java class in a PSM is enriched by providing access methods to all its attributes in the code level. So, in MDA, there usually exists a code model as specific or abstract as the textual code, which is transformed from the PSM based on pre-defined transformation rules. Finally, the textual code for the system is obtained from the code model.

It can be seen that, in MDA, code is produced from models rather than being written directly by programmers. As the system changes, developers only need to modify the model, which is normally more intelligible and manipulatable compared to the code because of its higher abstraction, and then the MDA environment synchronizes the code with the changed model. The model is always up-to-date and useful, no longer discarded at the outset of coding, but becoming the focus of development effort.

OMG are still working on the standardization of model transformation. One benefit of its work is that transformation and code generation could be fulfilled automatically by a computer. Even if it is not fully automated, the transformation is the exclusive task of those virtuosos in computer science. So software developers will solely concentrate on modeling the highly abstract PIM.

The above scenario of developing a system shows the following features of MDA:

- MDA raises the abstraction level to the model level. Increasing the abstraction

level is routine in software development. For example, the shift from assembly language programming to C programming is an upgrade from a machine-specific level to a machine-unconscious one. Every time the abstraction level is enhanced, a revolution is about to take place in the efficiency of software development.

- Modeling is the focus of the development activities, and the produced models become the most valuable asset of the development efforts. Since the model is completely consistent with the code, now it can be respected as a trusted documentation, and more importantly, can be reused to generate code in newly-emerged technologies, saving lots of efforts and reducing the time to market.

- The complicated transformation rules are pre-defined by a few experts who are knowledgeable in the platform-specific details, and these rules can be published to be shared for different projects. This means that the application developers need not know too much about the implementation techniques and can focus more on business logics, which lowers the threshold of software development. Model transformations are usually accomplished automatically in MDA environments. So, the advantages of MDA are more obvious when the model transformations involved are bulky.

6.2 Adopting MDA as Implementation Approach

Adopting MDA as an implementation approach is determined by the features of the xUML-to-Java translator under development. The working process of the translator can be viewed as a pipeline model, in which there are three highly coarse-grained software components–the first translator, the second translator and the code generator. For simplicity, we use TR1 to represent the first translator, and TR2 the second translator. In parallel, T1 is used to represent the first translation, performed by TR1, and T2 to the second translation, performed by TR2. Each component takes the output of the preceding one as the input except for TR1 which has no precursor.

TR1 shall be equipped with the knowledge of xUML and ALx, as well as translation rules. More specifically, TR1 must be provided with the syntactic definitions both of xUML and ALx and their translation rules so that it is able to recognize the concerned languages and then carry out translations following the rules. All xUML models must

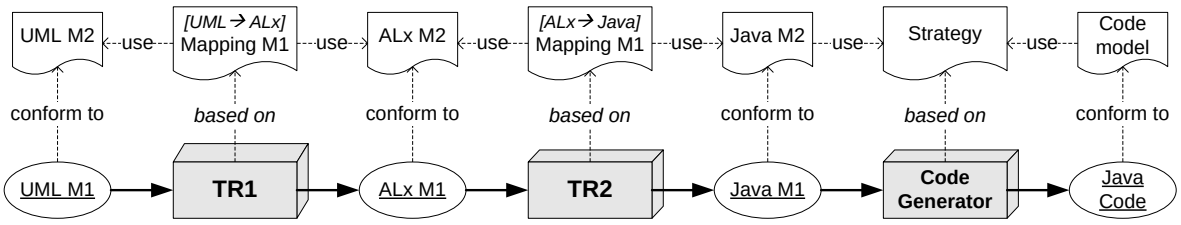


Figure 6.1: Models and metamodels required by the translator

conform to the syntactic definition of xUML to be valid, so the former are instances of the latter. In this sense, from the viewpoint of the four-layer metamodeling architecture mentioned in Chapter 2, we can refer to the syntax definition of xUML as an xUML metamodel. For the same reason, the syntax definition of ALx can be called an ALx metamodel. Furthermore, we regard the translation rules as a mapping model because each specific translation occurrence is an instance of the translation rules. To sum up, it is the prerequisite that TR1 must rely on the xUML metamodel, ALx metamodel and their mapping model to carry out translations from xUML models to ALx models.

The same also applies to TR2, which must be equipped with the ALx metamodel, Java metamodel and their mapping model to perform the translations from ALx models to Java metamodels.

The code generator needs to recognize the Java metamodel and shall be given a strategy for how to generate code. The strategy is characterised differently depending on the specific implementation of the code generator. For example, some code generators consider the input model as a tree and then produce code through visiting the nodes residing in the tree, so the strategy of code generator is specified in the tree visitor. For another example, some code generators adopt the notion of templates to carry out their duty, which is principally characterized by pattern match and has the advantage of better usability.

Now we summarize the knowledge, mainly metamodels and mapping models, required by the constituent components of the translator and their dependencies in Figure 6.1, and we sum up some features of the overall translation system as follows.

- The system is densely populated by models and metamodels; thus preparing the metamodel is a primary task in system building.

- The data pertaining to the translator can be classified into two hierarchical layers: the data in the lower layer are instances of those in the upper layer, as is evident

because models are instances of the corresponding metamodels.

- The major tasks of this system are model transformations. From the conceptual design, model translations occur pervasively in the whole working process.
- The system involves code generations.

With the features of the xUML-to-Java translator in mind, it is fairly natural to consider MDA as a potential approach to prototype the translator. Recall the notion of MDA, which preaches that systems are achieved through model transformations, model evolutions, and which also corroborates four-layer hierarchical classification of information. In this sense, MDA is intended for this kind of system like the translator, which is full of models, metamodels and model transformations.

MDA seems to be highly appropriate to implementing the translator, but we still need to answer the following three questions before making the final decision of adopting MDA as the implementation approach.

- It is very important to find out whether MDA is capable enough to describe the abstract syntax of programming languages. Namely, it is necessary to investigate whether or not UML is expressively sufficient to describe the languages involved, because UML is the adopted modeling language of MDA.
- It is necessary to investigate how mapping models are specified in MDA.
- We need to know how MDA is supported by tools, including model editors, code generators, model transformation engines, etc. Note that MDA is just a conceptual approach and thus it is not practical until some tools realize it.

6.3 Applicability of MDA to Programming Languages

6.3.1 Representing AST in UML

A parallel

We consider that there exists a strong parallel between MDA's four-layer information-representing hierarchy and the conventional way of representing the static aspect of programming languages using BNF-like meta-grammars. As mentioned in Chapter 2,

the four-layer information-representing hierarchy is the corner stone of MDA, where information appears as model elements that are classified into four hierarchical layers based on the *instance-class* notion. For instance, the metamodel of UML, the specification of UML, resides in Meta-model (M2) layer, while UML models that are the specific instances of the UML language are located in Model (M1) layer. In addition to UML, various other modeling languages also exist in the M2 layer, such as Common Warehouse Metamodel (CWM) [75], XML Metamodel Interchange (XMI) [80] and Business Process Definition Metamodel (BPDM) [81].

The number of M2-layer metamodels is still increasing; this gives a rise to the need of a global Abstract Syntax Tree (AST) that can characterize all metamodels. Therefore, OMG uses the Meta-Object Facility (MOF) to address this issue. It is claimed by OMG that such integration of metamodels would eliminate the issue of incompatibility between modeling tools, enhancing tool cooperation [77].

MOF is featured as follows. 1) The MOF is the only entity in the M3 layer. 2) MOF is self-describing. This is why there are only four layers in the framework because no upper layer is necessitated to describe MOF. 3) MOF, like UML, also employs the object-oriented paradigm to view the system ('system' here refers to metamodels). 4) MOF reuses the core of UML as its core.

The characteristics of the four-layer architecture strongly suggest a parallel between itself and the traditional formalization of programming languages. The M3 layer corresponds to the meta-grammar level. For instance, BNF is a grammar of grammar and also defines itself; the M2 layer corresponds to the grammar level; the M1 layer corresponds to the code level; the M0 level corresponds to a dynamic execution of the given program.

Translating grammars to UML representations

It is necessary to investigate whether UML is adequate for specifying the grammars of programming languages; i.e., is MOF sufficient for describing grammars for programming languages? Since MOF incorporates the core of UML as its core—the part for describing class diagrams, we only need to look into whether UML class diagrams are expressive enough for describing grammars of programming languages.

The grammar of programming languages comprises terminals $\mathcal{T}$, non-terminals $\mathcal{N}$,

and production rules of the form

$$\mathcal{N} \rightarrow \quad s_1 \mid s_2 \mid \ldots \mid s_n$$

where $\mathcal{N}$ is a non-terminal and s_1, s_2, ..., s_n denotes strings of grammar symbols, each of which is a sequence of terminals, non-terminals or their blend. Terminal symbols either describe aspects of the concrete syntax like keywords, or have a dedicated semantic meaning like operators. Some of the non-terminals abstract from certain tokens of the source program like identifiers or numbers.

For the convenience of illustration, we distinguish non-terminals, based on the right-hand side of their defining rules, as follows:

1. Simple non-terminals. A *simple non-terminal* has multiple possible forms, however each of the forms is simply a terminal. The most outstanding simple non-terminals are operators of a language.

2. Single-formed non-terminals. A *single-formed non-terminal* has only one form, which may be purely a terminal, or a sequence of terminals and non-terminals.

3. Multi-formed non-terminals. A *multi-formed non-terminal* has more than one form. There is no special requirement for its forms, one of which may be as simple as a terminal or a non-terminal, or a sequence of terminals and non-terminals.

Now we extract a simplified fraction of ALx grammar (shown in Syntax 6.1) to illustrate this idea of classification. In the grammar, 'InfixOp' is a simple non-terminal; 'Exp' and 'Stmt' are two multi-formed non-terminals; 'BlockStmt' is a single-formed non-terminal.

Syntax 6.1 A simplified fraction of ALx grammar.

```
InfixOp    →  "+"  |  "−"  |  ">"  |  "<"  |  ...

Exp        →  IntLiteral  |  "self"  |  Exp InfixOp Exp  |  ...

Stmt       →  ";"  |  BlockStmt  |  "return" Exp? ";"  |
              "if"  "("  Exp  ")"  Stmt  "else"  Stmt  |  ...

BlockStmt  →  "{"  Stmt*  "}"
```

The right-hand side of the rule of a multi-formed non-terminal can be rewritten so that all its forms are purely non-terminals. This is achieved through creating a new non-

terminal, preferably meaningfully named, to substitute for each terminal or composite form within the right-hand side of the rule. Meanwhile, a new grammar rule is added to define each new substituting non-terminal. All newly-created non-terminals are single-formed because they each have only the form they have replaced in the original rule. For instance, the multi-formed non-terminal 'Exp' can be rewritten to

```
Exp    → IntLiteral | SelfExp | BinExp | ...
BinExp → Exp InfixOp Exp
SelfExp → "self",
```

where all forms of the multi-formed non-terminal 'Exp' are single-elemented, either a terminal or a single-formed non-terminal. This kind of substitution can be also applied to 'Stmt'. Ultimately we obtain a new equivalent grammar as shown in Syntax 6.2.

Syntax 6.2 The grammar resulted from rewriting the multi-formed non-terminals of Syntax 6.1.

```
(1) InfixOp    → "+" | "−" | ">" | "<" | ...
(2) Exp        → IntLiteral | SelfExp | BinExp | ...
(3) SelfExp    → "self".
(4) BinExp     → Exp InfixOp Exp.
(5) Stmt       → EmptyStmt | BlockStmt | ReturnStmt | IfStmt | ...
(6) EmptyStmt  → ";"
(7) ReturnStmt → "return" Exp? ";"
(8) IfStmt     → "if" "(" Exp ")" Stmt "else" Stmt
(9) BlockStmt  → "{" Stmt* "}"
```

After a series of substitutions, no multi-formed non-terminals have composite forms. These kinds of multi-formed non-terminal free of composite forms are referred to as *abstraction non-terminals* thereafter. Since every single multi-formed non-terminal can be rewritten to an abstraction one, in an arbitrary grammar there are only three kinds of non-terminals: simple, single-formed and abstraction.

We propose a general idea of using UML to represent a grammar rule that has been pre-processed by substitution. The idea is specified in terms of three transformation rules, each for a type of non-terminal. The transformation of a rule always begins with observing the type of right-hand-side non-terminal of the rule:

Rule 1 If it is a simple non-terminal, then an enumeration type is created to represent it: the type name is usually the same as the name of the non-terminal if there is name collision; the enumeration literals are the terminals that occur in the forms on the right-hand side.

Rule 2 If it is a single-formed non-terminal, then a class is created to represent it: the class name is usually the same as the name of the non-terminal, say C_0. Next, each component occurring in the only form of this non-terminal must fall into one of the following cases:

Case 1 If it is a terminal, do nothing.

Case 2 If it is a simple non-terminal, we use an attribute of C_0 to represent it: the attribute name is usually the same as the name of this non-terminal and its type is the enumeration type that denotes the non-terminal.

Case 3 If it is not a simple non-terminal, and, we assume, has been denoted by a class C_1, then create a relation between C_0 and C_1. It is not excluded that this non-terminal may be modified by a cardinal symbol like the optional ('?'), the obligatory ('+') or the multiple ('$*$'). Under this circumstance, the relation connecting C_0 and C_1 should be complemented by an appropriate multiplicity at the end of C_1.

Rule 3 If it is an abstraction non-terminal, firstly an abstract class is created to represent it. Secondly, all classes representing non-terminals on the right-hand side inherit this class.

Now we follow the raised rules to systematically transform into UML classes the grammar rules shown in Syntax 6.2. The reader is recommended to refer to Figure 6.2 when reading the following explanation.

- For Production (1), where 'InfixOp' is a simple non-terminal, Rule 1 is applied, leading to a UML enumeration type illustrated in (a) of Figure 6.2.

- For Production (2), where 'Exp' is an abstraction non-terminal, Rule 3 is applied, leading to a homonymous abstract class, which is made the super class of the classes that represent terminals or non-terminals in the forms, as shown in (b) of Figure 6.2.

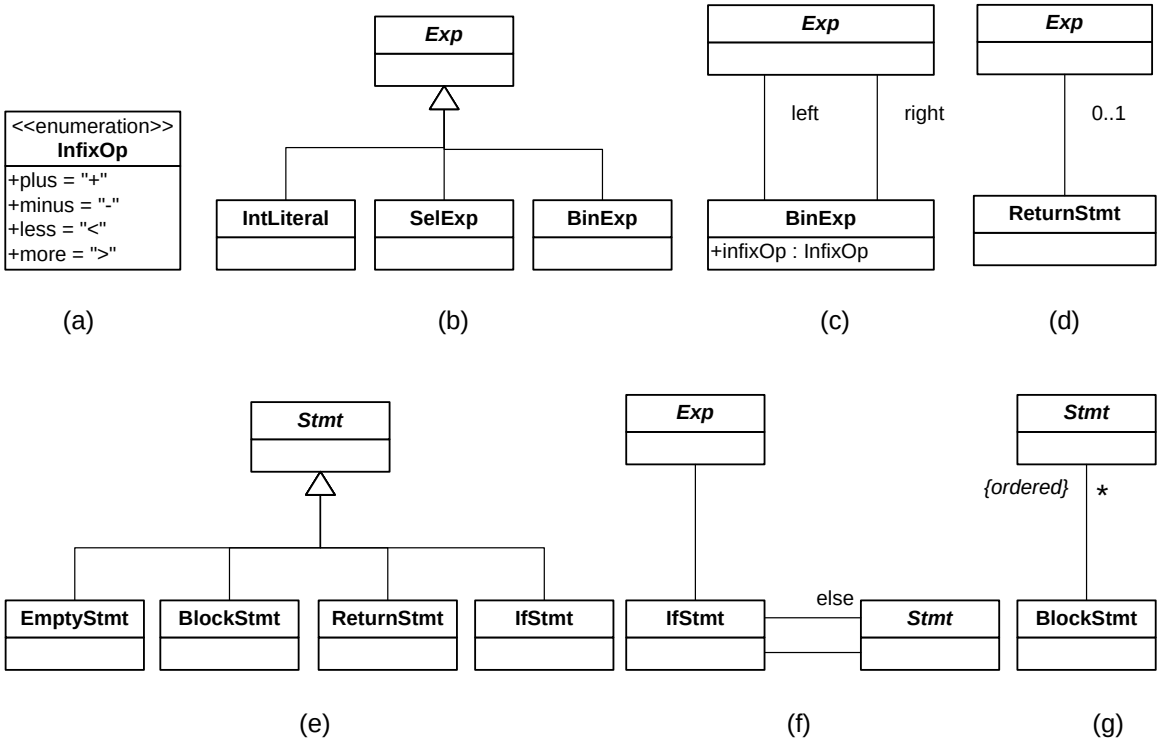


Figure 6.2: Illustration of transforming context-free grammars into UML classes.

- For Production (3), where 'SelfExp' is a single-formed non-terminal, Rule 2 is applied, resulting in a homonymous class, shown in (b) of Figure 6.2. Besides, the only component in its sole form is a terminal, so Case 1 is selected, doing nothing.

- For Production (4), where 'BinExp' is a single-formed non-terminal, Rule 2 is applied, resulting in a homonymous class 'BinExp'. Furthermore, for the first component in the form, which is not a simple non-terminal, Case 2 is selected, leading to a relation between the class 'Exp' and 'BinExp'. For the second component, a simple non-terminal, Case 1 is selected then an attribute is used to represent it. For the third component, a similar relation is also produced. Note that even though the first component and second component are the same non-terminals, their positions in the form are also significant. Take this case as an example: the different evaluation order makes their position unchangeable. So, this signification of position must be reflected in the class diagram. This depends on implementation of the class diagram. Here, we use two relations. See (c) of Figure 6.2 for illustration.

- For Production (5), where 'Stmt' is an another abstraction non-terminal, this corresponds to classes in (e) of Figure 6.2.

- For Production (6), where 'EmptyStmt' is a single-formed non-terminal with a simple terminal form, this corresponds to the class 'EmptyStmt', as shown in (e) of Figure 6.2.

- In Production (7), 'ReturnStmt' is a single-formed non-terminal; its form is a component modified by the cardinal symbol ('?'), so this is reflected in the class diagram by a corresponding multiplicity of the association end of the class 'Exp', as shown in (d) in Figure 6.2.

- For Production (8), the corresponding class diagram is shown in (f) of Figure 6.2, being conscious of the significance of positions of the same non-terminals.

- For Production (9), the corresponding class diagram is shown in (g) of Figure 6.2, where a string 'ordered', a predefined constraint keyword in UML, is employed to indicate the position significance.

We have used these rules to manually translate ALx and MiniJava abstract syntax descriptions (Appendix A, H) into their metamodels in UML (Appendix E, F). However, the transformation could be made by computer through a three-pass scan. In the first pass, a BNF-aware parser identifies the type of occurred non-terminals and marks them. In the second pass, substitution is performed to convert some multi-formed non-terminals to abstraction non-terminals. Finally, the transformation based on the rules is conducted to generate UML class diagrams, most likely in a textual format.

Another point is needed to be considered: it is well known that programming languages, in most cases, are context-sensitive, involving context-sensitive constraints or static semantics. So in the next section, we discuss how UML, the pillar of MDA, can be used to represent it.

6.3.2 Representing Static Semantics

The context-sensitive languages are referred to as Type-1 languages in the Chomsky hierarchy [24], and the static aspects of such languages are usually defined in the manner of context-free grammars augmented by context-sensitive constraints. Therefore, in addition to the previous investigation about using UML representing context-free grammars, it is essential to study how MDA specifies context-sensitive constraints.

The notion of context-free grammars only loosely confines programming languages. In a context-free grammar one defines syntactical categories such as expressions, commands and assignments for expressing how the sentences of the programming language are formed. Nevertheless, it is likely that some sentences may conform to the context-free grammar but cannot be regarded as well-formed as a result of a failure to meet some contextual requirements. For one instance, in a strongly-typed programming language, variables can only be used after they are declared explicitly previously in their scopes. For another, in an assignment statement of an imperative language, the type of the value to be yielded by evaluation of the right-hand-side expression is normally required to be consistent with the type of the variable on the left-hand side in terms of type equivalence. Constraints of this kind are referred to as the static semantics of a programming language and cannot be captured in an EBNF-like grammar.

To specify the static semantics of programming languages, various well-tested approaches have been around for ages, among which the notion of *attribute grammars* [95] is the most outstanding and widely used. When specifying an attribute grammar, one always begins with constructing a BNF grammar, which generates the base syntactic sets. Then, attributes with type and value are incorporated into those base syntactic sets. The incorporated attributes can fall into two categories, *synthesized* and *inherited*. Viewing a program or sentence as a parse tree, the values of the *synthesized attributes* of a node are used in evaluating attributes of its *parent*. In contrast, the values of the *inherited attributes* of a node are used in evaluating attributes of its *children*. Subsequently, the evaluation rules of these attributes are provided to figure out values of attributes of nodes in the parse tree of a particular sentence or program. Finally, some conditions upon these attributes are given. The type check would fail if one or more conditions attached to the nodes in the parse tree obtained as a result of parsing a program/sentence are tested false.

Attribute grammars are widely employed in generating parsers capable of type-checking [48, 30, 50]. To show an example of attribute grammars, we use the extended Syntax 6.2 as the context-free part, on the base of which the context-sensitive information is specified.

Firstly, we incorporate a variety of attributes, either synthesized or inherited, into some non-terminals. For details, see Figure 6.3. In this example, all attributes are named 'type' (under real circumstances, this is not limited to 'type'), which is evaluated

Syntactic sets	Synthesized	Inherited
Int	*type*	
Bool	*type*	
VarDecl		*type*
Var	*type*	
Exp		*type*
BinExp		*type*
Type	*type*	

Figure 6.3: Synthesized and inherited attributes

to an element in the set {'bool', 'int', 'undefined'}, in which 'undefined' denotes the circumstances that a variable is undeclared, or other unknown situations. That is, the attribute 'type' is an enumeration type which has three literals, valued 'bool', 'int' and 'undefined'.

Syntax 6.3 An example attribute grammar.

(1) i: Int $\rightarrow$ [0-9] | [1-9][0-9]$^{+}$. $\{i.\text{type} = \text{'int'}\}$

(2) b: Bool $\rightarrow$ "true" | "false". $\{b.\text{type} = \text{'bool'}\}$

(3) InfixOp $\rightarrow$ "+" | "−" | ">" | "<".

(4) d: VarDec $\rightarrow$ t:Type v:Var ";". $\{v.\text{type} = t.\text{type}, d.\text{type} = t.\text{type}\}$

(5) e: Exp $\rightarrow$ i: Int $\{ e.\text{type} = i.\text{type} \}$ |
b: Bool $\{ e.\text{type} = b.\text{type}\}$ |
v: Var $\{ e.\text{type} = v.\text{type}\}$ |
x: BinExp $\{ e.\text{type} = x.\text{type}\}$.

(6) x: BinExp $\rightarrow$ e_1: Exp o: InfixOp e_2: Exp $\{x.\text{type} = e_1.\text{type}\}$.
$[\![$ if o = '+' or '-' then $e_1.\text{type} == e_2.\text{type} ==$ 'int'
else $e_1.\text{type} == e_2.\text{type} ==$ 'bool' $]\!]^{\text{cond1}}$.

(7) Stmt $\rightarrow$ EmptyStmt | BlockStmt | Assignment | If-Stmt | VarDec.

(8) BlockStmt $\rightarrow$ "{" Stmt* "}".

(9) EmptyStmt $\rightarrow$ ";".

(10) Assignment $\rightarrow$ v: Var "=" e: Exp ";". $[\![v.\text{type} == e.\text{type}]\!]^{\text{cond2}}$

(11) If-Stmt $\rightarrow$ "if" "(" e: Exp ")" Stmt "else" Stmt. $[\![e.\text{type} == \text{'bool'}]\!]^{\text{cond3}}$

(12) t: Type $\rightarrow$ "boolean" $\{t.\text{type} = \text{'bool'}\}$ | "int" $\{t.\text{type} = \text{'int'}\}$.

Secondly we provide various evaluation rules and conditions, embedded in the production rules, shown in Syntax 6.3. All evaluation rules, enclosed in the grammar by curly braces ('{}'), are self-evident. Three conditions are involved in this attribute

grammar, enclosed by double square brackets ('⟦ ⟧'), and they are explained as follows:

- The *cond1* in Rule (6) indicates that the types of two sub-expressions must be equivalent and be consistent with the operator.
- The *cond2* in Rule (10) indicates that the type of the variable on the left must be same as that of the expression on the right.
- The *cond3* in Rule (11) indicates that condition expressions must be bool-typed.

As mentioned, pure UML class diagrams, though context-free-grammar complete, are not fine enough to specify context-sensitive grammars. We consider that the UML class diagram would become expressive enough to specify context-sensitive grammars if it is augmented by OCL expressions. This consideration stems from the parallel: analogous to attribute grammars complementing EBNF to specify static semantics, the OCL that is created to provide well-formity to UML models can also represent context-sensitive information of programming languages.

Now we proceed to discuss how attribute grammars can be translated into UML class diagrams augmented by OCL expressions.

First, transform the context-free aspect of the attribute grammar into a UML class diagram based on the transformation rules specified in Subsection 6.3.1. In this step, the context-sensitive information in the attribute grammar, including attributes, evaluation rules and conditions, is transparent to the transformer.

Second, represent attributes of non-terminals as UML attributes of the corresponding UML classes/types. More specifically, if the concerned attribute grammar assigns an attribute to a non-terminal, whether synthesized or inherited, then a corresponding attribute is incorporated into the corresponding UML class/type.

Third, the evaluation rules in the attribute grammar are translated to OCL expressions. OCL is adequate to represent the evaluation rules due to two facts: it can specify the initial values of attribute, and it can specify the value of the derived attributes.

Fourth, conditions are expressed by OCL invariants. By definition, OCL invariants are the conditions expressed in OCL that must be hold for all instances of the targeted UML model.

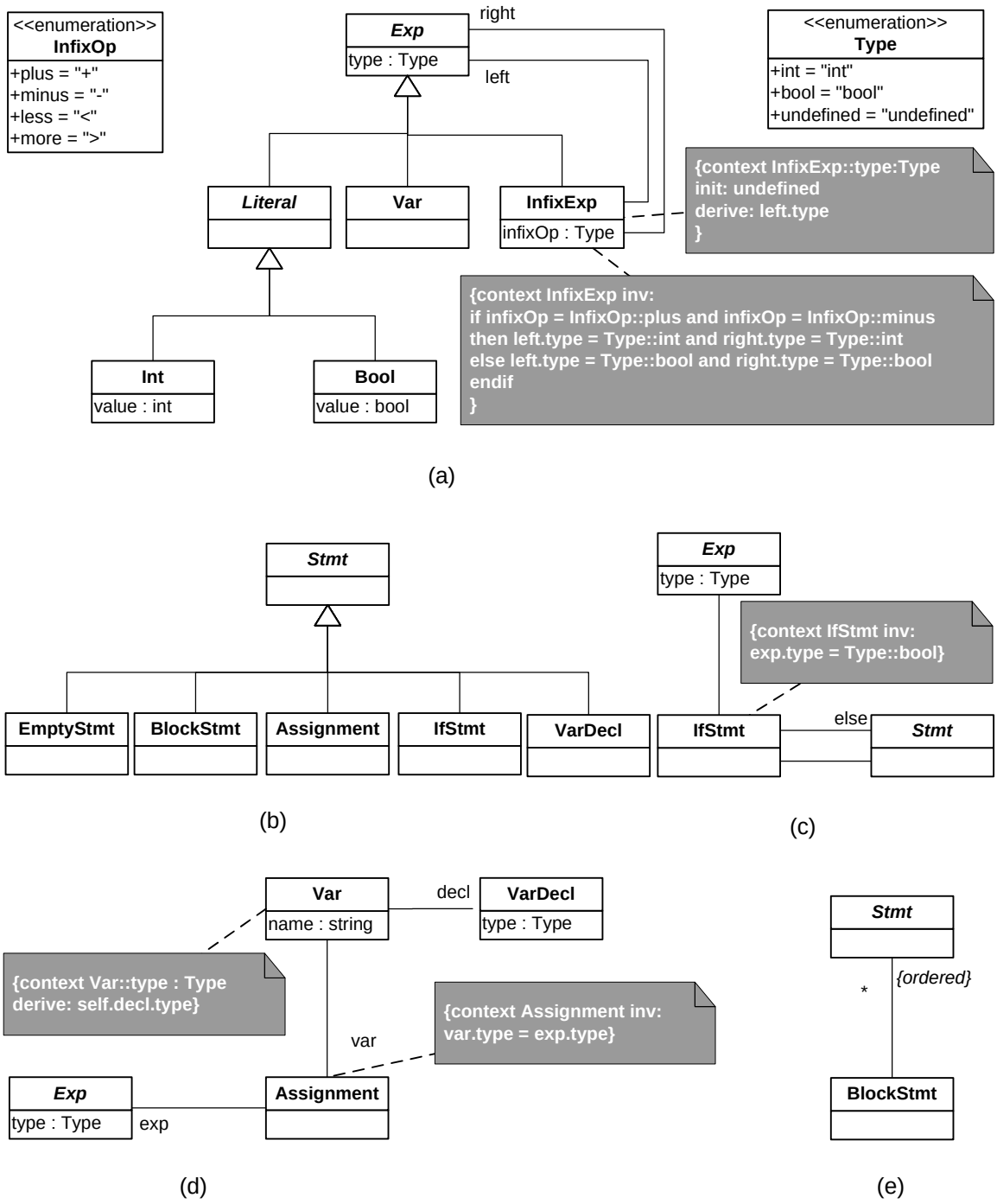


Figure 6.4: Illustration of transforming attribute grammars into UML enchanced by OCL

Following this procedure, the attributes stated in Syntax 6.3 can be translated into a class diagram, augmented by OCL expressions and constraints, shown in Figure 6.4.

6.4 Implementing the xUML-to-Java Translator

6.4.1 Related Eclipse Projects

The overall process of MDA is described by OMG in the MDA Guide [62], which is more about rough specification rather than a ready-to-use tool. We need to identify an MDA implementation, usually referred to as MDA tools or MDA development environments, to serve the purpose of prototyping the translator. At present, albeit in their infancy, various MDA tools have emerged recently: some of them partially support MDA like pure code generation tools; others are more fully-fledged model-driven tools.

A fully-fledged MDA tool is preferred in our case as implementing the translator spans model creation, model transformation and code generation. A fully-fledged MDA tool can support MDA practises in most aspects, including model composition, model loading/persistency, model transformation, code generation and occasionally model weaving, but is not limited to these. To be exempted from mandatory licensing issues, we use an open source MDA tool instead of a commercial one. To our knowledge, MDA-related Eclipse tools [5], resulting from several open-source projects, are the best candidate. The Eclipse projects most relevant to our need are the EMF (Eclipse Modeling Framework) [22] project and the M2M (Model to Model) project [46].

The EMF project provides a modeling framework and code generation facility for building tools as well as other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. The core EMF framework includes a meta model (Ecore) [22] for describing models and runtime support including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically. It should be noticed that although Ecore is referred to as a metamodel in the documentation of the EMF project, however in effect it defines itself, so in this sense, Ecore can be also regarded as an approximation of MOF and as a meta-metamodel from the perspective

of the four-layer metamodeling framework in MDA. Therefore, parallel to MOF, Ecore can specify not only the models but also metamodels.

The M2M project of Eclipse is an implementation of model transformation, which is a key aspect of model-driven development. It delivers various model-to-model transformation languages in which the transformation models are specified, a transformation engine which is essentially the virtual machine of the transformation languages, and some accompanying facilities such as syntax-aware editors, model/metamodel loading, and transformation launching.

In our case, the Atlas Transformation Language or ATL [46, 12] is employed to implement mapping models. ATL is developed by the Atlas group (INRIA & LINA), and is a hybrid transformation language, consisting of a blend of declarative and imperative constructs in which the declarative is encouraged. ATL provides ways to produce a set of target models from a set of source models. ATL transformations are unidirectional, operating on source models and producing target models. Source and target models for ATL may be expressed in the XMI OMG serialization format. Source and target metamodels may also be expressed in XMI or in the more convenient KM3 notation [45].

An ATL transformation can be decomposed into three parts: headers, helpers and rules. The headers are used to declare general information such as the module transformation name, the source and target metamodels and imported libraries. Helpers are subroutines (based on OCL) that are used to avoid code redundancy. Rules are the heart of ATL transformations as they describe how target elements (conforming to the target metamodel) are produced from source elements (conforming to the source metamodel). They are made up of bindings, each one expressing a mapping between a source element and a target element.

The ATL Integrated Development Environment (IDE) provides a number of standard development tools, such as a syntax highlighting editor, debugger, outline view, etc., to facilitate the development of ATL transformations.

The ATL Engine includes two key components, an ATL compiler and an ATL Virtual Machine(ATL VM). The former is responsible for compiling the given ATL transformations to programs in a specific byte-code. Then the ATL VM executes the resultant byte-code. In addition to the byte-code, the execution of an ATL transformation entails that the ATL VM has the knowledge of the metamodels of the source model and the

target. This means, the ATL VM needs to load at least three models or metamodels for a transformation, so the task of loading models is not trivial and is usually complicated by the diversity of models and metamodels. Therefore, the ATL VM is designed to run on top of various model management systems. To isolate the ATL VM from their specifics an intermediate level is introduced called the Model Handler Abstraction Layer. This layer translates the instructions of the VM for model manipulation to the instructions of a specific model handler. Model handlers are software components that provide programming interfaces for model manipulation.

Building and launching ATL Transformations vary depending on whether the ATL translation takes place in the Eclipse environment or works as a stand-alone application.

In the Eclipse environment, the ATL compiler is automatically called on each ATL file in all ATL projects during the Eclipse build process. By default, this process is triggered when a file is modified (e.g. saved). Executing an ATL transformation requires the declared source and target models and metamodels to be bound to actual models. This is done through the launch configuration wizard. The ATL engine delegates reading and writing models to the underlying model handler. When the launch configuration is ready, the user can trigger the transformation by clicking the 'launch' button in the Eclipse IDE.

Under some circumstances, the ATL transformation may be implemented as a stand-alone Java application. To satisfy such a need, the ATL project provides Apache Ant tasks [1] for loading metamodels, serializing resultant models and executing an ATL transformation. The sequence of Ant tasks can be either scripted in an Ant file, which is subsequently executed by an active Ant engine, or directly invoked by a chunk of Java code.

The ATL transformation also supports chained transformation. For example, two sequential transformations A2B and B2C are required to chain, where, preferably, only the final model is serialized, but the intermediary model is not.

The other Eclipse project concerning us is the Model to Text (M2T) project which focuses on the generation of textual artefacts from models and is useful for us to generate Java code from the yielded Java models. This project provides a powerful tool for generating source code: JET (Java Emitter Templates) [3]. With JET one can use a JSP (JavaServer Pages) -like syntax (actually a subset of JSP syntax) that makes it easy to write templates that express the code you want to generate. JET is a generic

template engine that can be used to generate SQL, XML, Java source code and other output from templates. A JET Development Environment based on Eclipse GUI is also provided to ease editing JET template by means of syntax-highlighting. When a JET template is ready, it will be automatically transformed to one or more Java classes, which, together with the provided JET runtime library, can be integrated into an application to play a role of code generation.

6.4.2 Implementing the Conceptual Design in Eclipse

So far, the needed metamodels have been ready for use, including the xUML metamodel (See D), the ALx metamodel (see Appendix E) and the MiniJava metamodel (a simplified Java metamodel provided in Appendix F), which are all specified using Ecore, the metamodel provided by Eclispe EMF project. Moreover, the ATL files implementing the xUML2ALx and ALx2Java mapping models have also been composed and compiled to ATL byte code, which is to be executed in the ATL VM. In addition, the code generator is also generated from the composed JET templates.

With the required metamodels and ATL files ready, thanks to the Eclipse ATL project, the implementation of the translator becomes straightforward. The whole system is constituted by two major components: the ATL engine, which is already an off-the-shelf component provided as a part of the ATL runtime libraries, and the code generator, which is automatically produced from the JET templates composed by us. Their behaviours are coordinated in a Java main method, shown in Figure 6.5. The process of a particular transformation could be decomposed into the following steps:

1. Instantiate an ATL VM and Instantiate an EMF model handler.

2. Load the required metamodels and an input xUML model. The loading order is not significant.

3. Launch the first translation by the following configuration.

   (a) Bind the loaded xUML metamodel and ALx metamodel to the corresponding variables declared in xUML2ALx ATL file for metamodel references.

   (b) Bind the loaded input xUML model to the corresponding variable declared in the xUML2ALx ATL file for model reference.

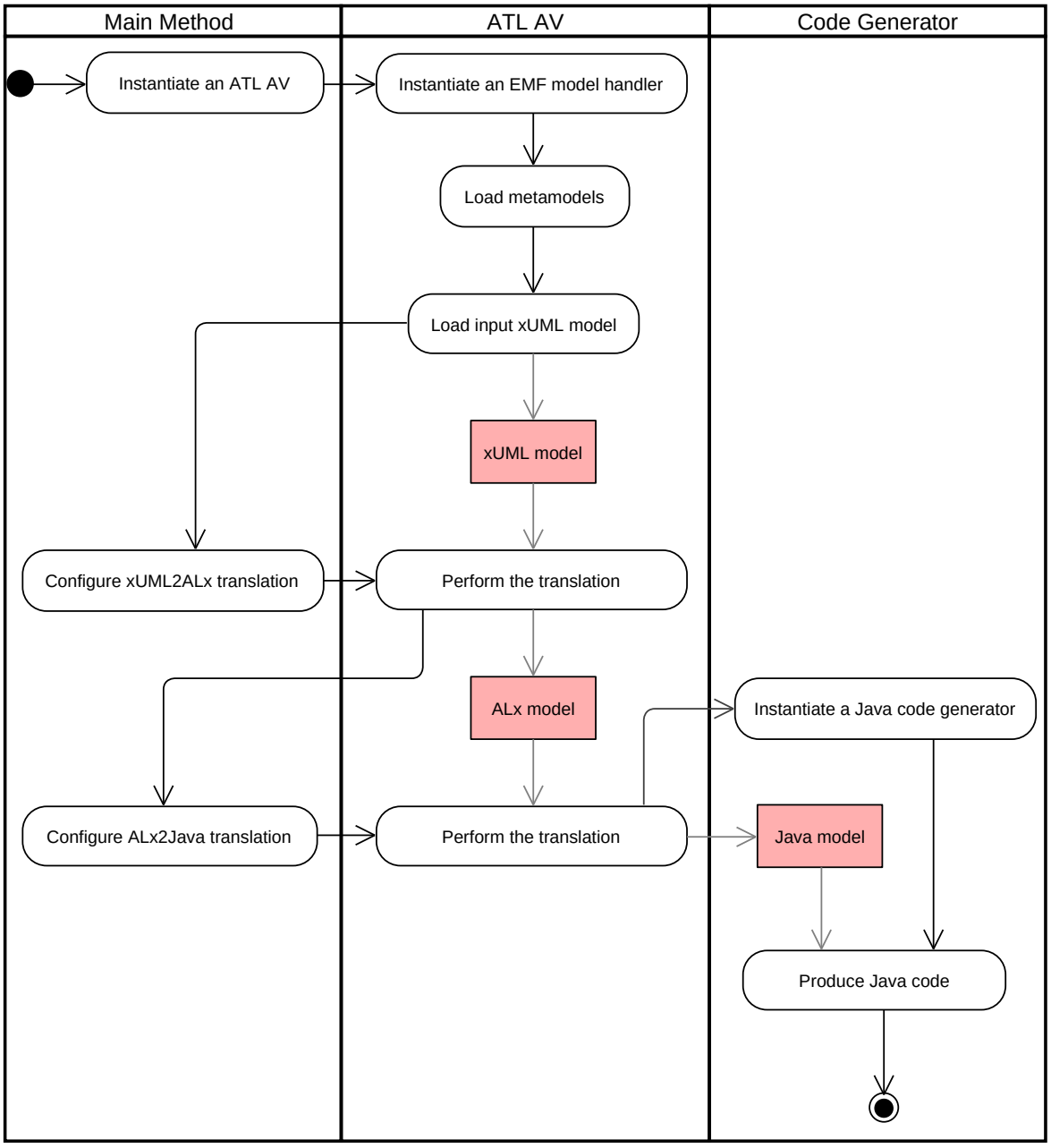


Figure 6.5: Activities in a particular xUML-to-Java translation.

(c) Specify the compiled xUML2ALx ATL file as the required transformation files.

4. If required, serialize the intermediary in-memory ALx model resulted from the first translation.

5. Launch the second translation by the following settings.

   (a) Bind the loaded ALx metamodel and Java metamodel to the corresponding variables declared in ALx2Java ATL file for metamodel references.

   (b) Bind the loaded input UML model to the corresponding variable declared in the ALx2Java ATL file for model reference.

   (c) Specify the compiled ALx2Java ATL file as the required transformation files.

6. Invoke the code generator to generate Java code from the resultant in-memory Java model.

The whole execution scenario is completed when the Java code is generated. Often, the generated code is then fed to an instantiated Java VM to be executed for the purpose of model simulation, which is not difficult to implement with the help of Ant tasks and thus will not be detailed in the thesis.

It can be observed that both of the sub-transformations are in fact performed by a single instantiated ATL VM, but launched by different settings. For details of the Java main method, see Appendix G.

6.5 Metamodels and ATL files

We use UML to represent the metamodels of xUML, ALx and MiniJava. The three metamodels in UML are neutral to MDA environments. These metamodels should be rewritten in Ecore so that they can be recognized by the ATL engine because, in our case, we employ the Eclipse MDA environment where Ecore is the primary metamodeling tool. Nevertheless, the rewriting is straightforward and mostly a one-to-one mapping as Ecore is an approximation of MOF or the UML core. The reader is referred to Appendix D, E and F for the complete UML representation.

Two major ATL files are involved to implement the xUML-to-ALx mapping rules and the ALx-to-Java mapping rules. As mentioned, ALx is designed to be the textual

correspondent of xUML, so their mapping rules are straightforward and implementing them in ATL is also trivial. The ALx-to-MiniJava mapping rules have already been provided in Chapter 4, their implementation in ATL is also straightforward. So we do not explain the ATL files in detail in this thesis. The reader can navigate to Appendix J for excerptions of ATL files,

6.6 Generated Java Code for the Elevating System

We use a set of example xUML models (in XMI) to test the developed translator, we find that the translator functions well and produces the Java code as expected. In Section I.5 of Appendix I, we show the generated Java code for the elevating system that is provided in Chapter 4 as the running example of this thesis.

6.7 Conclusion and Discussion

The system of the ALx-to-Java translator can be considered as a model transformer, which is central to modeling and composing mapping models. We consider that MDA is the best approach to implementing the translator, because MDA is created for model transformation. In this chapter, the feasibility of applying UML to programming languages is investigated, and an approach to translating the static aspects of programming languages into UML models is proposed. The Eclipse implementation of MDA is employed to develop the translator because Eclipse provides a complete spectrum of ready-to-use facilities to support MDA, such as model transformation languages and engines, and model serialization.

The practice of using MDA to develop the translator shows that MDA is formal, can raise abstraction level and save time and effort. In the following chapter, we further corroborate this point by comparing MDA to a conventional way of implementing the translator.

Chapter 7

MDA Comparison with Conventional Approach

The practice of adopting MDA to implement the xUML-to-Java translator presented in Chapter 6 shows that MDA is a viable approach to programming language translators, especially in the situation where rapid development is expected. However, we still seek to further corroborate the fact that MDA is advantageous over the conventional approach in implementing language translators. Therefore, we construct another translator using the conventional approach so that we can make a comparison between the two approaches.

This chapter begins with the introduction to the background and a conventional approach of language implementation. We subsequently describe the key activities involved in implementing the ALx-to-Java translator using the conventional approach. Finally, a comparison is made between MDA and the conventional approach based on these practices, and the pros and cons of the two approaches are analyzed.

7.1 Background of Language Implementation

7.1.1 Compilation and Interpretation

Generally, two approaches exist to develop language implementation: interpretation and compilation [10]. An interpreter takes as input a program in some language, and performs the actions written in that language on some machine, while, using the approach of compilation, a compiler takes as input a program in some language, and

translates that program into some other language, which may serve as input to another interpreter or another compiler.

The key difference between an interpreter and a compiler lies in the fact that a compiler does not directly execute the program: ultimately, in order to execute a program via compilation, it must be translated into a form that can serve as input to an interpreter, which could be real machines or virtual machines.

The ALx-to-Java translator under development is a kind of compiler. However it has the feature of transforming a high-level language to another high-level language (from ALx to Java), unlike other compilers that translate source code from high-level programming languages to lower level languages, e.g., assembly language or machine language. So, usually the term 'translator' is used to refer to the former, whereas the term 'compiler' is used in the latter.

7.1.2 Conventional Language Implementation

There are many language-implementation approaches, which differ in terms of paradigms and application areas. MDA is one example. For another example, some researchers are inclined to generate the implementation of a language in a computerized manner based on its formal specification. This paradigm of implementing language enables fast development, assures logical correctness and reduces inevitable hand-coding errors. It is mostly applied in academic research, critical applications and domain specific languages.

It is not the scope of this research to address all the available language-implementation paradigms or approaches, as well as their strengths and weaknesses compared to MDA. Instead, of interest to us is only the traditional and typical language-implementation approach, which is predominantly adopted in commerce. In the current thesis, we refer to this approach as the *conventional approach*. Note that the word 'conventional' does not mean that this approach was once used in the past and now it has become obsolete, but means that this approach has a longer history than MDA.

The conventional approach segments the process of compilation or translation into two sequenced phrases: *the front end* and *the back end* [10]. The front end would analyze the source code to build up an in-memory representation of the program, called the *intermediate representation* or *IR* [85]. The back end maps the produced IR into target code, which, in a fully-fledged language processor, usually requires some actions

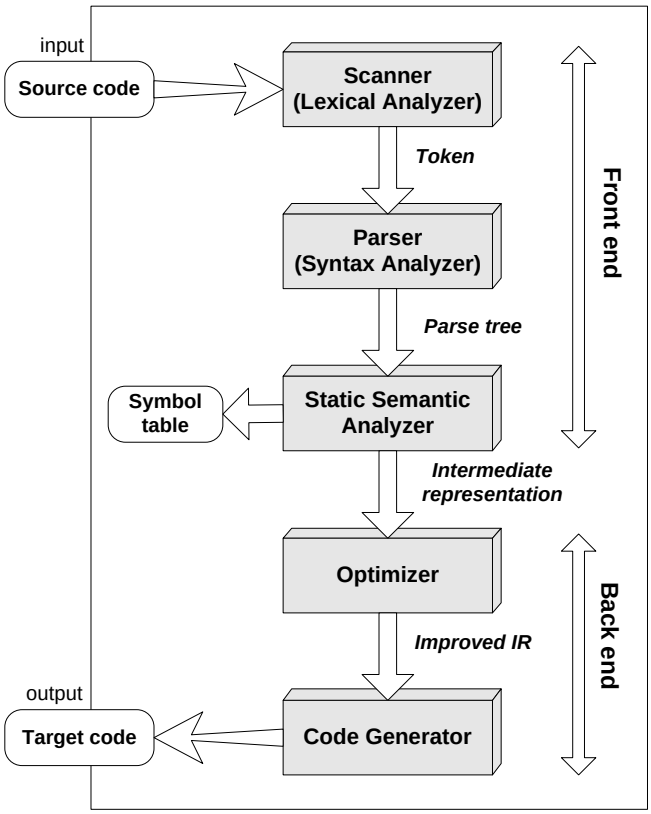


Figure 7.1: Architecture of conventional compilers or translators

like code analysis and code optimization over the IR. See Figure 7.1 for details.

Bear in mind that the departure point of building the xUML-to-Java translator in the conventional approach is that we need to compare the MDA with the conventional approach in the respect of language implementation. In building the translator in MDA, code optimization, a significant and sophisticated part in the back end, is not considered, thus we do not consider it either when using the conventional approach. For this reason, in describing the conventional approach, we mostly concentrate on the front end.

More specifically, the front end can be further decomposed into the following phases: *lexical scanning*, *syntactic parsing* and *static semantic analysis*.

Lexical scanning

A lexical scanner reads source files and break down the text stream into tokens, each of which is a single atomic unit of the language, e.g., a keyword, an identifier, a numeral or a symbol name. The syntax of tokens, often referred to as the *micro-syntax* of the language, is typically specified by a regular language, thus a finite state au-

tomaton constructed from the regular expressions is employed to recognize it. In the conventional approach, the scanners or lexical analyzers are mechanically generated by parser-generating tools such as YACC [44], JavaCC [6], etc., and the underlying theory has been well studied and practised.

Syntactic parsing

The tokens identified in lexical scanning are passed to the parser, which checks that the correct language syntax is being used in the program. In this step, the program is converted to its parse tree representation.

Parsers may vary in their parsing strategy, which can be categorized into two kinds:

- Top-down parsing [37]: a means of analyzing the token sequence by hypothesizing general parse tree structures and then determining whether the known fundamental structures conforms to the hypothesis. The action of hypothesizing is a process of deciding which production of a non-terminal should be followed to continue the parsing. The decision making requires the preliminary computation of First and Follow set of all non-terminals as well as their nullability.

  Top-down parsers are often referred to as recursive-descent parsers, because they are typically built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

- Bottom-up parsing [37]: is known as shift-reduce parsing and is a strategy for analyzing unknown data relationships that attempts to identify the most fundamental units first, and then to infer higher-order structures from them. It attempts to build trees upward toward the start symbol. This type of parser is not described in this thesis because they are not related to our research.

The parsers most relevant to our implementation are predictive parsers, which are a kind of recursive descent parser. The distinguishing feature of predictive parsing are that it does not require backtracking and it is possible only for the class of LL(k) grammars [37]. In addition, predictive parsers run in linear time.

Parsers are usually generated by parser generators, such as JavaCC and YACC, based on analytic grammars written in a specified notation akin to EBNF. YACC

generates parsers in the C programming language, while JavaCC produces parsers in Java. The latter is adopted in this research and is detailed in Section 7.2.1.

Static semantic analysis

The static semantic analyzer mainly performs the construction of the symbol table, type checking and building IRs. They are fulfilled by the following three software components.

- Symbol table constructor, which adds semantic information to the parse tree and maintains the symbol table (also called *environments*) that maps identifiers to their meanings. When the declaration of types, variables and methods are processed, their identifiers are bound to *meanings* in the symbol tables. Take a class declaration of an OOP as an example: the class name is bound to the definition of this class, composed of the definitions of the contained variables and methods.

- Type checker. This may be available in the front end if the language is designed to have static type checking, which means that the process of verifying and enforcing the context-sensitive constraints occurs at compile time. The type checking that is performed at runtime is called dynamic type checking. The type checker would consult the previously-built symbol table to know the meanings of identifiers, such as the type of variables, the definition of classes, etc. The type checking is conducted according to the formal context-constraints rules, and it raises errors when encountering ill-formed programs.

- IR producer, which performs the final tasks of the front end—translating the parse tree into abstract machine code. The IR producer is necessary in the sense that although it is possible to translate directly to real machine code, such doing hinders portability and modularity. An IR is expressed in a kind of abstract machine language that can express the target-machine operations without committing too much machine-specific detail, and it is also independent of the details of the source language.

7.2 Conventional-Approached Translator

For simplicity, the translator implemented with the conventional approach is not xUML-to-Java but an ALx-to-Java translator. To guarantee the comparability of the two approaches, functionalities of the two translators should be identical, so the xUML-to-Java translator implemented in MDA should be programmed as an ALx-to-Java translator. This is done by turning off the first sub-translation (from xUML-to-ALx) and incorporating an ALx model loader for loading source ALx files (The ALx model loader is generated from the ALx metamodel using Eclipse M2T tools). Furthermore, the MDA translator deals with no code optimization, thus the conventional ALx-to-Java translator is also thin: the phrases involved in the translation include only lexical analysis, parsing, symbol table building, type checking and code generation, omitting the back end.

7.2.1 Background of JavaCC

JavaCC [6] is adopted as the platform to build the translator. Its main features are:

1. The target language of JavaCC is Java.

2. JavaCC generates top-down or recursive-descent parsers and is confined to the LL(k) class of grammars excluding left recursions.

3. JavaCC parsers do not support a back-tracking mechanism, however JavaCC allows customizing look-ahead symbol numbers to compensate for the compromised recognizing power incurred by the lack of back-tracking.

4. The scanning and the parsing of JavaCC parsers take place in a single pass. In this pass, scanning and parsing alternate: scanning is triggered when the parser starts to consume the next token; when the token is identified by the lexical analysis and passed up to the parser, parsing is started.

5. JavaCC parsers construct parse trees from the bottom up albeit being top-down parsers.

6. JavaCC provides a user library, called JJTree, to facilitate the construction of parse trees. Additionally, JJTree provides full support for the tree-visitor design

pattern, which greatly eases constructing the symbol table builder and the type checker.

7. JavaCC is a mature open-source project and accessible from its website; the documentation and manuals are well written.

7.2.2 Major Development Activities

The following major activities are involved in building the translator in the conventional approach.

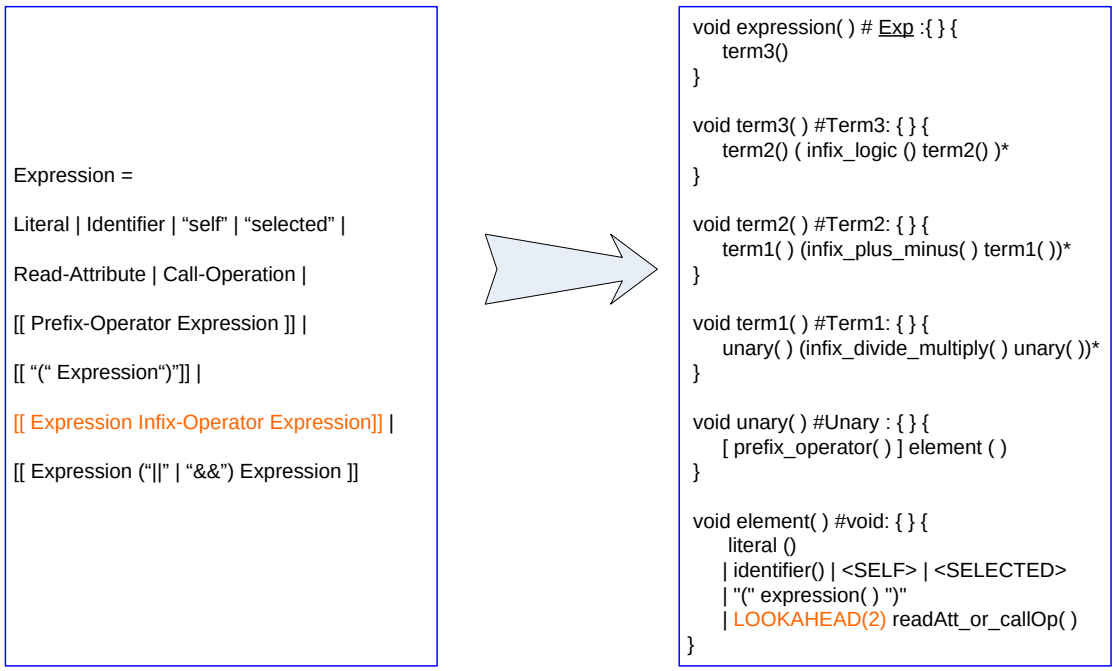
Specifying the micro-syntax

The micro-syntax of ALx is specified in a form that is recognizable by JavaCC. The analytic micro-syntax specifies three lexical states of the lexical scanner: IN_SINGLE_LINE_COMMENT, IN_FORMAL_COMMENT and IN_MULTI_LINE_COMMENT. Additionally there is a pre-defined state, called the DEFAULT state. The generated lexical scanner is at any moment in one of these lexical states, and starts off in the DEFAULT state when initiated. The name of each state explicitly indicates what kind of situation it denotes. For instance, the state IN_SINGLE_LINE_COMMENT refers to a situation when lexical scanner is in processing a single line of comment. The transition to this state is triggered by the symbol '//', while the transition out of this state is trigged by a new line symbol ('\n', '\r' or '\r\n').

All tokens are generated in the state DEFAULT. Unlike the strings in comments, which are not passed to upper-layer parsers although identified, the matched tokens are conveyed to the parsers. Tokens primarily include identifiers, keywords, literals and operator symbols. In JavaCC, all aspects of a token, including the image, type and location of the token, are encapsulated in an object of the pre-defined class 'Token'. It is necessary to highlight the distinction between tokens and special tokens: in JavaCC, the special tokens do not participate in parsing; whereas the tokens do.

Specifying the generative grammars

We follow the abstract syntax of ALx (provided in Appendix A) to compose the corresponding generative grammars. In structure, the latter is parallel to the former because JavaCC, as mentioned, is a kind of recursive-descent-parser generator. In the process,



(a) The abstract syntax of Expression

(b) The corresponding generative grammars

Figure 7.2: Illustration of left recursion elimination and local LOOKAHEADs settings

special care is paid to circumvent left-recursion and backtracking. The following two cases are highlighted to illustrate how to rewrite left-recursion grammars and avoid backtracking.

The first case is in relation to the grammar of ‘Expression’. See (a) of Figure 7.2: one production of the non-terminal ‘Expression’ is defined left-recursively as ‘Expression = Expression Infix-Operator Expression’. Without removing this left-recursion, the generated parser would suffer the problem of non-termination. Hence, it is necessary to rewrite the left-recursive definition to an alternative form which is absent of left recursion but is equivalent to the original one in semantics.

We apply a standard rewriting method [37] to transform all left-recursions in the grammar. Simply speaking, this method is characterized by creating new non-terminals, which are right-recursively defined and are usually called ‘tails’ or ‘rests’, to replace the problematic left-recursive parts. By so doing, left recursions are transformed into right ones, which are then free of the problem of non-termination. For a complete and formal description of this method, the reader is referred to [37]. We apply this method to the non-terminal ‘Expression’ to remove the left recursion. See (b) in Figure 7.2 for the result, where the precedence of operators are also considered.

The second case is to illustrate how to customize local LOOKAHEAD to circumvent

backtracking. Backtracking cannot be ignored if the parser could make wrong decisions in choosing production rules. As mentioned, JavaCC parsers do not support backtracking but allow for the local lookahead settings to direct the parser to make right decisions at the ambiguous places. See (b) in Figure 7.2 for an example. In the right-hand side of the non-terminal 'Expression', the 'Identifier', 'Read-Attribute' and 'Call-Operation' are three forms of the 'Expression'. According to their definitions, they all begin with identifiers. Without proper lookahead, the parser would always choose the first case (Identifier), never switching to the other two possible cases (Read-Attribute and Call-Operation). This is not correct. As a result, a local lookahead setting is necessitated here to make the parser to look more symbols ahead to achieve the right decision.

Generating the scanner and parser

Now that the micro-syntax and the generative grammars of ALx have been composed in the form that is friendly to JavaCC, the generation of the scanner and parser is simple. There is no need to detail this activity.

Coding the symbol-table builder

The functionality of building the symbol table is implemented following the tree-visitor design pattern. In the run-time of the translator, immediately after the parse tree is produced, the symbol-table builder is called to operate on nodes of the parse tree. The symbol-table builder specifies an action for each type of node. That is, when a node of a type is visited, an appropriate action is performed. Note the action for a type of nodes may be null: for instance, the tree visitor does nothing on expressions and statements. The nodes of ALx parser trees that have effects on the symbol table are the declaring nodes, which include the declarations of local variables, formal parameters, classes, fields, operations, relations, events, state machines and states.

The procedure of updating the symbol table could be abstracted as follows: binding names (variable names, class names, state names, etc.) to the definitions and then adding this binding as an entry to the symbol table. The definitions within the table can be retrieved later by name. The definitions vary in data structure: for instance, the definition of a local variable comprises only its name and its type, whereas the definition of a class is far more complicated, containing the class name, a collection of references

to the definitions of its member fields, and another collection to its member methods.

In our case, the records of the symbol table are not linearly arranged, but are distributed to the corresponding scopes. In the grammar of ALx, the pair of curly braces ('{ }'), occurring in the constructs like block statements, class declaration, state machine declaration, etc., marks the beginning and closure of a scope. Scopes are nestable, so they are essentially structured as rooted trees. The records resulting from declarations directly contained in a scope is entered by being associated with this scope. Therefore, the symbol table for ALx is a complicated data structure: its backbone is a rooted tree; the nodes within the tree are used to represent scopes; each node may associate with one or more records produced by the declarations in the scope that the node represents. The close integration of scopes with the symbol table determines that the scope building and the entry of symbol-table records are performed together.

Some operations are provided on the symbol table. The method **lookup (Symbol key)** serves for retrieving the meaning that is associated with the symbol given as the parameter. ALx adopts the *static scoping* or *lexical scoping* mechanism [8], where a variable always refers to its top-level environment, and matching a variable to its binding only requires static analysis of the program text, irrelevant to the runtime call stack. So, the lookup of a variable always searches the current scope first and then go up to the parent scopes, which is a recursive process, terminated either if the desired definition is found or if the root scope is reached.

Implementing the type checker

ALx is a static and strongly-typed language, thus its type checking is undertaken at compile time. As with the implementation of the symbol-table builder, the type checking of ALx is also implemented using the tree-visitor pattern. The type-checking actions defined in the type checker are distinguished into three categories:

- If the node being visited is an identifier, retrieve definitions of identifiers from the global symbol table.

  To retrieve the proper definition of an identifier, it is crucial for the type checker to know which scope the current identifier is in. This is assured by the means that in the process of traversing the parse tree, the type checker traverses the scope tree accordingly.

The retrieved information, like the declared type of a local variable, is then used in the subsequent type-checking actions. This shows that the type checker has data dependency on symbol building, and this is why type checking occurs after the symbol-table building is completed.

- Inferring the attribute of nodes. The 'attribute' here refers to the one in the concept of *attribute grammars*. For instance, the type of an 'expression' node is inferred based on its child node 'operator'.

- Checking type-validity rules. These actions verify that the type of any expression is consistent with that expected in the context where the expression appears. For an example rule, the logical operators can be only applied to bool-typed expressions. For another, a local variable has to been declared before it is used. If the checking of a rule fails, the type checker would issue an error to the console.

The type checker adopts the following criteria to judge type consistency. 1) For primitive types, two types are considered consistent if their names are identical. 2) Because ALx is a language with sub-typing, for two classes (suppose class A and class B), if A is a subtype of B, then a value of type A can be used in a context where one of type B is expected, but the reverse is not true.

Implementing the code generator

The code generation is the final stage in the translation where the Java code is produced. It comprises two sequential steps. The first step is converting the parse tree into an intermediary internal representation, a parse tree augmented by code-generative information, which is the abstract syntax tree of MiniJava. The conversion is guided by the mapping rules between ALx and MiniJava, which is formally specified in Appendix H. The second step is turning the MiniJava abstract syntax tree into textual Java code, which is the reverse operation of parsing Java programs. The two steps are implemented both by forms of tree visitors. The detailed description of the two steps is ignored.

Note that the generated Java code is not a complete system. Like the Java code produced by the MDA translator, it must be combined with the Java library mentioned in Chapter 6.

Category	Indices	MDA	Conventional Method
Developing Efforts	LOC	*1789*	*3230*
	Developing Threshold	*easy*	*difficult*
Code Quality	Re-usability	*good*	*bad*
	Portability	*good*	*bad*
	Adaptability	*good*	*bad*
	Separation of Concern	*good*	*bad*

Figure 7.3: Comparison result of source code quality.

7.3 Comparing Two Approaches

To further investigate the features and advantages of the two approaches, a comparison is conducted. We consider that a combined comparison method that is a hybrid of qualitative comparison and quantitative comparison is appropriate in our case. A range of variables have been singled out to be compared, such as *development effort*, *code quantity* and *system performance*. Some of them can be measured in quantity, such as the number of code lines that reflects development effort, and the time elapsed for translating example xUML models. While, for some of them, it is more practical to analyze in a qualitative manner, such as code portability, code reusability and code maintenance.

The comparison is made on three overall aspects: development effort, code quality and performance of the resultant translators. Each aspect may be sub-categorized into multiple variables. The comparison results in the aspects of development effort and code quality are summarized in Figure 7.3.

7.3.1 Development Effort

The *development effort* is measured in two dimensions, the number of code lines and the ease of development.

Lines of Code (LOC)

The constituent artefacts of the two translators differ in their origin: they may be newly composed code, imported libraries or generated by tools. Hence, it is necessary

to clarify which artefacts should be regarded as code that would be taken into account in the comparison. Two principles are used for this issue.

Firstly, reused code or a library is not counted. Both approaches have reused third-party libraries or software components, but in varying degrees. For instance, in MDA, the reused software components include the model de-serializers and the ATL VM; in the conventional approach, various JavaCC run-time libraries are reused, among which the JJTree is most notable. These third-party codes are not considered in the LOC comparison.

Secondly, code generated automatically is not taken into account. For instance, in MDA, we use the metamodel of xUML (augmented with OCL expressions) to generate a part of the code of the model validator, which thus is not counted in the comparison. As to the conventional approach, the scanner and parser are generated and not considered when counting code.

In addition, not all artefacts composed by us are taken into account. In the MDA translator, the following efforts are not considered.

1. Modeling the static aspect of the three languages. We do not attribute this effort to MDA in that it is a part of designing languages rather than implementation.

2. Composing the library part. Even though significant work, this is not considered for comparison because it is shared in both approaches.

Therefore, for MDA, we count only the LOC of three artefacts: ATL transformation rules, code-generative JET templates and the Java main method for model transformation configuration.

Regarding the conventional approach, specifying the micro-syntax and the generative grammars of ALx is not considered as significant efforts. This is because despite involving some treatments such as lookahead settings and left-recursion elimination, generally it is just a straightforward process of rewriting the regular expressions and EBNF-like grammars into JavaCC-friendly forms, which is relatively trivial. Therefore, as to the conventional approach, we count the code of the tree visitors for symbol table construction, type checking, IR evolution (transforming the parse tree to Java abstract syntax tree) and code generation, as well as the code for symbol table.

We count code lines based on the mentioned principles, finding that the number of code lines in MDA is only 1789, in contrast to 3230 in the conventional approach. The

latter is almost twice the former. We attribute such a big difference mostly to the two merits of MDA: excellent modularization and the reusability of models. The excellent modularization of MDA is embodied by the fact it draws a sharp line between model composition (including domain model and mapping model), model transformation and code generation. This means, the tool or software components, which are relatively invariant parts of a large number of heterogeneous systems, can be developed separately and reused in different contexts. The reusability of models is embodied by the fact that some design models become valuable in development rather than a simple documentation.

Ease of Development

Although the LOC is a widely-used and quantity-based index to measure programming effort, its authenticity depends on the truth of the assumption that the real average cost of a line of code is equivalent or at least is not so different. To address the weakness of LOC comparison, we analyze qualitively the threshold of applying the two approaches for developers, indicating how easy the development is.

The practice of using MDA shows that it is able to lower the threshold in software development through raising the level of abstraction from the code level to the model level. In our case, the prerequisite of using MDA includes three aspects: understanding the basic concept of MDA paradigm, being able to use UML to compose models and knowing how to establish mapping models. Undoubtedly, some basic formal language theory knowledge is preferable.

In contrast, to use the conventional approach, we need to know more sophisticated formal language techniques, such as those of removing left recursion, lookahead setting, implementing abstract syntax in Java, tree-visitor pattern and type checking.

Therefore, we conclude that the MDA implementation is considerably easier than using the conventional approach if one starting with no prior knowledge.

7.3.2 Code Quality

We measure the source code quality of two approaches in terms of the following four aspects.

Resuablity of Code

The practice shows that MDA brings about better code (or model) reusability compared to the conventional approach. The reusability in this context refers to whether an artefact produced can be reused for other purposes. It is creditable that in MDA, the major reusable artefacts, including language metamodels and mapping models, can be reused in other similar applications and for other purposes. For instance, the models and metamodels can not only be reused independently of implementation, but also be regarded as system documentation.

However, in the conventional approach, no artefacts can be reused with such ease because they are language-specific and strongly mutually dependent. For instance, the scanner and parser generated by JavaCC are specific to Java, and the type checker has data dependency on the symbol table constructor, which prevents the reusability of the artefacts harvested in the conventional approach.

Portability of Code

The practice also shows that MDA produces code (or models) with better portability in comparison to the conventional approach.

The better portability of MDA artefacts comes from the fact that the primary departure point of MDA is raising the level of abstraction and not introducing implementation-specific details in the early stage of the system development. In our case, although the MDA platform we adopted is Java-specific, we are not confined to it. Instead, we can choose other MDA platforms in other programming languages if required, but models and metamodels can be still reused because they are language-independent. This case is not true for the conventional approach. If the translator is expected to be programmed in another language, the work would be tedious, involving choosing another parser generator, re-composing the grammar files, re-coding the symbol constructor and type checker.

Adaptability of Code

It is observed that the code resulting from MDA has superior adaptability of code over that from the conventional approach. This adaptability of code is highly demanded because the aimed translator is a prototype system, where the language models are

changed frequently and the time-to-run is critical for timely observation of the effect of changes. This merit of MDA arises from the fact that models are code. Compared to the textual code in traditional programming paradigms, models are more abstract and high-level, thus enjoying better readability. More important, in MDA, modifying the system can be achieved by modifying the models. This advantage of MDA was very useful in designing xUML and ALx.

However, in the conventional approach, changes have to be made at the code level. This means more time and efforts would be consumed, and the test of a new design is not as quick as using MDA.

Apart from that, the characteristic of MDA that models are the system also makes MDA advantageous over the conventional approach in terms of readability, maintenance, complexity of the code. Their detailed description is omitted here.

Separation of Concerns

We also find that MDA has the feature of separation of concerns, embodied primarily by the following aspects:

- the separation of the model description and in-memory or in-persistence model representation .
- the separation of the description and the executions of translation rules.
- the separation of model de-serialization and static semantic checking.

Owing to these separations of concerns, some services, such as the OCL engine (for checking OCL expressions) and ATL execution engine (for executing translation rules), can be provided by third parties.

However, in the conventional approach, the separation of concerns is not obvious. The components within the conventional translator have strong dependency in terms of both data and functionalities. For instance, the structure of the symbol-table-constructing tree-visitor depends on the structure of the abstract syntax of ALx; the type checking depends on the structure of the symbol tables produced in the preceding construction of the symbol tables. Such tight coupling of components in the conventional approach makes even a minor modification to code problematic.

7.3.3 Performance Comparison

We gauge the performance of the two translators in the perspective of time and memory space consumed in their execution. In the experiments, we ran the two translators with a set of well-designed sample xUML models as input, and made record of the memory footprint and execution speed of each execution instance. All the experiments were made on a Windows XP work station with the following features:

```
Intel Pentium(R)4 3.0G CPU.
1.5G DDR 333MHZ RAM,
SAMSUNG SP0411C 40G HDD.
Windows Xp Professional SP3 Version 2002.
Java<TM> SE Runtime Environment (build 1.6.0_07)
Java HotSpot<TM> Client <build 10.0-b23>
```

We created four size-varied sample xUML models (see Appendix I for details) for the experiments, the scale of which depends on the number of the classes, associations, state machines and ALx code lines of the model.

- The xUML model of the elevating system, which is the running example described in Chapter 4.
- The xUML model for a gas station system.
- The xUML model for a taxi-booking system.
- A hypothetical xUML model for simulating the relay of a message, which is large-scale and has 1000 homogeneous classes and state machines. The model is automatically generated.

We integrate into the source code of the translators a separate thread to keep track of the memory consumption at an interval of 0.1 seconds, and several lines of time-counting code to work out the elapsed time and print it out. The overhead of performance analysis code is minor compared to the overall system time and thus ignored.

For each execution instance, the memory consumption is measured by the combination of the peak memory consumption (PMC) and average memory consumption

Sample xUML models		MDA			Conventional		
Name	Scale	Elapsed time (second)	Memory Usage(KB)		Elapsed time (second)	Memory Usage(KB)	
			Peak	Average		Peak	Average
Traffic Light	2 Classes 1 Associations 1 State machine 35-line Alx code	1.134s	623	421	0.970s	321	213
Elevator	8 Classes 5 Associations 1 State machine 78-line Alx code	1.139s	634	423	0.976s	323	223
Gas Station	8 Classes 9 Associations 2 State machines 147-line Alx code	1.139s	655	424	0.976s	325	232
Taxi-Booking	13 Classes 14 Associations 1 State machine 328-line Alx code	1.141s	678	431	0.978s	334	256
Message Relay	1000 Classes 999 Associations 1,000 State machines up to10k Alx code	2.23s	892	721	1.87s	429	388

Figure 7.4: Comparison result of performance

(AMC). Each sample xUML model is executed five times, and then the average performance of the five executions, including average PMC and average AMC, is worked out. See Table 7.4 for details of the comparison.

From Table 7.4, the conventional translator is more efficient than the MDA in terms of both execution speed and the memory resource consumed. This is reasonable because the MDA translator makes use of various general-purpose components, such as the model reader/writer and ATL VM, which are intended to accommodate general use, thereby inevitably compromising efficiency. Whereas the components yielded by the conventional approach are more specific, serving solely for our purpose, hence being more efficient. However, it can be noticed that the margin between them is so minor that it is negligible when the application is not time-critical. In prototyping a language, usually the time to run and code adaptability are the major concerns. So we assert that MDA is more suitable in rapid development of a source-to-source translator.

7.4 Discussion and Conclusion

We use a typical conventional method to implement an ALx-to-Java translator and compare it with the MDA one, finding that MDA has advantages over the conventional method in terms of code quality, such as better usability, better portability, loose component coupling, and less developing efforts.

However, it is still unsafe to assert that MDA is absolutely advantageous over the conventional approach. The ALx-to-Java translator is a language implementation using a source-to-source translation approach. In our case, the translation is focused mainly on the syntactic level and static semantics of the languages, dealing little with dynamic semantics. As a result, the comparison is unclear about the performance of MDA in coping with dynamic aspects of the system, thus it cannot be concluded that MDA has advantages over the conventional approach in processing the dynamic aspects of languages.

Another point should be considered. We define the conventional approach as one where a parser generator like JavaCC and YACC is used to generate a lexical scanner and parser from a language specification, from which the following components are developed, such as symbol table builder, IR producer, code optimizer, and object code generator. However, the approaches to developing language implementations

are not limited to MDA and the conventional approach. Some researchers use Meta-environment [101] which is based on an algebraic semantics and term re-writing techniques to prototype language implementation. In addition, functional languages and logical programming languages are also creditable in prototyping languages. So it is future work to compare MDA to these approaches.

Chapter 8

Conclusion

8.1 Summary

This work was motivated mainly by the semi-formal nature of UML. Although the static aspects of UML, a general-purpose visual modeling language, are well-defined in the four-layer metamodeling approach, however its dynamic semantics is specified in a plain natural language, English. This inevitably leads to loopholes, ambiguities and inconsistency of its semantics and thus precludes reasoning about and simulation of system models which are specified by it. Many attempts have been made to supply UML with formal semantics using set theory, first-order logic and graph transition, or translating UML to existing specification languages, such as Z, B and CASL. However, none of them were based on a mature semantics-describing framework, hence lacking solid ground in theory and practice as well as tool supports, or they just address a particular diagram of UML. Therefore, we were determined to specify the semantics of UML in a mature and user-friendly framework.

We reviewed some main-stream semantics-describing approaches such as denotational semantics, operational semantics and Action Semantics (AS), and the emphasis was put on AS. AS is a hybrid semantic description framework taking advantages of denotational semantics, structural operational semantics and algebraic specification. It defines as the major semantic entities a set of actions whose execution semantics are well-defined using structural operational semantics. To describe semantics of a language, one only needs to be concerned with translating the constructs in this language to the appropriate actions or other semantic entities like yielders and data. The transla-

tions are expressed in semantic functions defined by semantic equations. Furthermore, action semantics provides some ready-to-use predefined data types so that users can easily import some of them in their description for efficiency. Flexibly, users are also allowed to define their own types depending on what languages they are describing. The notations of actions are carefully selected to those intuitive English words and phrases to achieve best comprehensibility. Unified algebraic formalism is extensively used in describing the action notations, sorts, data types, semantic functions, semantic equations, and transition rules of the action machine.

We compared AS with the traditional semantics-describing frameworks, finding that AS has advantages of readability, extensibility, modularity and practicability. These properties of AS are required by UML's formal semantics in that UML is a hybrid and composite modelling language, evolving fast, sharing some concepts with AS and being intended for more general users. Therefore, we decided to adopt AS as the vehicle to formalize UML. Furthermore, instead of defining UML directly, we designed an Action Language, called ALx, and used it as the intermediary between UML and action semantics of UML. ALx is characterized by heterogeneity, combining simultaneously the features of Object Oriented Programming Languages (OOPL), Object Query Languages (OQL), Model Description Languages (MDL) and complicated behaviours like state machines. Thus using AS to formalize such a hybrid language had considerable significance in exploring the adequacy and applicability of AS.

The major product of this work is the action semantics description of ALx, which is composed of three parts: the abstract syntax specification, the semantic function specification and the semantic entity specification. These three parts are respectively given in Appendix A, B and C. We used Chapter 4 to explain the action semantics of some unique constructs of ALx, such as object selection, link navigation and the run-to-completion process of state transition. Through the practice of formalizing ALx with AS, we found that AS is expressively adequate to formalize a heterogeneous language like ALx, and observed that the resulting action semantics of ALx is readable.

We expected that we could check the validity of the action semantics description on a versatile AS environment. Thus, a survey of existing AS tools was conducted to seek a suitable AS tool for this purpose. However, we found that none of the tools was able to test the ASD of ALx as expected. Therefore, we decided to build a translator which transforms xUML models into Java code based on the formal semantics of xUML so

that we could observe the behaviours of xUML models through running the generated Java code. In Chapter 5, we presented the conceptual design of the translator, which decomposes an xUML-to-Java translation into two sub-translations, and explained the main activities involved in the translation. In the conceptual design of the xUML-to-Java translator, we allocated the implementation of a significant part of the semantics to a Java library, which was intended to be precompiled and shared across various systems, to make the translator simpler and enhance the translation efficiency. The Java library explored some features of the Java language, such as static fields, inner classes and delegate pattern, to simulate links, relations, object-identity lists, object-link lists, state machines and inheritances in a manner suggested by the action semantics description of ALx.

We analyzed the features of the ALx-to-Java translator, finding that it was modeling-intensive and could be considered as a model transformer. We considered that MDA was the best potential approach to implement the translator. This is because MDA is born for model transformation and code generation, and it also provides modeling framework to specify models and metamodels, as well as the translation models. In Chapter 6, we analyzed the feasibility of applying UML, the core of MDA, to programming languages and gave rules of how to translate the abstract syntax and the static semantics of a programming language into metamodels in UML. The Eclipse implementation of MDA was adopted in developing the translator. Eclipse community provides a complete range of ready-to-use and open-source facilities to support MDA, such as the model transformation language (ATL), the model transformation engine (ATL Virtual Machine) and the modeling language (ECore). All of these makes the implementation of the translator very fast.

The practice of using MDA to implement the xUML-to-Java translator showed that MDA is a viable and excellent approach towards implementing language translators, especially in the situation where rapid development is expected. However, to further corroborate this advantage of MDA, in Chapter 7 we constructed another translator using a typical conventional approach and then made a comparison between two approaches. The analysis of the pros and cons of the two approaches showed that MDA has advantages over the conventional method in terms of code quality, such as better usability, better portability, looser component coupling, and less developing efforts, but it was still unclear whether MDA could apply to other language implementations which

have more capable back-ends.

8.2 Limitations, Discussion and Future work

8.2.1 Concurrency of UML

We currently do not explore describing the concurrency of UML, e.g., asynchronous calls to behaviors, co-existence of multiple *active* objects [93] each of which has its own thread, and asynchronous signal response, using communicative actions. This is because the AN-1 is not suitable, or at least not elegant, to describe some notions such as light-weight processes and threads, which probably share stores and necessitate synchronous communications.

Therefore, one future work is to cover the concurrency of UML using the newly developed AN-2, and update the xUML-to-Java translators correspondingly. We are confident that the newly developed AN-2 would make life easier in coping with concurrency since AN-2 allows agents to share and have global access to the storage.

8.2.2 MDA for Dynamic Semantics

The question still exists about whether MDA is power enough to deal with dynamic parts of programming languages and to implement more sophisticated language processors with more powerful back-ends, albeit the fact that we find that UML, the key element of MDA, can be easily harnessed to represent the static aspects of programming languages, and that MDA is a ready approach to implement the source-to-source language translator.

In our research, we have formally translated xUML into Java with MDA, but the essence of this effort is using MDA to represent the translational semantics of xUML in a translation language, and the dynamic semantics of xUML is actually denoted by the target language—Java. Hence, it can be said that in this process MDA deals little with the dynamic semantics of xUML, at least not directly.

Therefore, we would attempt to create a UML profile as a graphical formalism to specify the dynamic semantics of programming languages. If this work is accomplished, UML would be made complete in formalizing programming languages. That is, it can represent both static semantics and dynamic semantics; thus the language processors,

not limited to source-to-source translator, can be generated in MDA from the platform-independent UML description of programming languages. Another significance is that the readability and usability of formal semantics can be improved because the semantic descriptions are specified in terms of more intelligible graphical models rather than mysterious and sophisticated mathematical symbols. This will attract more people to use formal semantic technique. However, we must cope with such a risk that even a simple dynamic semantics requires overwhelming graphical representations.

8.2.3 Comparing MDA to Other Language-Implementing Approaches

We have only compared MDA to the conventional approach. The problem is, as mentioned, in addition to the conventional approach, there are a variety of language-prototyping approaches, such as the algebraic specification approach centered around the ASF formalism and Meta-Environment [101], the meta-language approach that adopts Kodiyak [42] to develop comprehensive translators, modular monadic semantics that allows the modular development of interpreters from semantic specifications by means of monad transformers [56], and a Prolog framework [94] enabling rapid prototyping activities on language processors with attribute grammars. Hence, it is necessary in the future to compare the MDA to such approaches.

8.2.4 Testing the ASD of ALx in an AS Tool

We have not experimented with the action semantics description of xUML (or ALx) using an AS environment or tool. The major reason for this is that all existing AS environments or tools are prototype systems, according to our experience, which can function well for small-scale AS descriptions, but for large-scale ones, it is very easy for the user to be entangled in usability problems. So one future work includes making an AS tool more usable and workable for large-scale action semantics to test our ASD of xUML.

8.2.5 Other Future Work

Some other interesting future work is listed here:

- Expanding xUML to a full executable UML. Now xUML is only a part of a full UML: it only consists of three diagrams, class diagrams, collaboration diagrams and state charts. Even these three kinds of diagrams themselves are not fully supported. Future work is necessitated to use the AS framework to specify a fully-fledged UML to further test the expressivity of the AS framework. Meanwhile, this work would contribute further to the resolution of the problem of formalizing UML.

- Using larger sample xUML models to test the xUML-to-Java translators. Even though we have tested the translators with some examples (see Chapter 7), the size and complexity of these samples are still considerably limited. In the future, we expect to build some realistic systems to test the translators.

- Translating xUML into more target languages. Currently, we only formalize the mapping rules from xUML to Java. It is desired that more target languages are supported.

- Integrating the xUML-to-Java translator into an xUML tool. We have made an attempt to implement a prototype xUML graphical authoring tool to better illustrate xUML. We expect to complete this and integrate the xUML-to-Java translator as the interpreter into this tool.

8.3 Concluding Remarks

In this thesis, we presented the action semantics of xUML, which is a toy executable UML, and two ways of implementing the xUML-to-Java translator, both of which are action-semantics-directed and are for the purpose of giving some assurance of the composed formal semantics. Additionally, we examined the applicability of MDA in specifying the static semantics, including the abstract syntax and contextual constraints, of programming languages, finding the cornerstone of the MDA, UML, is appropriate to specify the static semantics of programming languages. We also compared MDA with the conventional approach in building the xUML-to-Java translator, finding that the MDA has advantages of time and effort efficiency in prototyping the high-level source-to-source language translator. Although much has been investigated, even more remains

to be discovered and explored. It is our hope that this investigation could continue and the mentioned future work can attract the interest of the reader.

Thank you for reading.

Appendix A

Abstract Syntax of ALx

ALx/Abstract Syntax

A.1 Expressions

(1) Read-Attribute = ⟦ Identifier "." Identifier ⟧

(2) Call-Operation = ⟦ Expression "." Identifier "(" Arguments ")" ⟧

(3) Arguments = ⟨ Expression ⟨ "," Expression ⟩* ⟩?

(4) Expression = Literal | Identifier | "self" | "selected" |
⟦ Prefix-Operator Expression ⟧ |
⟦ Expression Infix-Operator Expression ⟧ |
⟦ Expression ("||" | "&&") Expression ⟧ |
⟦ "(" Expression ")" ⟧ |
Read-Attribute | Call-Operation.

(5) Prefix-Operator = "-" | "!" | "empty"

(6) Infix-Operator = "==" | "!=" | "<" | ">" | "<=" |
">=" | "+" | "−" | "*" | "/" | "%"

(7) Literal = Boolean-Literal | Integer-Literal | "null"

(8) Boolean-Literal = "true" | "false"

(9) Integer-Literal = ⟦ digit^{+} ⟧

(10) Identifier = ⟦ letter(letter | digit)* ⟧

A.2 Statements

needs: Declarations, Expressions.

Block

(1) Block-Statements = ⟦ Statement* ⟧

(2) Statement = ⟦ ";" ⟧ | ⟦ "{" Block-Statements "}" ⟧ |
⟦ Variable-Declaration ";" ⟧ |
⟦ Call-Operation ";" ⟧ |
⟦ (Assignment | Write-Attribute) ";" ⟧ |
⟦ (Object-Creation | Object-Deletion) ";" ⟧ |
⟦ (Link-Creation | Link-Deletion)";" ⟧ |
⟦ (Object-Selection | Link-Navigation) ";" ⟧ |
⟦ (Event-Generation | State-Transition) ";" ⟧ |
⟦ Object-Reclassification ";" ⟧ |
⟦ "return" Expression? ";" ⟧ |
⟦ "if" "(" Expression ")" Statement "else" Statement ⟧ |
⟦ "while" "(" Expression ")" Statement ⟧.

Assignments

(1) Assignment = ⟦ Identifier "=" Expression ⟧

Object Manipulation

(1) Write-Attribute = ⟦ Expression "." Identifier "=" Expression ⟧

(2) Object-Creation = ⟦ "create-object" Identifier "of" Identifier "("Arguments")" ⟧

(3) Object-Deletion = ⟦ "delete-object" Identifier ⟧

(4) Object-Reclassification = ⟦ "reclassify" Identifier Identifier "→ " Identifier ⟧

Link Manipulation

(1) Link-Creation = ⟦ "link" Identifier "→ " Identifier "(" Identifier ")" ⟧.

(2) Link-Deletion = ⟦ "unlink" Identifier "→ " Identifier "(" Identifier ")" ⟧.

Event Generation

(1) Event-Generation = ⟦ "send-event" Identifier "→ " Identifier ⟧

(2) State-Transition = ⟦ Expression ">>" Identifier⟧

Object Query

(1) Object-Selection = ⟦ "select-one" Identifier "of " Identifier ⟨ "(" Expression ")" ⟩? ⟧ |
⟦ "select-many" Identifier "of " Identifier ⟨ "(" Expression ")" ⟩? ⟧

(2) Link-Navigation = ⟦ Identifier "=" Identifier "→ " Identifier ⟨ "(" Expression ")" ⟩? ⟧ |
⟦ Identifier "=" Identifier "→ *" Identifier ⟨ "(" Expression ")" ⟩? ⟧

In the actual concrete syntax of ALx, we provide the following variants of object queries for coding conveniences. Note that they are in essence forms of expressions, and their formal semantics and tranlation rules are similar to above and thus ignored in this thesis.

(3) Object-Selection-Exp = ⟦ "select-one" Identifier ⟨ "(" Expression ")" ⟩? ⟧ |
⟦ "select-many" Identifier ⟨ "(" Expression ")" ⟩? ⟧

(4) Link-Navigation-Exp = ⟦ Expression "→ " Identifier ⟨ "(" Expression ")" ⟩? ⟧ |
⟦ Expression "→ *" Identifier ⟨ "(" Expression ")" ⟩? ⟧

(5) Expression = ... | Object-Selection-Exp | Link-Navigation-Exp.

A.3 Declarations

needs: Statements, Expressions.

Relation Declaration

(1) Relation-Declaration = ⟦ "relate" Identifier Identifier "→ " Identifier ";" ⟧

Class Declaration

(1) Class-Declaration = ⟦ "class" Identifier ⟨ "extends" Identifier ⟩? "{"
Field-Declaration*
Constructor-Declaration?
Method-Declaration*
State-Machine-Declaration? "}" ⟧

(2) Field-Declaration = ⟦ Type Identifier ";" ⟧

(3) Constructor-Declaration = ⟦ I: Identifier "(" F: Formal-Parameters ")"
"{" "super" "A: Arguments" ";" B: Block-Statements "}" ⟧

(4) Method-Declaration = ⟦ ("void" | Type) Identifier "(" Formal-Parameters ")" "{"
Block-Statements "}" ⟧.

State Machine Declaration

(1) State-Declaration = ⟦ "state" Identifier "{"
⟨ "entry" "{" Block-Statements "}" ⟩?
⟨ "exit" "{" Block-Statements "}" ⟩?
"}" ⟧

(2) State-Machine-Declaration = ⟦ "state-machine" "{"
State-Declaration+
"initial-state:" Identifier
"transition-table" "{" Transition-Entries "}"
"}" ⟧

(3) Transition-Entries = Transition-Entry ⟨ ";" Transition-Entry ⟩*

(4) Transition-Entry = Identifier "," Identifier "," Identifier

Event Declaration

(1) Event-Declaration = ⟦ "event" Identifier Identifier "→ " Identifier "}" ";" ⟧.

A.4 Misc

(1) Formal-Parameters = ⟨ Formal-Parameter ⟨ "," Formal-Parameter ⟩* ⟩?

(2) Formal-Parameter = ⟦ Type Identifier ⟧.

(3) Variable-Declaration = ⟦ Type Identifier ⟧ | ⟦ Type Identifier "=" Expression ⟧

(4) Type = "int" | "boolean" | "set" | Identifier | "set" "[" Type "]".

A.5 Model

needs: Expressions, Statements, Declarations.

(1) Executable-Model = ⟦ Class-Declaration*
Relation-Declaration*
Event-Declaration*
"main" "{" Block-Statements "}" ⟧

Appendix B

ALx/Semantic Functions

needs: ALx/Abstract Syntax, ALx/Semantic Entities, [Mosses 1992]/Action Notation.

We re-used, extended or adapted Watt's action semantics of JOOS [105] to compose our action semantics for ALx. The action semantics of JOOS demonstrates the main concepts of Java, including classes, inheritance, dynamic method selection and constructors. The reuse with some minor adaptations was made on the semantics of expressions, the common imperative statements, including assignment, loop constructs and call-operation, and some declarations such as class, method and field. The reusability of AS allowed us to be more focused on the unique constructs of ALx like link navigation, object query and state transition.

B.1 Expressions

introduces: evaluate _ , respectively evaluate _ , apply-prefix _ , apply-infix _ , the value of _ .

- evaluate _ :: Read-Attribute → action [giving a value | diverging | escaping]
  [using current bindings | current storage]

(1) evaluate ⟦ I_1: Identifier "." I_2: Identifier ⟧ =
  evaluate I_1 then
  | | give the value stored in ((field-variable-bindings the given object) at I_2)
  or
  | | check(the given reference is null)
  then
  | | escape with the null-reference-exception

- evaluate _ :: Call-Operation → action [giving a value | storing | diverging |
  escaping][using current bindings | current storage]

(2) evaluate ⟦ E: Expression "." I: Identifier "(" A: Arguments ")" ⟧ =
  | evaluate E and respectively evaluate A
  then

| | enact the application of the method I of the class of the given object#1 to
| | the given(object, value*)
| or
| | check the given reference#1 is null
| | then
| | escape with the null-reference-exception.

- respectively evaluate _ :: Arguments → action [giving value* | storing | diverging | escaping][using current bindings | current storage].

(3) respectively evaluate ⟨ ⟩ = give () .

(4) respectively evaluate E: Expression = evaluate E.

(5) respectively evaluate ⟨ E: Expression "," A: Arguments ⟩
= evaluate E and then respectively evaluate A.

- evaluate _ :: Expression → action [giving a value | storing | diverging | escaping][using current bindings | current storage] .

(6) evaluate L: Literal = give the value of L.

(7) evaluate I: Identifier = give the value stored in the variable bound to I.

(8) evaluate "self" = give the object bound to "self"

(9) evaluate "selected" = give the object bound to "selected"

(10) evaluate ⟦ O: Prefix-Operator E: Expression ⟧ =
evaluate E then apply-prefix O .

(11) evaluate ⟦ E_1: Expression O: Infix-Operator E_2: Expression ⟧ =
| evaluate E_1 and then evaluate E_2
then apply-infix O.

(12) evaluate ⟦ E_1: Expression "||" E_2: Expression ⟧ =
evaluate E_1 then
| | check (the given value is true) then give true
| or
| | check (the given value is false) then evaluate E_2

(13) evaluate ⟦ E_1: Expression "&&" E_2: Expression ⟧ =
evaluate E_1 then
| | check (the given value is true) then evaluate E_2
| or
| | check (the given value is false) then give false

(14) evaluate ⟦ "(" E: Expression ")" ⟧ = evaluate E .

- apply-prefix _ :: Prefix-Operator → action [giving a value][using the given value].

(15) apply-prefix "!" = give not (the given truth-value) .

(16) apply-prefix "-" = give the negation (the given true-value) .

- apply-infix _ :: Infix-Operator → action [giving a value][using the given value2].

(17) apply-infix "==" = give (the given value#1 is the given value#2).

(18) apply-infix "!=" = give not (the given value#1 is the given value#2).

(19) apply-infix "<" = give (the given value#1 is less than the given value#2).

(20) apply-infix “>” = give not (the given value#1 is less than the given value#2)

(21) apply-infix “<=” = not (apply-infix “>”)

(22) apply-infix “>=” = not (apply-infix “<”)

(23) apply-infix “+” = give the sum of (the given integer#1, the given integer#2)

(24) apply-infix “−” = give the difference of (the given integer#1, the given integer#2)

(25) apply-infix “∗” =give the product of (the given integer#1, the given integer#2)

(26) apply-infix “/” =give the integer-quotient of (the given integer#1, the given integer#2).

(27) apply-infix “%” =give the integer-remainder(the given integer#1, the given integer#2).

- the value of _ :: Literal → value.

(28) The semantics of Literal is intuitive and is omitted here.

B.2 Statements

Block

- execute _ :: Block-Statements → action [storing | diverging | escaping]
  [using current bindings | current storage].

(1) execute ⟦ S: Statement* ⟧ =
  furthermore execute S
  hence complete;

- execute _ :: Statement* → action [binding | storing | diverging | escaping]
  [using current bindings | current storage]

(2) execute ⟨ ⟩ = complete.

(3) execute ⟦ “;” ⟧ = complete.

(4) execute ⟦ “{” B: Block-Statements “}” ⟧ = execute B.

(5) execute ⟦ V: Variable-Declaration “;” ⟧ = elaborate V.

(6) execute ⟦ S: (Assignment | Write-Attribute) “,”⟧ = execute S.

(7) execute ⟦ O: (Object-Creation | Object-Deletion) “;” ⟧ = execute O.

(8) execute ⟦ L: (Link-Creation | Link-Deletion | Call-Operation)“;” ⟧ = execute L.

(9) execute ⟦ O: (Object-Selection | Link-Navigation) “;” ⟧ = execute O.

(10) execute ⟦ E: (Event-Generation | State-Transition) “;” ⟧ = execute E.

(11) execute ⟦ O: Object-Reclassification “;” ⟧ = execute O.

(12) execute ⟦ “return” “;” ⟧ = escape with the return of ().

(13) execute ⟦“return” E: Expression “;” ⟧ =
  evaluate E then
  escape with the return of the given value .

(14) execute ⟦ "if" "(" E: Expression ")" S_1: Statement "else" S_2: Statement ⟧=
evaluate E then
| | check the given value is true) then execute S_1
| or
| | check (the given value is false) then execute S_2

(15) execute ⟦"while" "(" E: Expression ")" S: Statement ⟧ =
unfolding
| evaluate E then
| | | check (the given value is true) then execute S then unfold
| | or
| | | check (the given value is false) then complete .

(16) execute ⟨ S_1: Statement S_2: Statement$^+$ ⟩ =
execute S_1 before execute S_2.

Assignments

- execute _ :: Assignment → action [giving a value | storing | diverging | escaping][using current bindings | current storage].

(1) execute ⟦ I: Identifier "=" E: Expression ⟧ =
| evaluate E
then
| store the given value in the variable bound to I .

Object Manipulation

- execute _ :: Write-Attribute → action [diverging | escaping]
[using current bindings | current storage]

(1) execute ⟦ I_1: Identifier "." I_2: Identifier "=" E: Expression ⟧=
| evaluate I_1 and evaluate E
then
| | store the given value#2 in the variable bound to
| | ((field-variable-bindings the given object#1) at I_2)
| or
| | | check(the given reference#1 is null)
| | then
| | | escape with the null-reference-exception.

- execute _ :: Object-Creation → action [storing | diverging | escaping | binding]
[using current bindings | current storage]

(2) execute ⟦"create-object" I_1: Identifier "of" I_2: Identifier "(" A: Arguments")"⟧ =
| allocate an object of the class bound to the class-token of I_2 and respectively evaluate A
then
| enact the application of the constructor of the class bound to I_2 to
| the given (object, value*) and bind I_1 to the given object#1 and
| recursively add the given object#1 to class (the given object#1).

- recursively add _ to _ :: object, class → action [storing | diverging]
[using current bindings | current storage]

(3) recursively add O: object to C: class =
| | give the object-list stored in the cell bound to
| | the object-list-token of (class-token C) then
| | | store concatenation (the given object-list, the list of O)
| | | to the cell bound to the object-list token of (class-token C)
| and give (superclass C)
then
| | check (the given tuple is()) and then complete
| or
| | check (not(the given tuple is()) and then recursively add O to the given class

- execute _ :: Object-Deletion → action [storing | diverging | escaping | binding][using current bindings | current storage]

(4) execute ⟦ "delete-object" I: Identifier ⟧ =
| give the object bound to I
then
| | un-instantiate field-variable-bindings (it)
| and
| | unbind I
| and
| | recursively remove (the given object) from the class (class it)

- recursively remove _ from the class _ :: object, class →
action [storing | diverging] [using current bindings | current storage]

(5) recursively remove O: object from the class C: class =
| | | remove O from the object-list stored in the cell bound to
| | | the object-list-token of (class-token C)
| | then
| | | store the given object-list in the cell bound to
| | | the object-list-token of (class -token C)
| and
| | give (superclass C)
then
| | check (the given tuple is ()) and then complete
| or
| | check (not(the given tuple is ())) and then
| | recursively remove O from the class the given class and its superclasses.

- execute _ :: Object-Reclassification → action [storing | diverging | escaping] [using current bindings | current storage]

(6) execute ⟦ "reclassify" I_1: Identify I_2: Identify "→ " I_3: Identity ⟧=

```
    | give the class bound to I2 and
    | give the class bound to I3 and
    | give the object bount to I1
  then
    | give the type-variable-bindings of the given class#1 and
    | give the type-variable-bindings of the given class #2 and
    | give the field-variable-bindings of the object bound to I1 and
    | get the super class of ( the given class#1, the given class#2 ) and
    | give the set stored in the field-variable-bindings of the given object#1
    | at "_ LinkRecord" and
    | give the variable yielded by
    | the field-variable-bindings of the given object#1 at "_ LinkRecord"
  then
    | | | selectively remove links in the given set#5
    | | | except the given class#4 and
    | | | give the given variable #6
    | | then
    | | | store the given set#1 in the gvien variable#2
    | and
    | | | give the given variable-bindings#3 and
    | | | give the intersection of
    | | | (the mapped-set of the given variable-bindings#3,
    | | | the mapped-set of the type-variable-bindings of the given class#4)
    | | then
    | | | un-instantiate variable-bindings#1 restricted to the given set#2
    | and
    | | | give the given variable-bindings and
    | | | give the intersection of
    | | | ( the mapped-set of the type- variable-bindings of the given class#2,
    | | | the mapped-set of the type-variable-bindings of the given class#4 )
    | | then
    | | | give the given variable-bindings and
    | | | instantiate the field-type-bindings of the class bound to I3 restricted to the given set .
  then
    | give the disjoint-union of ( the given variable-bindings#1,
    | the given variable-bindings#2)
  then
    | store the object of ( the class bound to I2,
    | the given variable-bindings, the identity of the object ) stored in the variable bound to I
```

- get the super class of (_ , _) :: class, class → action [giving a class | diverging]

(7) get the super class of (C_1: class, C_2: class) =

```
    give superclass C1 then
      | check ( the given class is in the superclasses of C2 ) and then
      | give the given class
    or
      | check ( not (the given class is in the superclasses of C2)) and then
      | the super class of ( the given class, C2).
```

- selectively remove links in _ except _ :: set, class → action [giving a set | diverging]

(8) selectively remove links in S: set except C: class =

choose a link [in S] then
| give the relation of the given link and regive
then
| give the associated classes of the given relation#1
| and give the superclasses of C
| and give selectively remove links in the
| intersection (S, the set of the given link#2) except C and
| give the given link#2
then
| | check(either(the given class#1 is in the given set#3,
| | the given class#2 is in the given set#3)) and then
| | give the given set#4
| or
| | check(not(either(the given class#1 is in the given set#3,
| | the given class#2 is in the given set #3))) and then
| | give the disjoint-union (the set of the given link#5, the given set #4).

Link Manipulation

- execute _ :: Link-Creation → action [storing | diverging] [using current bindings | current storage]

(1) execute ⟦ "link" I_1: Identifier "→ " I_2: Identifier "(" I_3: Identifier ")" ⟧ =
| allocate a cell then
| give the link of (I_3, (the object stored in the cell bound to I_1,
| the object stored in the cell bound to I_2), the given cell)
then
| add the given link to the object stored in the cell bound to I_1 and
| add the given link to the object stored in the cell bound to I_2.

- add _ to _ :: link, object → action [storing | diverging]
[using current bindings | current storage]

(2) add L:link to O : Object = give the field-variable-bindings of O
then give (the given variable-bindings at "_ LinkRecord")
then store disjoint-union of (the set stored in the given variable,
the set of L) in the given variable.

- execute _ :: Link-Deletion → action [storing | diverging]
[using current bindings | current storage]

(3) execute ⟦ "unlink" I_1: Identifier "→ " I_2: Identifier "(" R: relation ")" ⟧ =
| give the set stored in (the field-variable-bindings of I_1 at "_ LinkRecord")
| and
| give the set stored in (the field-varaible-bindings of I_2 at "_ LinkRecord")
then
| remove link (the given set#1, the given set#2, R)
then
| store the given set#1 in (the field-variable-bindings of I_1 at "_ LinkRecord")
| and
| store the given set#2 in (the field-variable-bindings of I_2 at "_ LinkRecord")

- remove link (_ , _ , _) :: set, set, relation → action
[giving a tuple | diverging] [using current storage]

(4) remove link s_1: set, s_2: set, r: relation = choose an link [in s_1] then
| | check (both (the given link is in s_2,
| | the given link is an instance of r)) and then
| | give the intersection (s_1, the set of the given link) and
| | give the intersection (s_2, the set of the given link)
| or
| | check(not (both(the given link is in s_2,
| | the given link is an instance of r)))
| | and then
| | remove link (the intersection (s_1, the set of the given link),
| | intersection(s_2, the set of the given link), r) and
| | give the given link
| | then
| | give the disjoint-union of (the set of the given link#3,
| | the given set#1) and
| | give the disjoint-union of (the set of the given link#3, the given set#2)

Event Generation

- execute _ :: Event-Generation → action [storing | diverging] [using current bindings | current storage]

(1) execute ⟦ "send-event" I_1: Identifier "→ " I_2: Identifier ⟧ =
give the object stored in the cell bound to I_2 then
| | get the current state of the given object and regive then
| | enact the application of the exit-action of the given state#1 to the given object#2
| and then
| | get the destination state of the given object when the event-token of I_1 and regive
| | then
| | | set the current state of the given object#2 to the given state#1 and then
| | | enact the application of the entry-action of the given state#1 to the given object#2.

- execute _ :: State-Transition → action [storing | diverging] [using current bindings | current storage]

(2) execute ⟦ E: Expression ">>" I: Identifier ⟧ =
evaluate E then
| | get the current state of the given object and regive then
| | enact the application of the exit-action of the given state#1 to the given object#2
| and then
| | give (state-bindings of the state-machine of (the class of the given object#2))
| | at the state-token of I and regive
| | then
| | | set the current state of the given object#2 to the given state#1 and then
| | | enact the application of the entry-action of the given state#1 to the given object#2.

Object Query

- execute _ :: Object-Selection → action [storing | diverging]
[using current bindings | current storage]

(1) execute ⟦ "select-many" I_1: Identifier "of" I_2: Identifier "(" E: Expression ")" ⟧ =
| select instances in (the object-list stored in the cell bound to
| the object-list-token of the class-token of I_2) satisfying E
then
| store the given set to the variable bound to I_1.

(2) execute ⟦ "select-many" I_1: Identifier "of" I_2: Identifier ⟧ =
| exhaust instances in (the object-list stored in the cell bound to
| the object-list-token of I_2)
then
| store the given set to the variable bound to I_1.

(3) execute ⟦ "select-one" I_1: Identifier "of" I_2: Identifier ⟧ =
exhaust instances in (the object-list stored in the cell bound to the object-list-token of I_2)
then
| choose an object [in the given set] then store the given object to the cell bound to I_1.

(4) execute ⟦ "select-one" I_1: Identifier "of" I_2: Identifier "(" E: Expression ")" ⟧ =
| select instances in (the object-list stored in the cell bound to
| the object-list-token of I_2) satisfying E
then
| choose an object [in the given set] then store the given object to the variable bound to I_1.

- select instances in _ satisfying _ :: object-list, Expression → action [giving a set | diverging] [using current bindings | current storage]

(5) select instances in I: object-list satisfying E: Expression =
| check (I is empty-list) and then give empty-set
or
| check (not (I is empty-list)) and then
| | give (head I) then bind "selected" to the given object
| | thence
| | | | evaluate E and select instances in (tail I) satisfying E and
| | | | give the given object
| | | then
| | | | | check(the given truth-value#1 is true) and then
| | | | | give disjoint-union(set of(the given object#3), the given set#2)
| | | | or
| | | | | check(not(the given truth-value#1 is true) and give the given set#2.

- exhaust instances in _ :: object-list | link-list → action [giving a set | diverging]

(6) exhaust instances in I: object-list | link-list =
| check (I is empty-list) and then give empty-set
or
| check (not(I is empty-list)) and then
| give disjoint-union(head I, instances in (tail I)).

- execute _ :: Link-Navigation → action [storing | diverging]
[using current bindings | current storage]

(7) execute ⟦ I_1: Identifier "=" I_2: Identifier "→ *" I_3: Identifier ⟧ =
| give (the object stored in the variable bound to I_2) and give the relation bound to I_3
then (regive and get the links from the given object#1)
then
| exhaust the linked objects of the given object#1
| from the given set#3 related by the given relation#2
then store the given set to the variable bound to I_1.

(8) execute ⟦ I_1: Identifier "=" I_2: Identifier "→ *" I_3: Identifier "(" E: Expression ")" ⟧ =
| give (the object stored in the variable bound to I_2) and give the relation bound to I_3
then (regive and get the links from the given object#1)
then
| exhaust the linked objects of the given object#1
| from the given set#3 related by the given relation#2
then pick objects in the given set satisfying E
then store the given set to the variable bound to I_1.

- pick objects in _ satisfying _ :: set, Expression → action [giving a set | diverging] [using current binding | current storage]

(9) pick objects in s: set satisfying E: Expression =
| check (s is empty-set) and then give empty-set
or
| check (not(I is empty-set)) and then
| | choose an object [in s] then (regive and bind "selected" to the given object)
| | thence
| | | evaluate E and and give the given object#1 and
| | | pick objects in (the intersection of (the given object#1, s)) satisfying E
| | | then
| | | | check(the given truth-value#1 is true) and then
| | | | give disjoint-union(set of(the given object#3), the given set#2)
| | | or
| | | | check(not(the given truth-value#1 is true) and give the given set#2.

(10) execute ⟦ I_1: Identifier "=" I_2: Identifier "→ " I_3: Identifier "(" E: Expression ")" ⟧ =
| give (the object stored in the variable bound to I_2) and give the relation bound to I_3
then (regive and get the links from the given object#1)
then
| exhaust the linked objects of the given object#1
| | from the given set#3 related by the given relation#2
then pick objects in the given set satisfying E
then choose an object [in the given set]
then store the given object to the variable bound to I_1.

(11) execute ⟦ I_1: Identifier "=" I_2: Identifier "→ " I_3: Identifier ⟧ =
| give (the object stored in the variable bound to I_2) and give the relation bound to I_3
then (regive and get the links from the given object#1)
then
| exhaust the linked objects of the given object#1
| | from the given set#3 related by the given relation#2
then
| choose an object[in the given set] then store the given set to the variable bound to I_1.

- exhaust the linked objects of _ from _ related by _ :: object, set, relation → action [giving a set | diverging]

(12) exhaust the linked objects of o: object from s: set related by r: relation =
| check (s is empty-set) and then give empty-set
or
| check (not (s is empty-set)) and then
| | choose a link [in s] then
| | | exhaust the linked object of o from the intersection of (s, the set of the given link)
| | | and give the given link
| | then
| | | check (the given link#2 is an instance of r) and then
| | | give disjoint-union (the set of the object
| | | linked with o by the given link#2, the given set#1)
| | or
| | | check (not(the given link is an instance of r) and then give the given set#1.

B.3 Declarations

needs: Statements, Expressions

introduces: elaborate _ , the type-bindings of _ , the method-bindings of _ , respectively formally bind _ , formally bind _ , the type denoted by _ .

Relation Declaration

- elaborate _ :: Relation-Declaration* → action [bindings] [using current bindings].

(1) elaborate ⟨ ⟩ = complete.

(2) elaborate ⟦ "relate" I_1: Identifier I_2: Identifier "→ " I_3: Identifier ";"⟧ =
bind I_1 to the relation of (I_1, the class bound to I_2, the class bound to I_3).

(3) elaborate ⟨ R_1: Relation-Declaration, R_2: Relation-Declaration$^+$ ⟩ =
elaborate R_1 before elaborate R_2.

Class Declaration

- elaborate _ :: Class-Declaration →
action [binding | storing][using current bindings | current storage].

(1) elaborate ⟦"class" I_1: Identifier "extends" I_2: Identifier "{"
F: Field-Declaration* C: Constructor-Declaration
M: Method-Declaration* "}"
S: State-Machine-Declaration⟧ =
| recursively bind the class-token of I_1 to
| the class of (the type-bindings of F, the method-bindings of M,
| the constructor of C , the state-machine of S, the class bound to the class-token of I_2).
and
| allocate a cell then
| | store an empty-list in it and bind the object-list-token of the class-token of I_1 to it.

(2) the semantics of the declaration of those classes that has no state machine or super class is akin to the above, so it is ignored here to save space.

- elaborate _ :: Class-Declaration* → action [binding] [using current bindings]

(3) elaborate ⟨ ⟩ = complete .

(4) elaborate⟨ C_1: Class-Declaration C_2: Class-Declaration$^+$ ⟩ =
elaborate C_1 before elaborate C_2 .

- the type-bindings of _ :: Field-Declaration* → type-bindings

(5) the type-bindings of ⟨ ⟩ = the empty-map .

(6) the type-bindings of ⟨ F_1: Field-Declaration F_2: Field-Declaration$^+$ ⟩ =
the disjoint-union of (the type-bindings of F_1, the type-bindings of F_2)

- the constructor of _ :: Constructor-Declaration → yielder [of a constructor][using current bindings].

(7) the constructor of ⟦I: Identifier "(" F: Formal-Parameters ")"
"{" "super" "A: Arguments" ";" B: Block-Statements "}"⟧ =
the closure of the abstraction of

furthermore
bind "self" to the given object#1 and
produce the field-variable-bindings of the given object#1
hence
furthermore
give the rest of the given (object, value*) then
respectively formally bind F
hence
given the given object#1 and respectively evaluate A.
then enact the application of
the constructor of the superclass of the class bound to I
to the given (object, value*)
and then
execute B
trap a return then complete.

(8) the constructor of $\langle\ \rangle$ =
the closure of the abstraction of complete.

- the method-bindings of _ :: Method-Declaration* → yielder
[of method-bindings][using current bindings]

(9) the method-bindings of $\langle\ \rangle$ = the empty-map.

(10) the method-bindings of ⟦ T("void" | Type) I: Identifier
"(" F: Formal-Parameters ")" "{" B: Block-Statements "}" ⟧ =
the map of I to the closure of the abstraction of
furthermore
bind "self" to the given object#1 and
produce the field-variable-bindings of the given object#1
hence
furthermore
give the rest to the given (object, value*) then
respectively formally bind F
hence
execute B
trap a return then give the returned-value of it.

(11) the method-bindings of $\langle$ M_1: Method-Declaration M_2: Method-Declaration$^+$ $\rangle$
= the disjoint-union of (the method-bindings of M_1, the method-bindings of M_2).

State Machine Declaration

- the state-machine of _ :: State-Machine-Declaration → yielder [of a state-machine]

(1) The state-machine of ⟦ "state machine" "{"
S: State-Declaration$^+$
"initial state: " I: Identifier ";"
"transition table: " T: Transition-Entries ";"
"}" ⟧ =
state-machine of (state-token of I, the transition-table of T, the state-bindings of S).

- the transition-table of _ :: Transition-Entries → yielder [of a transition-table]

(2) the transition-table of T: Transition-Entry = the transition-entry of T .

(3) the transition-table of $\langle$ T_1: Transition-Entry ";" T_2: Transition-Entries $\rangle$ =
disjoint-union (the transition-entry of T_1, the transition-table of T_2).

- the transition-entry of _ :: Transition-Entry → yielder [transition-table]

(4) the transition-entry of _ :: ⟦ "(" I_1: Identifier "," I_2: Identifier "," I_3: Identifier ")" ⟧ =
the map of (the state-token of I_1, the event-token of I_2) to the state-token of I_3.

- the state-bindings of _ :: State-Declaration$^+$ → yielder [of state-bindings]

(5) the state-bindings of ⟦ "state" I: Identifier "{"
E_1: ⟨ "entry action" "{" Block-Statements "}" ⟩$^?$
E_2: ⟨ "exit action" "{" Block-statements "}" ⟩$^?$ ⟧ =
the map of the state-token of I to state of (the state-token of I, the entry-action of E_1, the exit-action of E_2).

(6) the state-bindings of ⟦ S_1: State-Declaration S_2: State-Declaration$^+$ ⟧ =
disjoint-union (the state-bindings of S_1, the state-bindings of S_2)

- the entry-action of :: ⟨ "entry action" "{" Block-Statements "}" ⟩$^?$ →
yielder [of a entry-action$^?$]

(7) the entry-action of ⟨ ⟩ = ();

(8) the entry-action of ⟨ "entry action" "{" B: Block-Statements "}" ⟩ =
the closure of the abstraction of
| furthermore
| | bind "self" to the given object#1 and
| | produce the field-variable-bindings of the given object#1
| hence
| | execute B
| | trap a return then complete.

- the exit-action of :: ⟨ "exit action" "{" Block-statements "}" ⟩$^?$ → yielder [of a exit-action$^?$]

(9) the exit-action of ⟨ ⟩ = ();

(10) the exit-action of ⟨ "exit action" "{" B: Block-Statements "}" ⟩ =
the closure of the abstraction of
| furthermore
| | bind "self" to the given object#1 and
| | produce the field-variable-bindings of the given object#1
| hence
| | execute B
| | trap a return then complete.

Event Declaration

- elaborate _ :: Event-Declaration → action [bindings] [using current bindings].

(1) The semantics of Event-Declaration is not defined here because events are simplified to pure strings (or just labels). I.e., the Event-Declarations occurring in models are just for better illustration of the system.

B.4 Misc

- respectively formally bind _ :: Formal-Parameters → action [binding | storing] [using the given value* | current storage].

(1) respectively formally bind $\langle\ \rangle$ = complete.

(2) respectively formally bind F: Formal-Parameter = formally bind F.

(3) respectively formally bind $\langle\ F_1$: Formal-Parameter "," F_2: Formal-Parameters $\rangle$=
    | give the first of the given value$^+$ then formally bind F_1
    and
    | give the rest of the given value$^+$ then respectively formally bind F_2.

- formally bind _ :: Formal-Parameter → action [binding | storing]
    [using the given value | current storage].

(4) formally bind ⟦ T: Type I: Identifier ⟧ =
    allocate a variable initialised to the given value then
    bind I to it.

- elaborate _ :: Variable-Declaration → action [binding — storing]
    [using current storage]

(5) elaborate ⟦ T: Type I: Identifier ⟧ =
    allocate a variable then
    bind I to it.

(6) elaborate ⟦ T: Type I: Identifier "=" E: Expression ⟧ =
    evaluate E then allocate a variable initialised to the given value
    then bind I to the given variable.

- the type denoted by _ :: Type → type.

(7) the type denoted by "boolean" = the boolean-type.

(8) the type denoted by "int" = the integer-type.

(9) the type denoted by I: Identifier = the reference-type.

(10) the type denoted by "set" "[" Type "]" = the set-type.

B.5 Model

- run _ :: Executable-Model → action [storing] [using current storage].

(1) run ⟦ Executable-Model = ⟦ C: Class-Declaration*
    R: Relation-Declaration*
    E: Event-Declaration*
    "main" "{" B: Block-Statements "}" ⟧ =
    elaborate C before elaborate R hence execute B.

Appendix C

ALx/Semantic Entities

needs: [Mosses 1992] /(Data Notation, Action Notation).

Data

needs: Values, Variables, Types, Classes, Objects, Escapes, Tokens, Recording Lists, Instances

- datum = value | variable | type | class | method | constructor | type-bindings | variable-bindings | method-bindings | token | reason-for-escape (disjoint).
- token = (letter,(letter | digit)*).
- bindable = class | instance | variable .
- storable = value | object-list | state-token .

Tokens

needs: Identifiers

introduces: class-token, class-token of _ , object-list-token, the object-list-token of _ , event-token, the event-token of _ ; state-token, the state-token of _ , token;

- token = class-token | state-token | event-token | object-list-token | Identifier
- class-token = class-token of Identifier
- class-token of _ :: Identifier → class-token (total, injective)
- the object-list-token of _:: class-token → object-list-token (total, injective)
- event-token = the event-token of Identifier
- the event-token of _ :: Identifier → event-token (total, injective)
- state-token = the state-token of Identifier
- the state-token of _ :: Identifier → state-token (total, injective)

Values

needs: Objects

introduces: value

- value = truth-value | integer | reference | set ($disjoint$).

Recording Lists

needs: Objects, Links.

introduces: object-list, link-list, remove _ from _ .

- object-list = flat-list [object]
- link-list = link-list [link]
- remove _ from _ :: yielder [of (object | link)] , yielder [of (object-list | link-list] → action [give an object-list | link-list] (total)

(1) remove o_1: object from o_2 : object-list =
| check (o_1 is empty) and then give empty-list
or
| check (not(o_1 is empty)) and then
| | give (head o_2)
| then
| | | check (o_1 is the given object) and then give (tail o_2)
| | or
| | | check (not (o_1 is the given object)) and then
| | | give (concatenation (the list of (the given object-list), the object-list yielded by
| | | remove o_1 from (tail o_2))).

(2) remove l_1: link from l_2 : link-list =
| check (l_1 is empty) and then give empty-list
or
| check (not(l_1 is empty)) and then
| | give (head l_2)
| then
| | | check (l_1 is the given object) and then give (tail l_2)
| | or
| | | check (not (l_1 is the given object)) and then
| | | give (concatenation (the list of (the given object-list), the object-list yielded by
| | | remove l_1 from (tail l_2))).

Variables

needs: Values.

introduces: variable, allocate a variable initialised to _.

- variable = cell.
- allocate a variable initialised to _ :: yielder [of a value] →
  action [storing — giving a variable] [using current storage]

(1) allocate a variable initialised to v: yielder [of a value] =
    | allocate a variable and give v
    then
    | store the given value#2 in the given variable#1 and
    | give the given variable #1.

Types

needs: Values, Objects.

introduces: type, boolean-type, integer-type, reference-type, default-value _ .

- type = boolean-type | integer-type | reference-type | set-type (individual).
- default-value _ :: type → value (total).

(1) default-value (boolean-type) = false.

(2) default-value (integer-type) = 0.

(3) default-value (reference-type) = null.

(4) default-value (set-type) = empty-set.

Instances

needs: Objects, Links, Classes, Relations.

introduces: instance, null, reference, _ is an instance of _ .

- instance = object | link (disjoint).
- null: reference.
- reference= null | instance (disjoint).
- _ is an instance of _ :: reference, (class | relation) = truth-value.

(1) null is an instance of c: class | relation = false.

(2) o: object is an instance of r: relation = false.

(3) l: link is an instance of c: class = false.

(4) l: link = the link of (r: relation, (o_1: object, o_2: object), identity) ⇒ l is an instance of r = true.

(5) o: object is an instance of c: class = c is in superclasses (class o).

(6) r: reference is an instance of c: (class | relation) = false (default).

Classes

needs: Data, Types,

introduces: class, class-token _ , type-bindings, method-bindings, method, constructor, class of _ , field-type-bindings _ , method-bindings _ , constructor _ , superclass _ , superclasses _ , method _ of _, state-machine _.

- class = class of (class-token, type-bindings, method-bindings, constructor, state-machine$^?$, class$^?$).
- type-bindings = map [token to type].
- method-bindings = map [token to method].
- method = abstraction [giving a value$^?$ | storing | diverging | escaping]
  [using the given (object, value*) — current storage].
- class-token _:: class → class-token (total, injective)
- constructor _ :: = abstraction [storing | diverging | escaping]
  [using the given (object,value*) | current storage].
- class of _ :: (type-bindings, method-bindings, constructor, class$^?$) → class (total, injective).
- field-type-bindings_ :: class → type-bindings (total).
- method-bindings _ :: class → method-bindings (total).
- constructor _ :: class → constructor (total).
- state-machine _ :: class → state-machine$^?$ (total)
- superclass _ :: class → class$^?$ (total).
- superclasses _ :: class → set [class] (total)

(1) c = class of (ct: class-token, t: type-bindings, m: method-bindings, k: constructor, s: state-machine$^?$) ⇒

  (1) class-token $c = ct$;
  (2) field-type-bindings c = disjoint-union (the map of "_ LinkRecord" to set, t) ;
  (3) method-bindings $c = m$;
  (4) constructor $c = k$;
  (5) state-machine $c = s$;
  (6) superclass c =();
  (7) superclasses c = set of c.

(2) c = class(ct: class-token, t: type-bindings, m: method-bindings, k: constructor, s: state-machine$^?$, c_1: class) ⇒

  (1) class-token $c = ct$;
  (2) field-type-bindings c =disjoint-union (t, field-type-bindings c_1);
  (3) method-bindings c = overlay (m, method-bindings c_1);
  (4) constructor $c = k$;
  (5) state-machine $c = s$;
  (6) superclass $c = c_1$;
  (7) superclasses c = union (set of c, superclasses c_1).

- method _ of _ :: token, class → method (partial).

(3) method t: token of c: class = method-bindings c at t.

Objects

needs: Data, Variables, Types, Classes, Instances

introduces: object, variable-bindings, object of _ , class _ ,field-variable-bindings _ , _ is _ , allocate an object of _ , identity, identity _ , instantiate_ .

- object = object of (class, variable-bindings, identity).
- variable-bindings = map [token to variable].
- identity = cell.
- object of _ :: (class, variable-bindings, identity) → object (total, injective).
- class _ :: object → class (total).
- field-variable-bindings _ :: object → variable-bindings (total).
- identity _:: object → identity (total).

(1) o = object of (c: class, v: variable-bindings, i : identity) ⇒

  (1) class o = c;
  (2) field-variable-bindings o = v;
  (3) identity o = i.
  (4) i = identity o: object.

- _ is _ :: reference, reference → truth-value (total, commutative).

  (1) null is null = true.
  (2) null is o: object = false.
  (3) o_1: object is o_2: object = identity o_1 is identity o_2.

- allocate an object of _ :: yielder [of a class] → action ⟦ storing | giving an object] [using current storage | current bindings]

(2) allocate an object of c: yielder [of a class] =
  | instantiate the field-type-bindings of c and
  | allocate an identity and initiate state of c
  then
  | give the object of (the class yielded by c , disjoint-union
  | (the given variable-bindings#1, the given
  | variable-bindings#3), the given identity #2)

- instantiate _ :: yielder [of type-bindings] → action [storing | giving variable-bindings][using current storage].

(3) instantiate t: yielder [of type-bindings] =
  | check (t is the empty-map) and then
  | give the empty-map
  or
  | | give t and
  | | choose a token [in the mapped-set of t]
  | then
  | | instantiate (the given type-bindings #1
  | | omitting the set of the given token #2) and
  | | give the given token #2 and
  | | allocate a variable initialised to the default-value of the type yielded by
  | | (the given type-bindings #1 at the given token #2)
  | then give the disjoint-union of (the given variable-bindings #1,
  | the map of the given toke #2 to the given variable #3).

- initiate state of _ :: class →
  action [storing | giving variable-bindings] [using current storage].

(4) Initiate state of c: class =
  give the state-machine c then
  | check (not the given state-machine is ()) then
  | | give the initial-state-token of state-machine c then
  | | allocate a variable initialised to the given state-token then
  | | give the map of ("_ CurrentState" to the given state-token)
  | or
  | | check (the given state-machine is ()) then give empty-map.

- un-instantiate _ :: yielder [of variable-bindings] → action [storing] [using current storage].

(5) un-instantiate v: yielder [of variable-bindings] =
  | check (v is empty-map) and then
  | complete
  or
  | | give v and
  | | choose a token [in the mapped-set of v]
  | then
  | | un-instantiate (the given variable-bindings #1 omitting the set of the given token #2) and
  | | unreserve the variable yielded by the v at the given token#2

- get the links of _ :: yielder [of object] → action [giving a set | storing] [using current storage].

(6) get the links of y : object =
  give the set stored in (the type-variable-bindings of the object yielded by y at "_LinkRecord").

- get current state of _ :: yielder [of object] →
  action [giving a state | storing][using current storage] .

(7) get current state of o: yielder =
  | give state-bindings of the state-machine of (the class of o) and
  | give the state-token stored in
  | (the type-variable-bindings of o at "_CurrentState")
  then
  | give the given state-bindings#1 at the given state-token#2

- get the destination state of _ when _ :: object, event-token →
  action [giving a state | storing] [using current storage]

(8) get the destination state of o: object when e: event-token =
  | give the state-machine of (class o) and
  | give the state-token stored in
  | (the type-variable-bindings of o at "_CurrentState")
  then
  | give (the transition-table of the given state-machine#1 at
  | (the given state-token #2, e) and give the given state-machine#1
  then
  | give the state-bindings of (the given state-machine#2) at the given state-token#1.

- set the current state of _ to _ :: object, state → action [storing] [using current storage]

(9) set the current state of o: object to s: state =
  | give the type-variable-bindings of o at "_ CurrentState" and
  | give state-token s
  then store the given state-token#2 in the given variable#1.

Relations

needs: Data, Types

introduces: relation, relation of _ , the related classes of _ , the name of the relation _ , the associated classes of _ .

- relation = relation of (Identifier, class, class).
- relation of _ :: (Identifier, class, class) → relation (total, injective).
- the related classes of _:: relation → (class, class) (total).
- the name of the relation _ :: relation → Identifier (total).
- the associated classes of _ :: relation → (class, class) (total).

(1) r = relation of (i: Identifier, c_1: class, c_2: class) ⇒

  (1) the related classes of r = (c_1, c_2);

  (2) the name of the relation $r = i$.

  (3) the associated classes of r = (c_1, c_2)

Links

needs: Relations, Objects

introduces: link, link of _ , the linked objects of _ , identity_ , relation _ , the other object than_ of _ .

- link = link of (relation, (object, object), identity). (total, injective)
- the linked objects of _ :: link → (object, object) (total)
- identity _:: link → identity (total).
- the other object than _ of _ :: link, object → object (partial)

(1) the other object than o_1: object of l: link = when l is the link of (r: relation, (o_1, o_2: object), i: identity) then o_2

(2) the other object than o_2: object of l: link = when l is the link of (r: relation, (o_1: object, o_2), i: identity) then o_1.

(3) l = link of (r: relation, (o_1: object, o_2: object), i: identity) ⇒

  (1) the linked objects of l = (o_1, o_2);

  (2) identity $l = i$;

  (3) relation $l = r$.

States

needs: Tokens, Objects

introduces: state, state-token _, entry-action, exit-action, entry-action _ , exit-action _ .

- state = state of (state-token, entry-action$^?$, exit-action$^?$)
- entry-action = abstraction[storing | diverging | escaping]
  [using the given object | current storage]
- exit-action = abstraction [storing | diverging | escaping]
  [using the given object | current storage]
- exit-action _ :: state → exit-action$^?$ (total)
- entry-action _ :: state → entry-action$^?$ (total)
- state-token _ :: state → state-token (total)

(1) s = state of (st: state-token, en: entry-action$^?$, ex: exit-action$^?$) ⇒

  (1) state-token $s = st$;

  (2) entry-action $s = en$;

  (3) exit-action $s = ex$;

State-Machines

needs: States, Tokens

introduces: state-machine, event-tokens of _ , initial-state-token, initial-state-token of _ , state-bindings, state-bindings of _ , state-machine of _ ,

- state-machine = state-machine of (initial-state, transition-table, state-bindings)
- initial-state-token = state-token
- transition-table = map [(state-token, event-token) to state-token]
- state-bindings = map [state-token to state]
- state-machine of _ :: (initial-state-token, transition-table, state-bindings) → state-machine (total, injective)
- transition-table of _ :: state-machine → transition-table (total)
- state-bindings of _ :: state-machine → state-bindings (total)
- initial-state-token of _ :: state-machine → initial-state-token (total).
- exit-action of _ in _ :: state-token, state-machine → exit-action (partial)

(1) exit-action of st: state-token in sm: state-machine = state-bindings of sm at state-token

- entry-action of _ in _ :: state-token, state-machine → entry-action (partial)

(2) exit-action of st: state-token in sm: state-machine = state-bindings of sm at state-token

- destination state token from _ when _ in _ :: state-token, event-token, state-machine → state-token (partial)

(3) destination state token from st: state-token when e: event-token in sm: state-machine = transition-table of sm at (st, e).

(4) sm = state-machine of (c: class, e: event-tokens, i: initial-state, t: transition-table, s: state-bindings) ⇒

  (1) initial-state of sm = i;

  (2) transition-table of sm = t;

  (3) state-bindings of sm = s;

Escapes

needs: Values.

introduces: return, return of _ , returned-value _ , exception, null-reference-exception, reson-for-escape, _ trap _ then _ .

- return = return of value?.
- return of _:: value? → return (total, injective).
- Returned-value _ :: return → value? (total).
- returned-value (return of v: value?) = v.
- exception = null-reference-exception | □ (individual).
- reason-for-escape = return | exception(disjoint).
- _ trap _ then _ :: action [escaping], reason-for-escape, action → action.

(1) a_1: action trap ($r \leq$ reason-for-escape)then a_2: action = a_1 trap
  | check (there is a given r) and then a_2
  or
  | check (there is a given r) and then escape.

Appendix D

xUML Metamodel

This appendix presents the xUML metamodel which is designed to be extensible so that more diagrams can be incorporated. The elements in the xUML metamodel are grouped into the following packages:

- *fundamental package*, which defines basic abstract metaclasses, such as Element, NamedElement, TypedElement, etc., that deal with naming and typing of elements.
- *class package*, which defines the structure of xUML classes. Generally, it shows that an xUML class may have operations and properties, and the operations may be specified using ALx Block-Statement.
- *relational package*, where the core class is the Association that is a kind of Relationship. Association specializes Classifier; this means that associations may have instances — links.
- *state-machine package*, which defines the structure of state machines. A state machine has various states, each of which has entry and exit behaviours.
- *collaboration package.*This package shows there are two kinds of role, User role and Class role, that may be involved in a particular collaboration diagram. The user role is generally the initiator of system execution. Two kinds of event are supported in xUML: SendSignalEvent and CallEvent.
- *behavioural package*, which shows two kinds of behaviours are supported, operations and state machines.

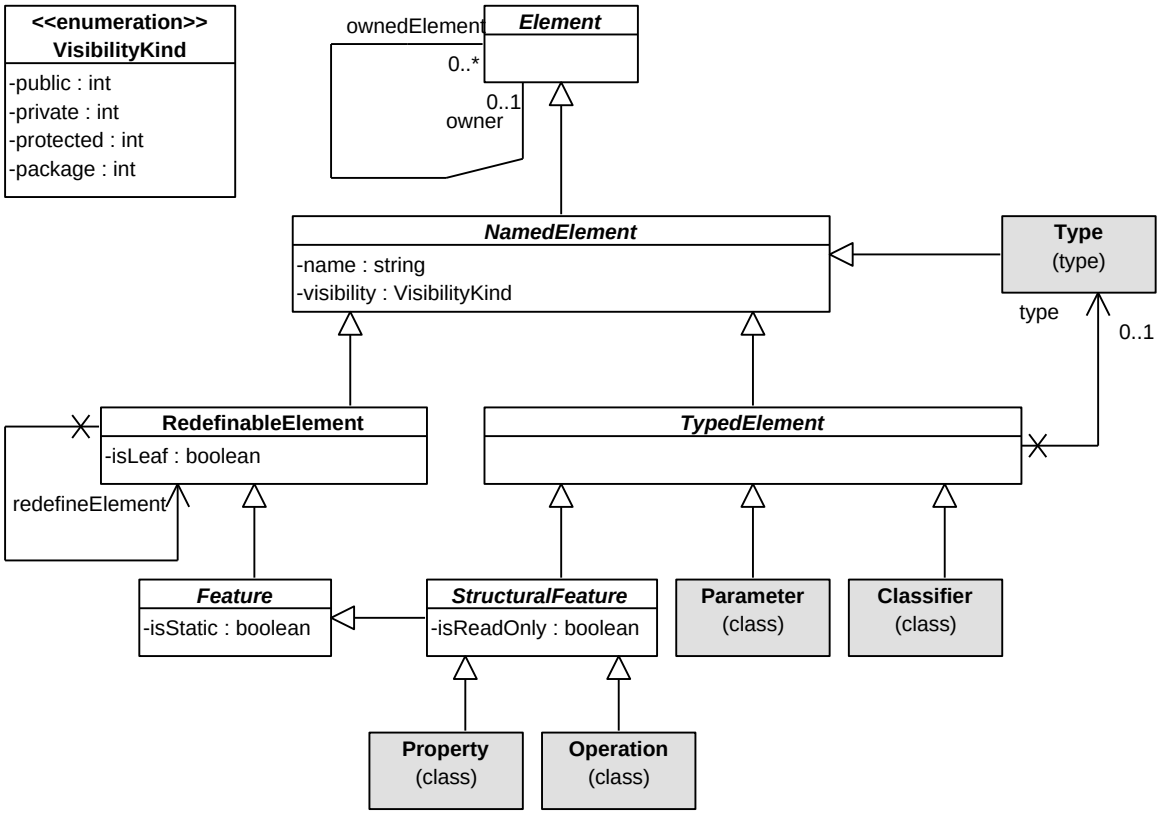


Figure D.1: Fundamental package of xUML metamodel

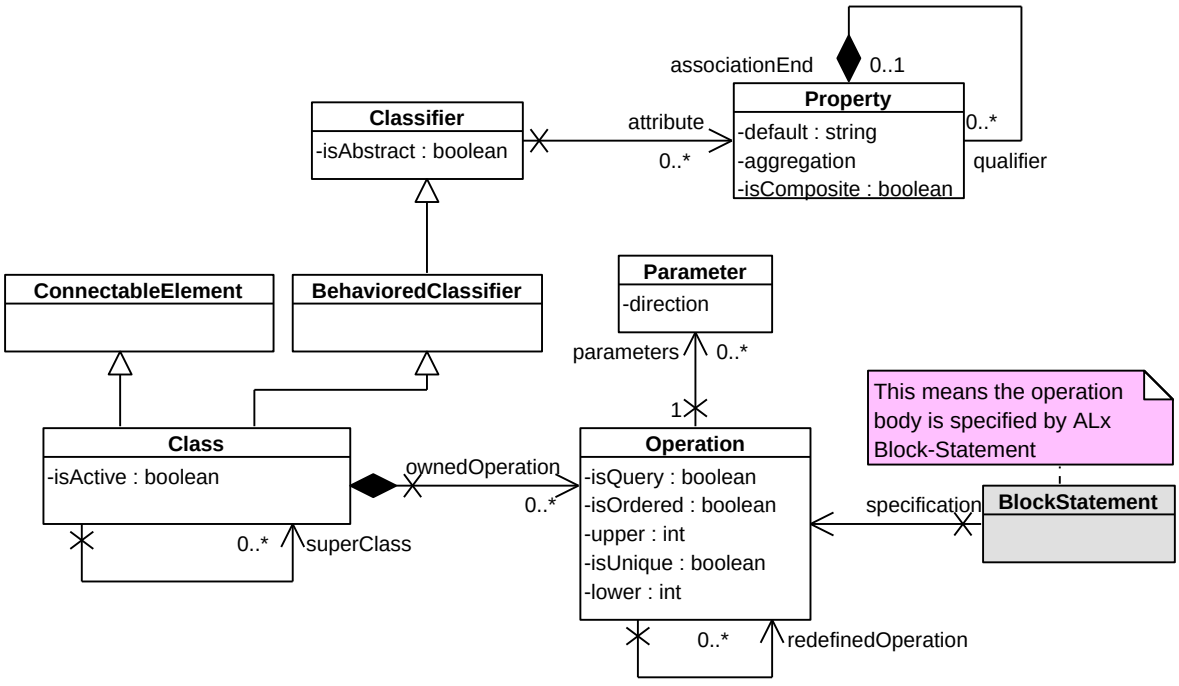


Figure D.2: Class package of xUML metamodel

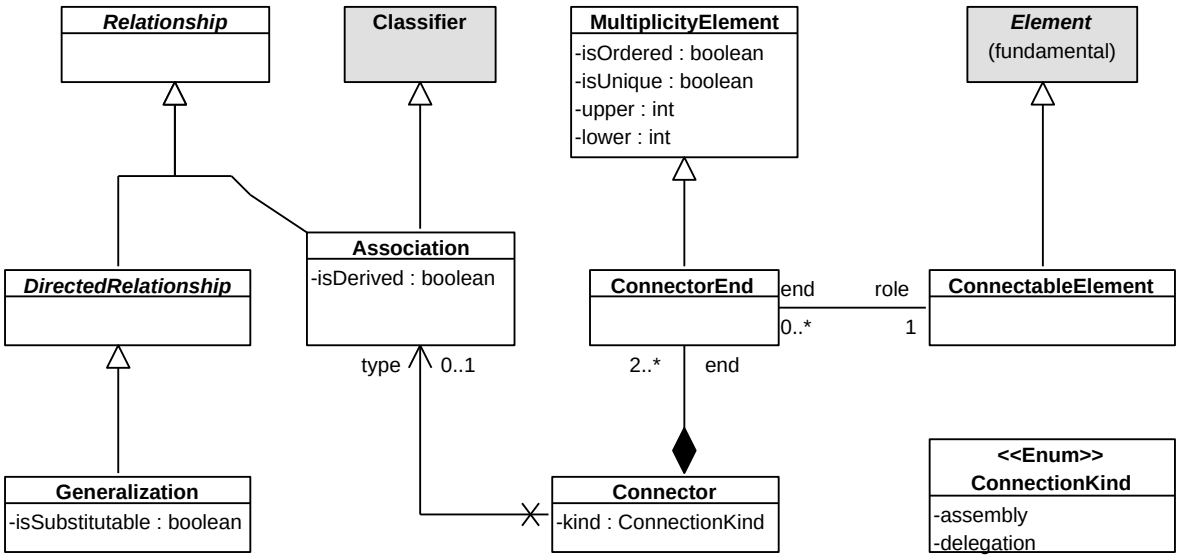


Figure D.3: Relational package of xUML metamodel

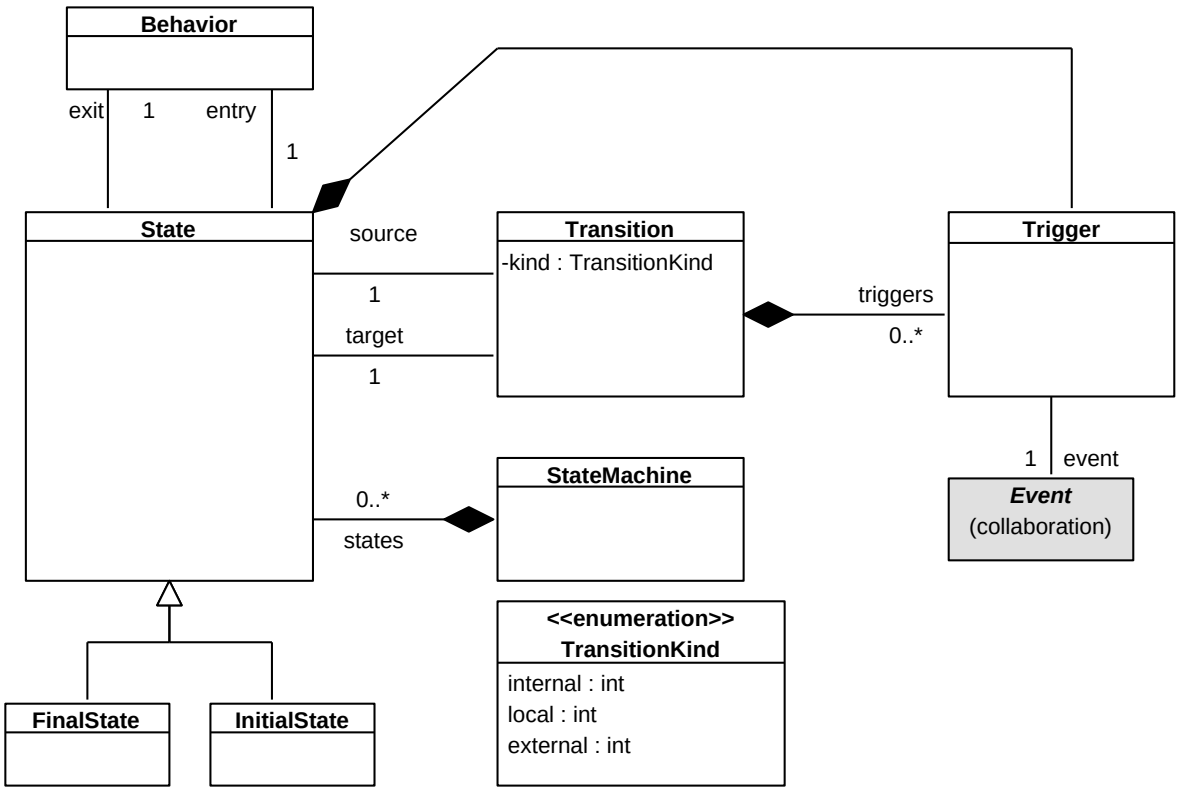


Figure D.4: State-machine package of xUML metamodel

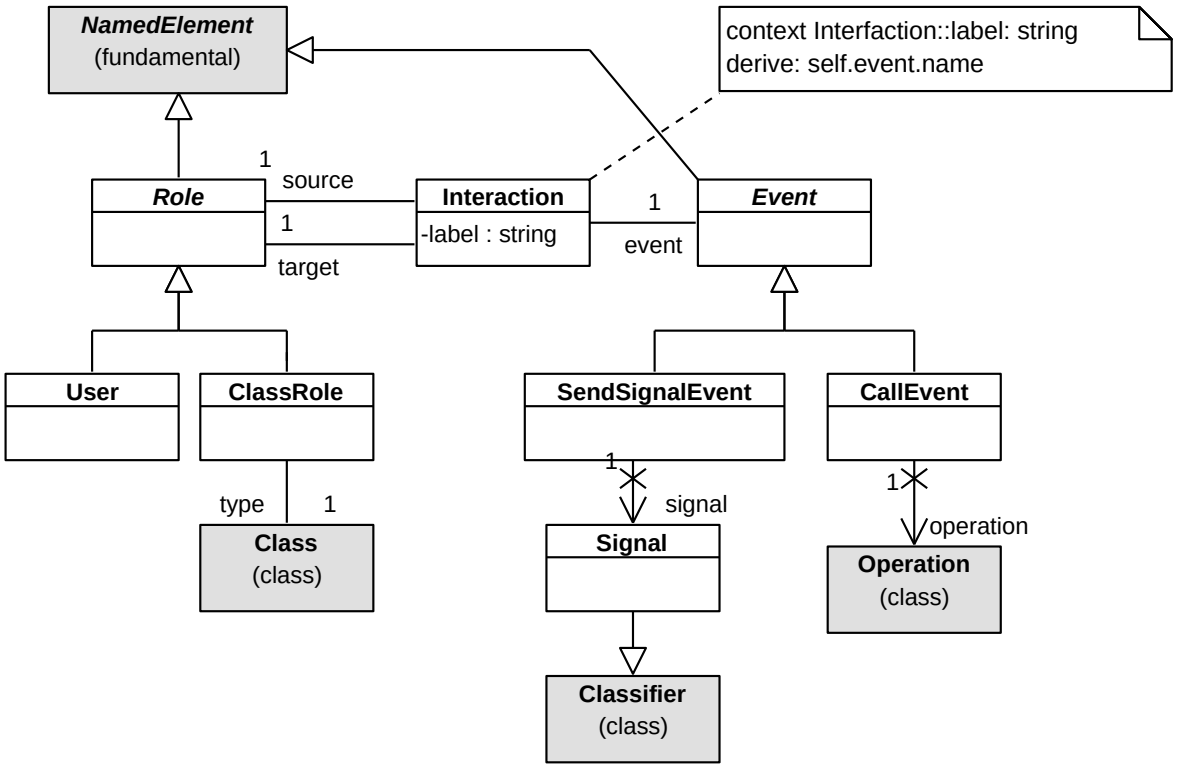


Figure D.5: Collaboration package of xUML metamodel

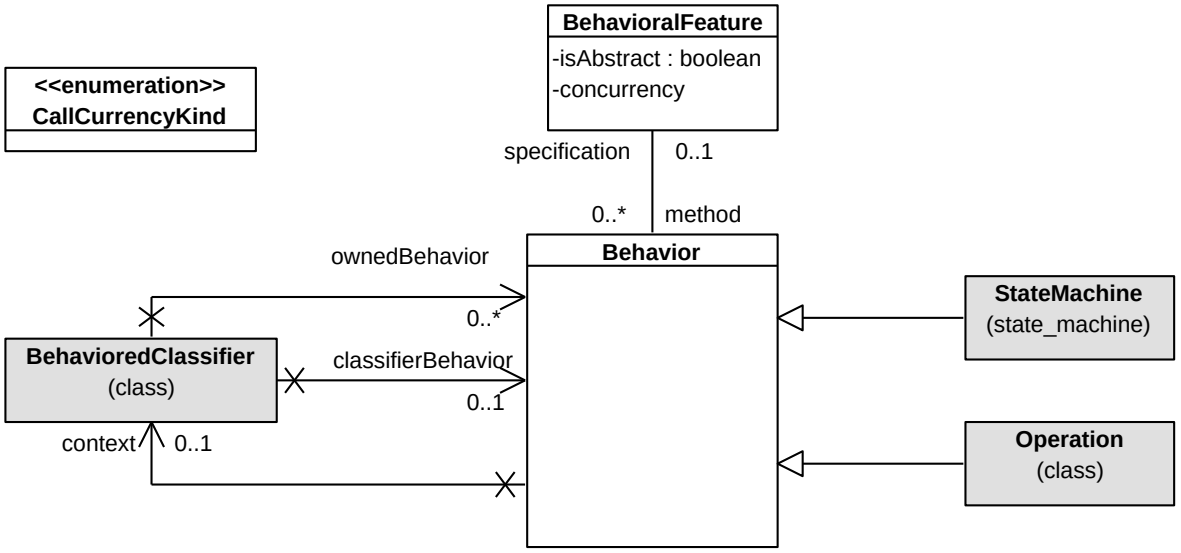


Figure D.6: Behavioural package of xUML metamodel

Appendix E

ALx Metamodel

ALx consists of a part for the common functionality available in the present action languages, and a model-describing part, which is a textual correspondent of the graphical xUML. In Chapter 6 we discuss the applicability of UML to describe the static aspects of programming languages and enumerate a set of rules for translating grammars to UML notations. We follow these rules to convert the ALx AST (provided in Appendix A) to UML representations, shown in the following figures.

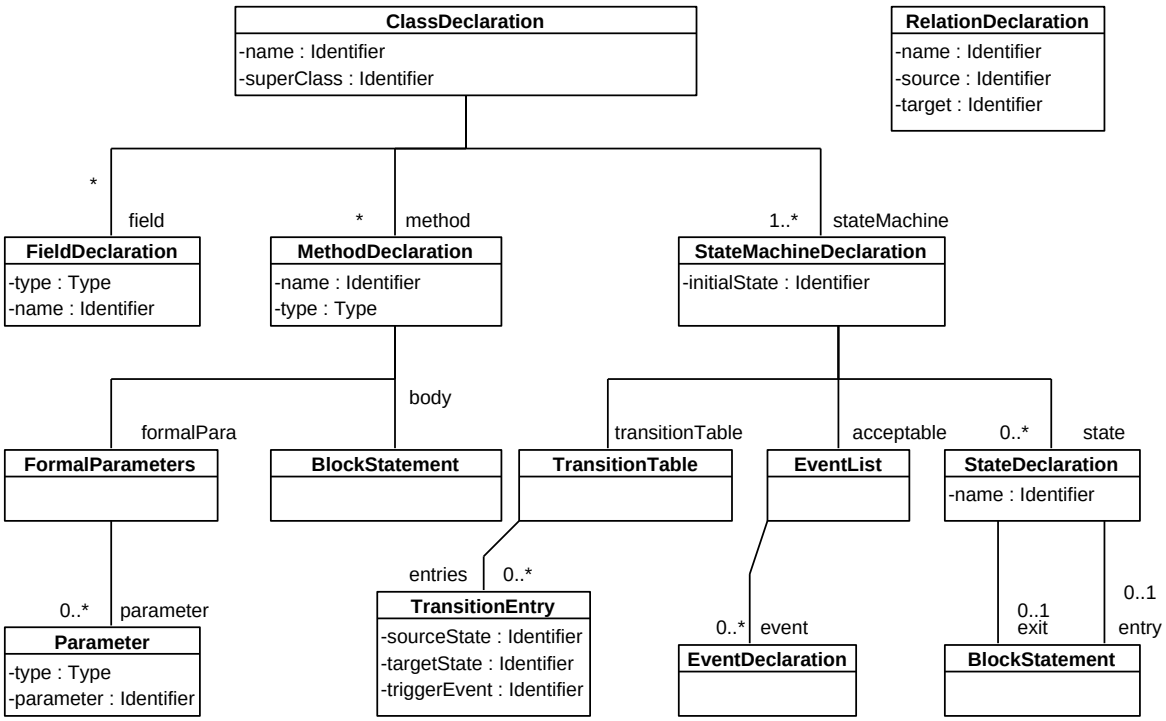


Figure E.1: Declarations of ALx

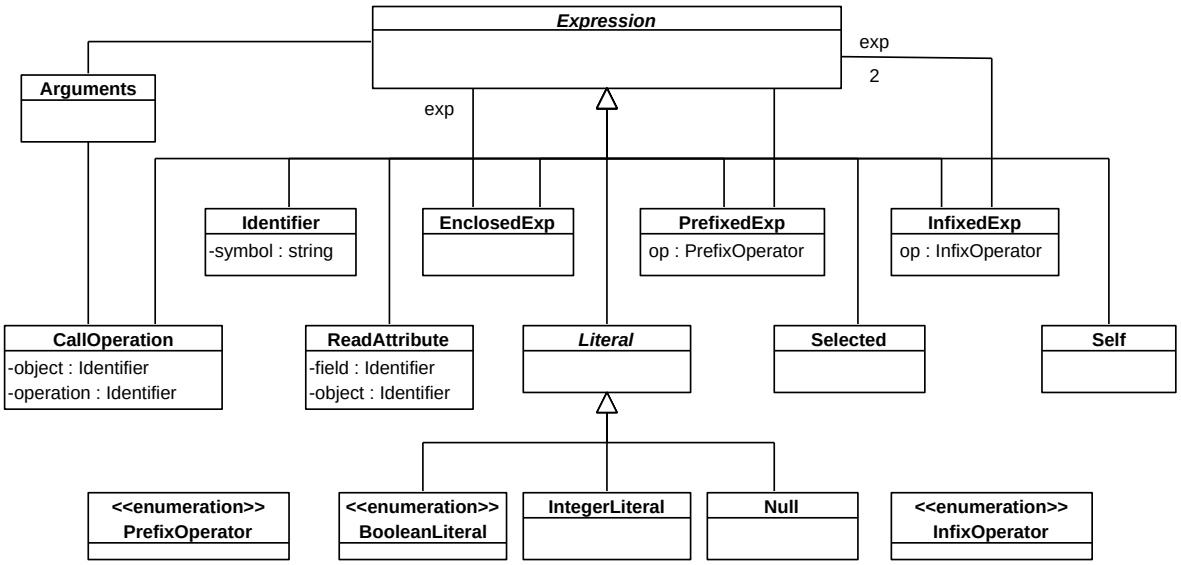


Figure E.2: Expressions of ALx

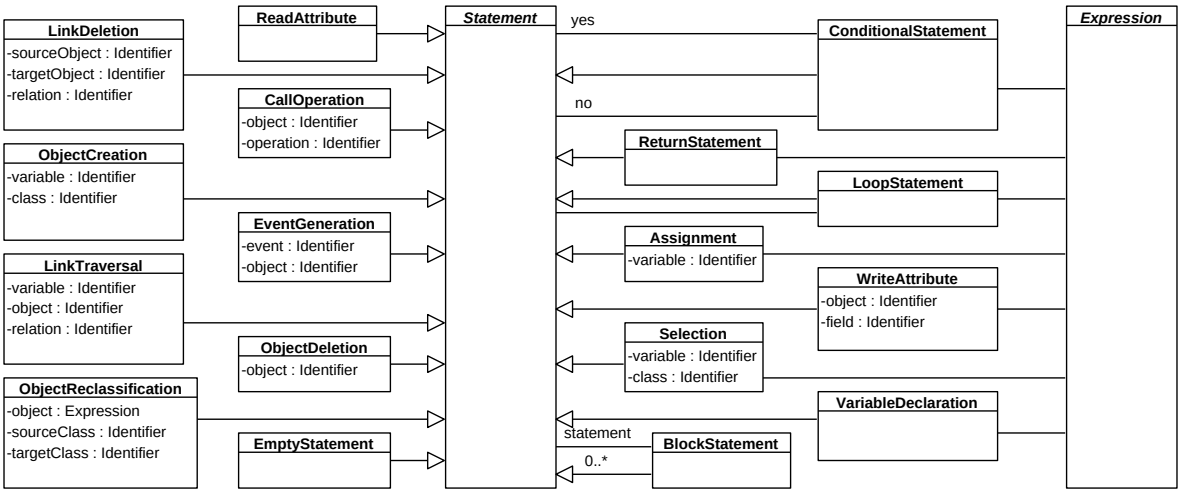


Figure E.3: Statements of ALx

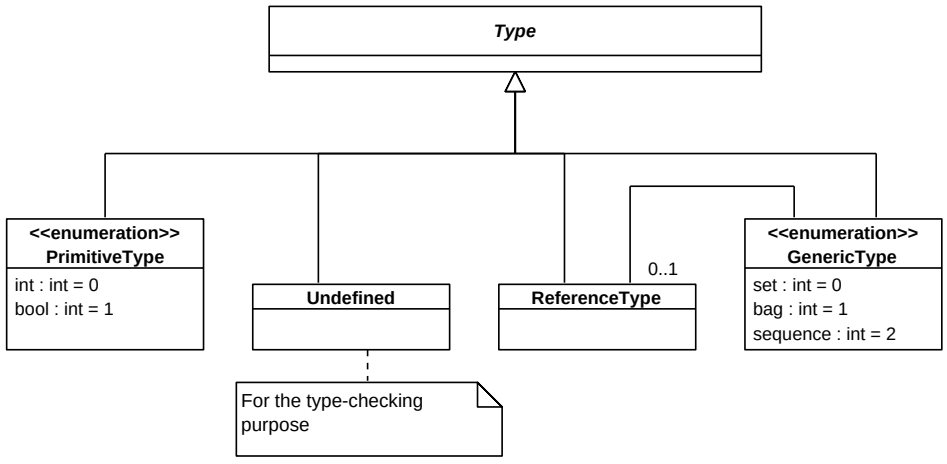


Figure E.4: Type system of ALx.

Appendix F

MiniJava Metamodel

MiniJava is an executable subset of the Java language, intended to be the target language of the xUML-to-Java translator. While making MiniJava as small as possible to ease the implementation, we assure that the expressivity of MiniJava is no less than that of xUML or ALx. Furthermore, MiniJava code is assured to be recognizable to the language implementations of the standard Java, such as the Sun Java compiler or Java Virtual Machine.

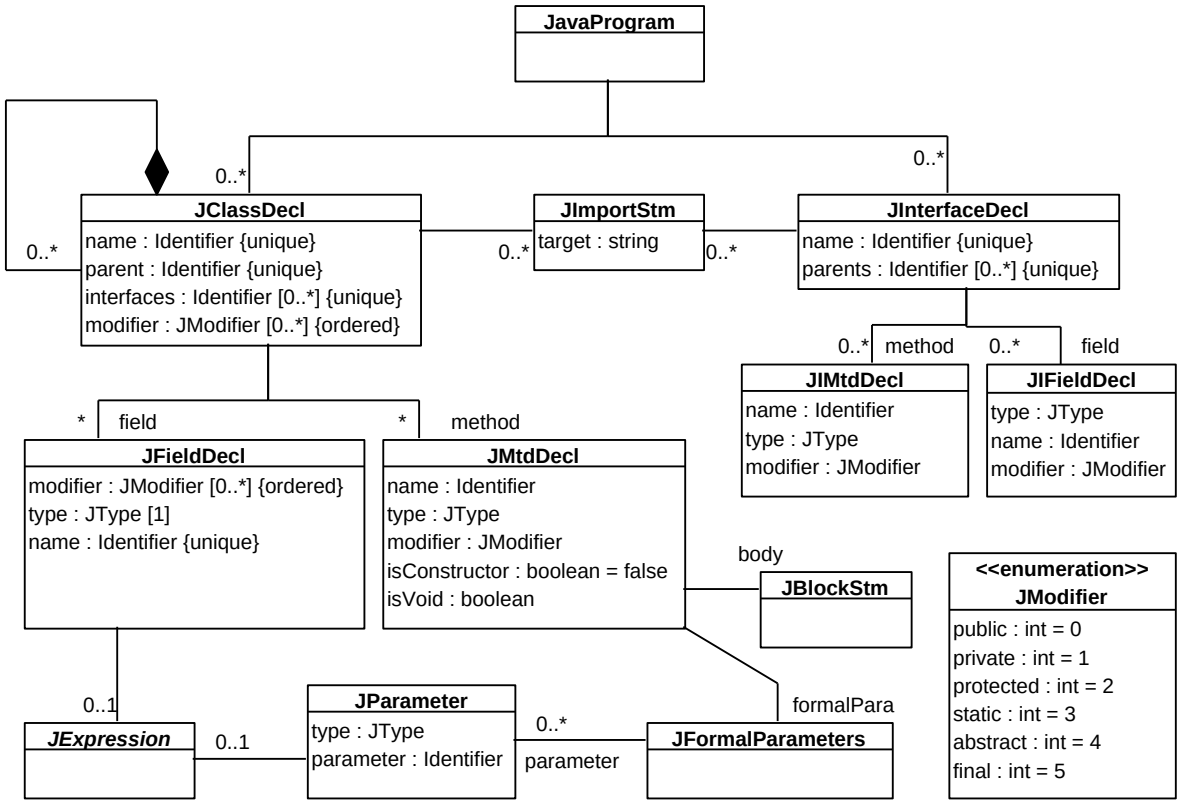


Figure F.1: Declarations of MiniJava

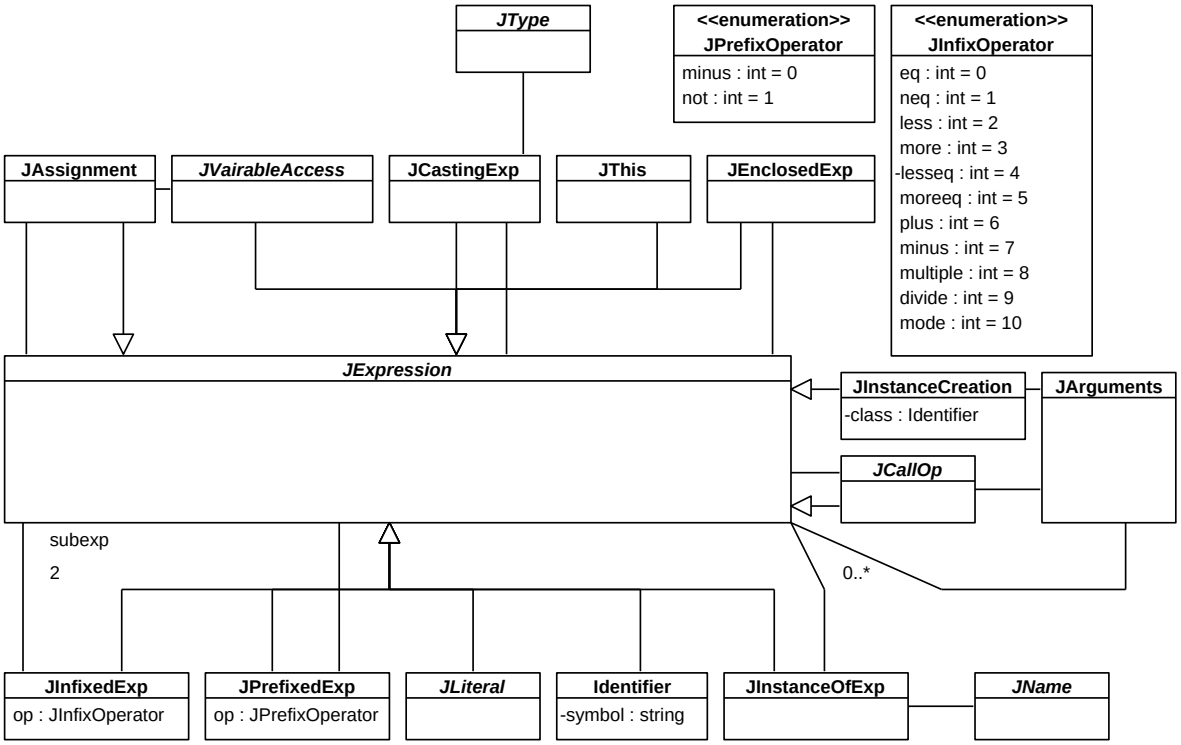


Figure F.2: Expressions of MiniJava

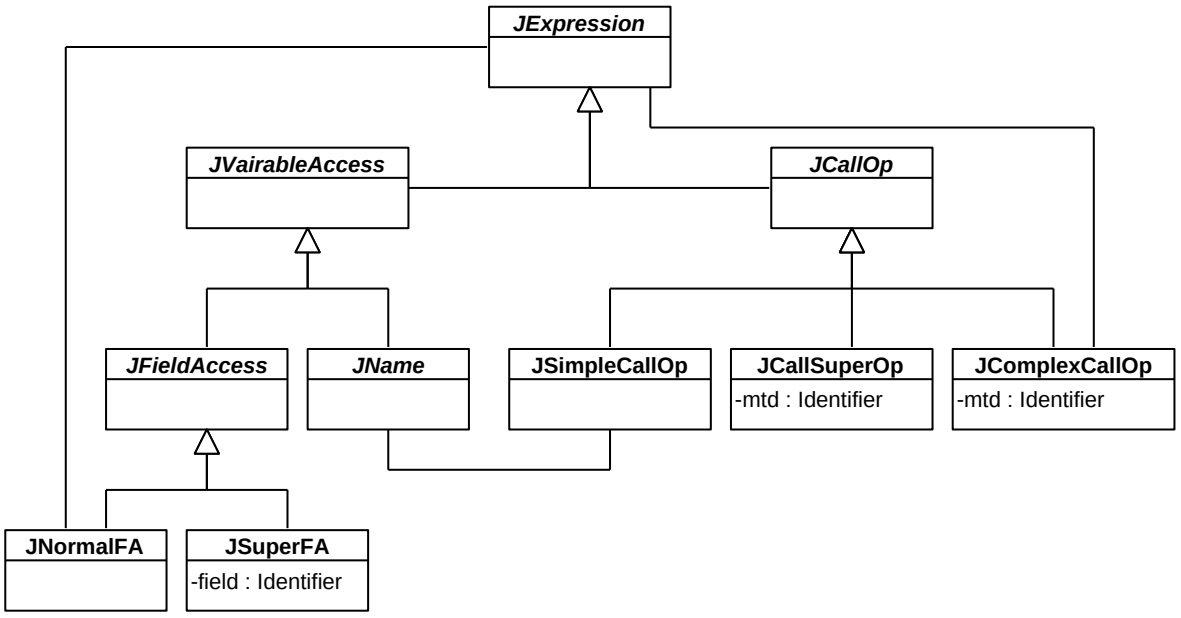


Figure F.3: Variable access of MiniJava.

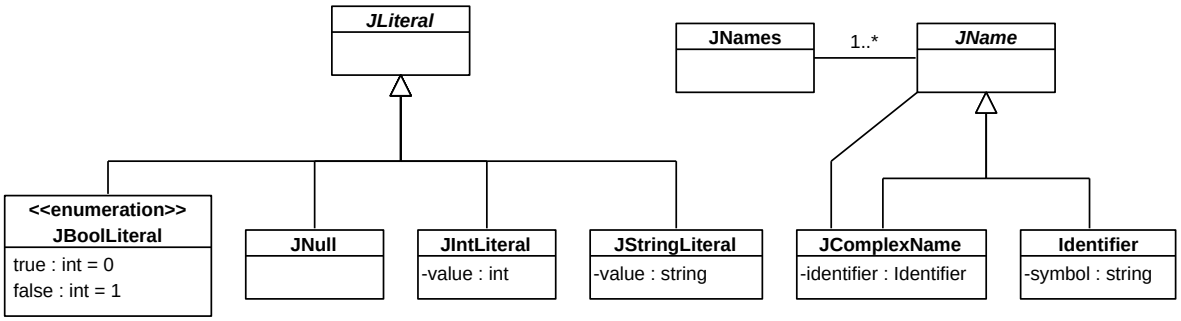


Figure F.4: Literals of MiniJava.

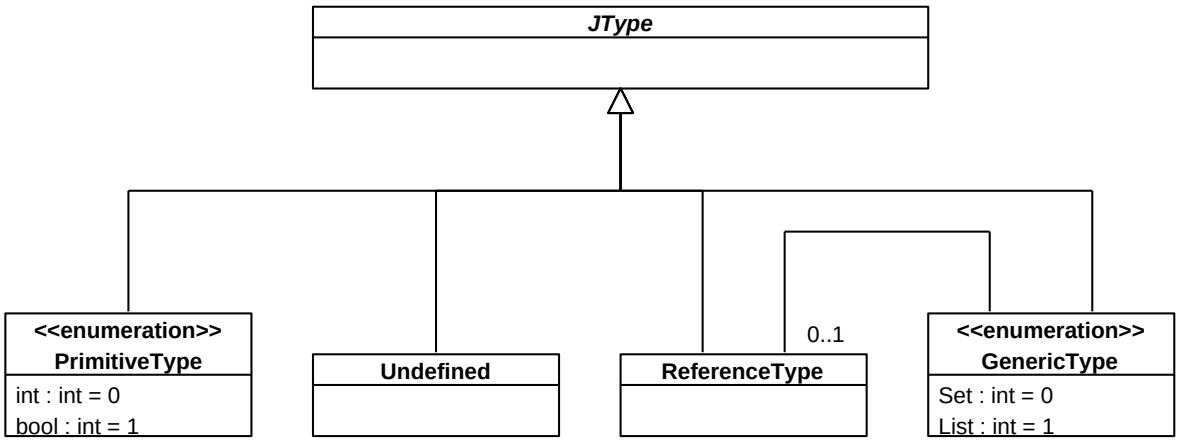


Figure F.5: Type system of MiniJava.

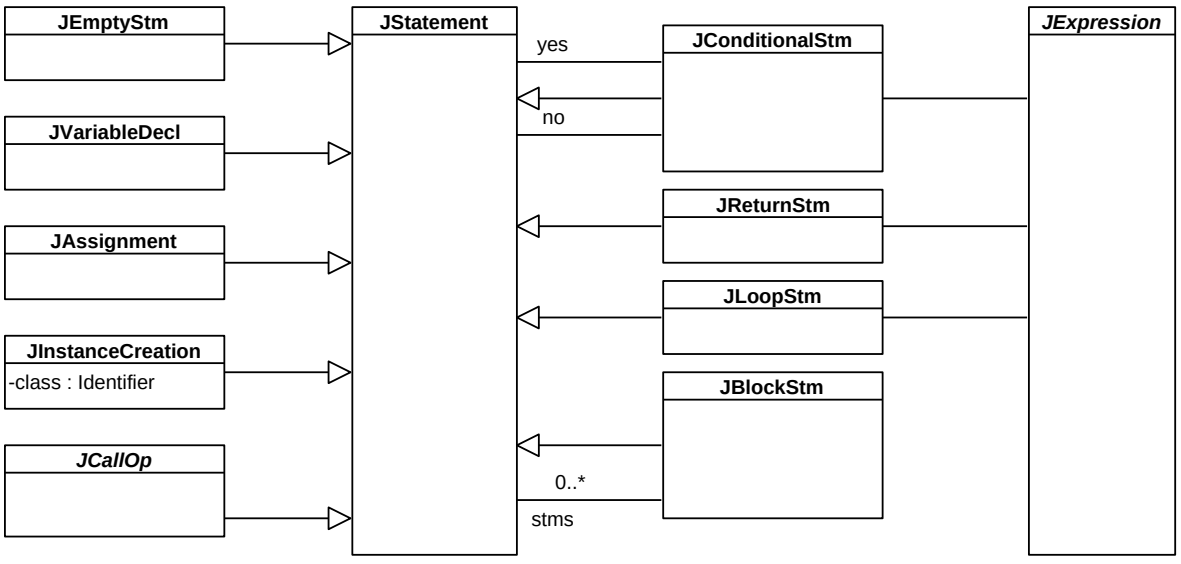


Figure F.6: Statements of MiniJava

Appendix G

Main Class of the xUML-to-ALx Translator

This appendix provides the core code of a standalone xUML-to-MiniJava translator, which is intended to illustrate how the required metamodels and models are loaded into run-time system, and how ATL engine is configured to perform the desired model transformations. Please refer to Eclipse ATL and EMF projects for the necessary background [2, 4].

With the required metamodels (xUML metamodels, ALx metamodels and MiniJava metamodels) and ATL files ready, the implementation of the translator is straightforward. The whole system is constituted by two major components: the ATL engine, which is already an off-the-shelf component provided as a part of the ATL runtime libraries, and the code generator, which is automatically produced from the JET templates composed by us. The whole process of a particular transformation can be decomposed into the following steps:

1. Instantiate the ATL VM.

2. Load metamodels and the input UML model. The order of the two is not significant.

3. Launch the first translation by the following settings.

   (a) Bind the loaded UML metamodel and ALx metamodel to the corresponding variables declard in UML2ALx ATL file for metamodel references.

(b) Bind the loaded input UML model to the corresponding variable declared in the UML2ALx ATL file for model reference.

(c) Specify the compiled UML2ALx ATL file as the required transformation files.

4. If required, serialize the intermediary in-memory ALx model resulted from the first translation.

5. Launch the second translation by the following settings.

   (a) Bind the loaded ALx metamodel and Java metamodel to the corresponding variables declard in ALx2Java ATL file for metamodel references.

   (b) Bind the loaded input UML model to the corresponding variable declared in the ALx2Java ATL file for model reference.

   (c) Specify the compiled ALx2Java ATL file as the required transformation files.

6. If required, serialize the intermediary in-memory Java model resulted from the second translation.

7. Invoke the code generator to generate Java code from the resultant in-memory Java model.

The whole execution scenario is completed when the Java code is generated (the code for generating textual Java code is ommitted here). Often, the generated code is then fed to an instantiated Java VM to be executed for the purpose of model simulation.

```
/*
 * Created on 3 March 2008. - last revision: 8.5.2008.
 * (C) 2008. Mikai Yang, Heriot-watt University.
 *
 * This software is a prototype translator
 * which can translate xUML model
 * into miniJava models.
 */
package atl;

import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.reflect.*;
import javax.jmi.reflect.*;
```

```
import org.xml.sax.Locator;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;

import org.eclipse.core.runtime.*;
import org.atl.engine.repositories.mdr4atl.*;
import org.atl.engine.vm.*;
import org.atl.eclipse.engine.*;
import org.atl.engine.extractors.*;
import org.atl.engine.extractors.ebnf.*;
import org.atl.engine.injectors.ebnf.*;
import org.atl.engine.extractors.xml.*;
import org.atl.engine.injectors.xml.*;
import org.atl.engine.repositories.emf4atl.*;
import org.atl.engine.vm.nativelib.*;

public class Xuml2JavaTranslator {


  // Relative location of transformations
  String atlFilesLoc = "..\\..\\transformations\\";
  // Relative location of metamodels
  String mmsLoc = "..\\..\\metamodels\\";

  // Obtain URLs of transformation files.
  private URL OneMM2OtherMMurl = Xuml2JavaTranslator.
    class.getResource(atlFilesLoc + "AL2G.asm");
  private URL Lib4Thisurl = Xuml2JavaTranslator.
    class.getResource(atlFilesLoc + "AL2G_LIB.asm");

  // EMF model handler
  private AtlModelHandler emfamh = null;

  // For keeping metamodel names
  Map MDRMetaModels = new HashMap();

  // Meta-models in EMF.
  private ASMModel oneMM = null;
  private ASMModel otherMM = null;
  private ASMModel xmlEMFmm = null;

  // Singleton ATLTransformation.
  private static Xuml2JavaTranslator translator = null;

  // General meta-models
  private ASMModel pbmm = null;
  // Markers for Problem metamodel
  private MarkerMaker markerMaker;
```

```
// Constructor
public Xuml2JavaTranslator() {
//Initialize EMF based metamodels
    initEMF();
}

/**
 * Initialize EMF model handler and
 * Ecore based metamodels
   */
private void initEMF() {

  if(emfamh == null) { // if EMF is not initialized

    // Initialize EMF model handler
    emfamh = AtlModelHandler.
      getDefault(AtlModelHandler.AMH_EMF);

    // URL's to Ecore (XMI) metamodels
    URL onemmurl = Xuml2JavaTranslator.class.
      getResource(mmsLoc +
      "OneMM\\AL_Simplified.ecore");
    URL xmlmmurl = Xuml2JavaTranslator.class.
      getResource(mmsLoc +
      "xml\\xml.ecore");
    URL othermmurl = Xuml2JavaTranslator.class.
      getResource(mmsLoc +
      "OtherMM\\AL_Simplified_Gen.ecore");

    try {
      // Load metamodels from persistence.
      oneMM = emfamh.loadModel("AL",
        emfamh.getMof(), onemmurl.openStream());
      xmlEMFmm = emfamh.loadModel("XML",
        emfamh.getMof(), xmlmmurl.openStream());
      otherMM = emfamh.loadModel("ALG",
        emfamh.getMof(), othermmurl.openStream());
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
  pbmm = emfamh.getBuiltInMetaModel("Problem");
  markerMaker = new MarkerMaker();
} // -- end of initEMF

 /**
 * Return a singleton translator.
 * @return default repository for input (EMF or MDR)
 */
  public static Xuml2JavaTranslator getTranslator() {
  if(translator  == null)
```

```
      translator = new Xuml2JavaTranslator();

    return translator;
  }

    /**
     * A helper method.
     * Validate the well-formedness of input XML file
     */
    private boolean checkWellFormedness(String file,
      boolean isFile) {
    SAXParser saxParser = null;
    DefaultHandler dh = null;
    InputStream in = null;

    // init parser
    try {
      SAXParserFactory spfactory =
        SAXParserFactory.newInstance();
      saxParser = spfactory.newSAXParser();
      dh = new DefaultHandler();
    }
    catch(Exception e) {
      System.out.println("Initialize SAX parser fails.");
      e.printStackTrace();
      return false;
    }

    // parse the XML document using SAX parser
    try {
      if (isFile == true) {
        in = new FileInputStream(file);
      }
      else {
        byte currentXMLBytes[] = file.getBytes();
        in = new ByteArrayInputStream(currentXMLBytes);
      }
      saxParser.parse(in, dh);
    }
    catch(SAXParseException spe) {
      System.out.println("File is not well-formed.");
      return false;
    }
    catch(SAXException se) {
      System.out.println("Error in parsing input XML file: "
        + file);
      se.printStackTrace();
      return false;
    }
    catch(FileNotFoundException f) {
      System.out.println("Error: File is not found");
      return false;
```

```
    }
    catch(IOException ioe) {
      System.out.println("Cannot read file.");
      return false;
    }
    return true;
  }

    /**
     * Inject input XML file to XML model
     * (instance of XML metamodel - MOF 1.4)
     */
  public ASMModel injectXMLModelFromFile(String file) {
    initEMF();

    ASMModel ret = emfamh.newModel("IN", xmlEMFmm);

    XMLInjector xmli = new XMLInjector();

    Map parameters = Collections.EMPTY_MAP;

    InputStream in = null;

    try {
      in = new FileInputStream(file);

      xmli.inject(ret, in, parameters);

    } catch(FileNotFoundException f) {
      System.out.println("Error: File is not found");
    } catch(IOException io) {
      System.out.println("Error: in injection of XML file");
    }

    return ret;
  } // -- end of injectXMLModelFromFile


    /**
     * Extract input XML model  to XML file
     */
  public void extractXMLModelToFile(ASMModel model,
    String file){
    initEMF();

    OutputStream out = null;

    Map parameters = Collections.EMPTY_MAP;

    XMLExtractor xmle = new XMLExtractor();

    try {
```

```
      out = new FileOutputStream(file);
      xmle.extract(model, out, parameters);
      out.flush(); out.close(); // close stream
    } catch(FileNotFoundException f) {
      System.out.println("Error: File is not found");
    } catch(IOException io) {
      System.out.println
        ("Error: In extracting XML model to XML file");
    }
  }

  /**
     * Extract input MM model to File
   */
    public void saveMMModelToFile(ASMModel MMModel,
      String file) {
    initEMF();

    OutputStream out = null;

    Extractor ext = new EBNFExtractor();
    Map params = new HashMap();
    params.put("format", MDRMetaModels.get("MM-TCS"));
    params.put("indentString", "\t");

    try {
      out = new FileOutputStream(file);
      ext.extract(MMModel, out, params);

    } catch(Exception e) {
      e.printStackTrace();
    }
  } // -- end of saveMMModelToFile


   /**
     * Launch ATL transformation
     */
  public ASMModel run(AtlModelHandler modelHandler,
     URL transformation, ASMModel inputModel,
      ASMModel inputMetamodel, ASMModel outputMetamodel,
      Map inParams, Map inLibs) {

    initEMF(); // Initialize MDR model handler

    ASMModel ret = null; // return model

    // Set launch configuration
    Map models = new HashMap();

    models.put(inputMetamodel.getName(), inputMetamodel);
    models.put(outputMetamodel.getName(), outputMetamodel);
```

```
    models.put("ac", inputModel);
    ret = modelHandler.newModel("acg", outputMetamodel);
    models.put("acg", ret);

    Map params = inParams; // Parameters
    Map libs = inLibs;     // Libraries

    // Launch ATL transformation
    AtlLauncher.getDefault().
      launch(transformation, libs, models, params);

    return ret;
  } /


    /**
     * run a traformation
     */
  public ASMModel getOtherMMFromOneMM(ASMModel oneModel) {

    //A library is used.
    Map libs = new HashMap();
      libs.put("Lib4This", Lib4Thisurl);

    // Run transformation and return output model
    return run(emfamh, OneMM2OtherMMurl,
      oneModel, oneMM, otherMM, Collections.EMPTY_MAP, libs);
  }

   /**
   * where the transformation takes palace.
   */
  public String transformOnetoOther(String InputOneFile,
    String OutputOtherFile) {
    // Check is input String well-formed (XML)
    if (!checkWellFormedness(InputOneFile, true))
      return new String("Document is not well-formed.");

    ASMModel xmlModel = injectXMLModelFromFile(InputOneFile);
    ASMModel otherModel = getOtherMMFromOneMM(xmlModel);
    extractXMLModelToFile(otherModel, OutputOtherFile);

    return new String("Translation completed.");
  }

  public static void main(String [] arguments) {

    // Create new instance of this class
    Xuml2JavaTranslator transformation =
      Xuml2JavaTranslator.getTranslator();

    String inputModel = "models\\one\\AClass.xmi";
```

```
        String outputModel = "d:\\Other.xml";

        // Transform input file to output file
        String message = transformation.
          transformOnetoOther(inputModel, outputModel);
        System.out.println(message);
      }

    }
```

Appendix H

ALx-to-Java Mapping Rules

This appendix provides the implementation-neutral ALx-to-MiniJava mapping rules that are defined in AS style, as well as the primary Java classes in the library part. The abstract syntactic definition of ALx has already been provided in Appendix A and reused in this description. For simplicity, in the translation rules, we do not explicitly define the translation of *Identifier*, considering that ALx identifiers are directly mapped to Java identifiers. The reader is recommended to refer to the library part to understand the translation rules because some constructs in ALx are mapped to facilities provided in the library classes.

H.1 MiniJava Abstract Syntax

Identifiers and Names

(1) Identifier = ⟦ letter(letter | digit)* ⟧ .

(2) JName = Identifier | ⟦ JName "." Identifier ⟧.

(3) JNames = ⟨ JName ⟨ "," JName ⟩* ⟩.

Types

needs: Identifiers and Names

(1) JType = JPrimitive-Type | JReference-Type.

(2) JPrimitive-Type = "boolean" | "int".

(3) JReference-Type = JName | JName "<" JName ">".

Literals

(1) JLiteral = JInteger-Literal | JBoolean-Literal | JString-Literal | "null".

(2) JInteger-Literal = □ .

(3) JBoolean-Literal = "true" | "false".

(4) JString-Literal = □ .

Expressions

needs: Identifier and Names, Types, Literals.

(1) JExpression = JLiteral | "this" |
JVariable-Access ⟦ JPrefix-Operator JExpression ⟧ |
⟦ JExpression JInfix-Operator JExpression ⟧ |
⟦ JExpression "instanceof" JName ⟧ |
⟦ "(" JType ")" JExpression ⟧ |
⟦ JExpression ("||" | "& &") JExpression ⟧ |
⟦ "(" JExpression ")" ⟧ |
JAssignment |
JCall-Operation |
JInstance-Creation.

(2) JCall-Operation = ⟦ JName "(" JArguments ")" ⟧
⟦ JExpression "." Identifier "(" JArguments ")" ⟧ |
⟦ "super" "." Identifier "(" JArguments ")" ⟧.

(3) JInstance-Creation = ⟦ "new" JName "(" JArguments ")" ⟧.

(4) JArguments = ⟨ JExpression ⟨ "," JExpression ⟩* ⟩?

(5) JPrefix-Operator = "-" | "!" .

(6) JInfix-Operator = "==" | "!=" | "<" | ">" | "<=" |
">=" | "+" | "−" | "*" | "/" | "%"

Variable Accesses

(1) JVariable-Access = JName | JField-Access.

(2) JField-Access = ⟦ JExpression "." Identifier ⟧ | ⟦ "super" "." Identifier ⟧

Assignment

(1) JAssignment = ⟦ JVariable-Access "=" JExpression ⟧.

Statements

needs: Declarations, Expressions.

Block

(1) JBlock-Statements = ⟦ JStatement* ⟧

(2) JStatement = ⟦ ";" ⟧ |
⟦ JVariable-Declaration ";" ⟧ |
⟦ "{" JBlock-Statements "}" ⟧ |
⟦ JAssignment ";" ⟧ |
⟦ JCall-Operation ";" ⟧ |
⟦ JInstance-Creation ";" ⟧ |
⟦ "return" JExpression? ";" ⟧ |
⟦ "if" "(" JExpression ")" JStatement "else" JStatement ⟧ |
⟦ "while" "(" JExpression ")" JStatement ⟧.

Declarations

needs: Identifiers and Names, Types, Statements, Expressions.

(1) JModifier = "public" | "private" | "protected" | "static" | "abstract" | "final".

Class Declaration

(1) JClass-Declaration = ⟦ JModifier* "class" Identifier JSuperclass-Clause? JInterface-Clause? JClass-Body ⟧.

(2) JSuperclass-Clause = ⟦ "extends" JName ⟧.

(3) JInterface-Clause = ⟦ "implements" JNames ⟧.

(4) JClass-Body = ⟦ "{" JClass-Body-Declaration* "}" ⟧.

(5) JClass-Body-Declaration = JField-Declaration | JConstructor-Declaration | JMethod-Declaration | JClass-Declaration.

Constructor Declarations

(1) JConstructor-Declaration = ⟦ JModifier* Identifier "(" JFormal-Parameters ")" JConstructor-Body ⟧.

(2) JConstructor-Body = ⟦ "{" JConstructor-Call? JBlock-Statements "}" ⟧.

(3) JConstructor-Call = ⟦ ("this" | "super") "(" JArguments? ")" ";" ⟧.

Method and Field Declarations

(1) JMethod-Declaration = ⟦ JModifier* ("void" | JType) Identifier "(" JFormal-Parameters ")" "{" JBlock-Statements "}" ⟧.

(2) JField-Declaration = ⟦ JModifier* JType Identifier ";" ⟧ |
⟦ JModifier* JType Identifier "=" JExpression ";" ⟧

Interface Declarations

(1) JInterface-Declaration = ⟦ JModifier* "interface" Identifier
JSuper-Interfaces-Clause
JInterface-Body ⟧

(2) JSuper-Interfaces-Clause = ⟦ "extends" JNames ⟧.

(3) JInterface-Body = ⟦ "{" JInterface-Member-Declaration* "}" ⟧.

(4) JInterface-Member-Declaration = JInterface-Field-Declaration | JInterface-Method-Declaration.

(5) JInterface-Field-Declaration = ⟦ JModifier* JType Identifier ";" ⟧.

(6) JInterface-Method-Declaration = ⟦ JModifier* ("void" | JType) Identifier "(" JFormal-Parameters ")" ";" ⟧.

Misc

(1) JFormal-Parameters = ⟨ JFormal-Parameter ⟨ "," JFormal-Parameter ⟩* ⟩?

(2) JFormal-Parameter = ⟦ JType Identifier ⟧.

(3) JVariable-Declaration = ⟦ JType Identifier ⟧ | ⟦ JType Identifier "=" JExpression ⟧

Compilation Units

(1) JCompilation-Unit = ⟦ JPackage-Declaration? JImport-Declaration* JType-Declaration* ⟧.

(2) JPackage-Declaration = ⟦ "package" JName ";" ⟧

(3) JImport-Declaration = ⟦ "import" JName ⟨ "." "*" ⟩? ⟧.

(4) JType-Declaration = JClass-Declaration | JInterface-Declaration.

Programs

(1) JProgram = JCompilation-Unit+ .

H.2 ALx-to-MiniJava Mapping Functions

needs: ALx Abstract Syntax, MiniJava Abstract Syntax.

H.2.1 Expressions

- TE _ :: Read-Attribute → JCall-Operation.

(1) TE ⟦I_1: Identifier "." I_2: Identifier⟧ = ⟦ I_1 "." getMtd(I_2) "(" ")"⟧.

The function 'getMtd' is intended to translate the identifier of an attribute to the corresponding 'get' method. For instance, an attribute named 'address' is accessed using the 'getAddress' method. The defintion of 'getMtd' is left open here.

- getMtd _ :: Identifier → Identifier

(2) getMtd I = □ .

- TE _ :: Call-Operation → JCall-Operation.

(3) TE ⟦E: Expression "." I: Identifier "(" A: Arguments ")"⟧ =
⟦TE(Expression) "." I "(" RTE(A) ")"⟧

- RTE _ :: Arguments → JArguments

(4) RTE ⟨ ⟩ = ⟨ ⟩.

(5) RTE E: Expression = TE(E).

(6) RTE ⟨ E: Expression ";" A: Arguments ⟩ = ⟨ TE(E) ";" RTE(A) ⟩.

- TE _ :: Expression → JExpression.

(7) TE L: Literal = TE(L).

(8) TE I: Identifier = I.

(9) TE "self" = "this".

(10) TE "selected" = "selected".

(11) TE ⟦O: Prefix-Operator E: Expression⟧ = ⟦TE(O) TE(E)⟧.

(12) TE ⟦E_1: Expression O: Infix-Operator E_2: Expression⟧ = ⟦TE(E_1) TE(O) TE(E_2)⟧.

(13) TE ⟦E_1: Expression "||" E_2: Expression⟧ = ⟦TE(E_1) "||" TE(E_2)⟧.

(14) TE ⟦E_1: Expression "&&" E_2: Expression⟧ = ⟦TE(E_1) "&&" TE(E_2)⟧.

(15) TE A: Read-Attribute = TE(A).

(16) TE C: Call-Operation = TE(C).

- TE Prefix-Operator → JPrefix-Operator.

(17) TE "-" = "-".

(18) TE "!" = "!".

(19) TE "empty" = □ .

- TE Infix-Operator → Infix-Operator.

(20) TE "==" = "==".

(21) TE "!=" = "!=".

(22) TE "<" = "<".

(23) TE “>” = “>”.

(24) TE “<=” = “<=”.

(25) TE “>=” = “>=”.

(26) TE “+” = “+”.

(27) TE “-” = “-”.

(28) TE “*” = “*”.

(29) TE “/” = “/”.

(30) TE “%” = “%”.

- TE Literal → JLiteral.

(31) TE B: Boolean-Literal = TE(B).

(32) TE I: Integer-Literal = TE(I).

(33) TE “null” = “null”.

- TE Boolean-Literal → Boolean-Literal.

(34) TE “true” = “true”.

(35) TE “false” = “false”.

- TE Integer-Literal → Integer-Literal.

(36) TE I: Integer-Literal = I.

- TE Identifier → Identifier.

(37) TE I: Identifier = I.

H.2.2 Statements

needs: Declarations, Expressions.

Statements

- TS Block-Statements → JBlock-Statements.

(1) TS ⟦ ⟨ ⟩ ⟧ = ⟨ ⟩.

(2) TS ⟦ S: Statement ⟧ = TS(S).

(3) TS ⟦ S: Statement B: Block-Statements ⟧ = TS(S) TS(B).

- TS _ :: Statement → JStatements.

(4) TS ⟦“;”⟧ = ⟦“;”⟧.

(5) TS ⟦“{” B: Block-Statements “}”⟧ = ⟦ “{” TS(B) “}” ⟧.

(6) TS ⟦ V: Variable-Declaration “;” ⟧ = ⟦ “{” TV(V) “;” “}” ⟧.

(7) TS $[\![$ C: Call-Operation ";" $]\!]$ = $[\![$ TS(C) ";" $]\!]$.

(8) TS $[\![A$: Assignment ";" $]\!]$ = $[\![$ TS(A) ";" $]\!]$.

(9) TS $[\![$ W: Write-Attribute ";" $]\!]$ = $[\![$ TS(W) ";" $]\!]$.

(10) TS $[\![$ R: Read-Attribute ";" $]\!]$ = $[\![$ TS(R) ";" $]\!]$.

(11) TS $[\![$ E: Event-Generation ";" $]\!]$ = $[\![$ TS(E) ";" $]\!]$.

(12) TS $[\![$ O: Object-Creation ";" $]\!]$ = $[\![$ TS(O) ";" $]\!]$.

(13) TS $[\![$ O: Object-Deletion ";" $]\!]$ = $[\![$ TS(O) ";" $]\!]$.

(14) TS $[\![$ L: Link-Creation ";" $]\!]$ = $[\![$ TS(L) ";" $]\!]$.

(15) TS $[\![$ L: Link-Deletion ";" $]\!]$ = $[\![$ TS(L) ";" $]\!]$.

(16) TS $[\![$ O: Object-Selection ";" $]\!]$ = $[\![$ TS(O) $]\!]$.

(17) TS $[\![$ L: Link-Navigation ";" $]\!]$ = $[\![$ TS(L) ";" $]\!]$.

(18) TS $[\![$ O: Object-Reclassification ";" $]\!]$ = $\square$.

(19) TS $[\![$ "return" ";" $]\!]$ = $[\![$ "return" ";" $]\!]$.

(20) TS $[\![$ "return" E: Expression ";" $]\!]$ = $[\![$ "return" TS(E) ";" $]\!]$.

(21) TS $[\![$"if" "(" E: Expression ")" S_1: Statement "else" S_2: Statement$]\!]$ =
$[\![$"if" "(" TS(E) ")" "" TS(S_1) "" "else" "" TS(S_2) ""$]\!]$.

(22) TS $[\![$ "while" "(" E: Expression ")" S: Statement $]\!]$ =
$[\![$ "while" "(" TS(E) ")" "" TS(S) "" $]\!]$.

(23) TS $[\![$ S_1: Statement S_2: Statement$^+]\!]$ = $[\![$ TS(S_1) TS(S_2)$]\!]$

- TS _ :: $[\![$ Variable-Declaration* → JVariable-Declaration*.

Assignments

- TS _ :: Assignment → JAssingment.

(1) TS $[\![$ I_1: Identifier "=" E: Expression$]\!]$ = $[\![$ I_1 "=" TE(E)$]\!]$.

Object Manipulation

- TS _ :: Write-Attribute → JCall-Operation.

(1) TS $[\![$ I_1: Identifier "." I_2: Identifier = "." E: Expression $]\!]$ =
$[\![$ I_1 "." setMtd(I_2) "(" TE(E) ")" $]\!]$.

The function 'setMtd' is intended to translate the identifier of an attribute to the corresponding 'set' method, a method for setting value of this attribute. For instance, an attribute named 'address' is written using the 'setAddress' method. The defintion of 'setMtd' is left open here.

- setMtd _ :: Identifier → Identifier.

(2) setMtd I = $\square$.

- TS _ :: Object-Creation → JAssignment.

(3) TS ⟦ "create-object" I_1: Identifier "of" I_2: Identifier "(" A: Arguments ")" ⟧ =

⟦ I_1 "=" newInstance(I_2, A)⟧

- newInstance _ :: Identifer, Arguments → JCall-Operation.

(4) newInstance I : Identifier, A: Arguments = ⟦ I "." "newInstance" "(" RTE(A) ")"⟧.

- TS _ :: Object-Deletion → JCall-Operation.

(5) TS ⟦"delete-object" I: Identifier⟧ = ⟦I "." "desroy" "(" ")" ⟧.

- TS _ :: Object-Reclassification → JAssignment.

(6) *We do not implement Object-Reclassification in the current version of the ALx-to-Java translator.*

Link Manipulation

- TS _ :: Link-Creation → JCall-Operation.

(1) TS ⟦ "link" I_1: Identifier "→ " I_2 Identifier "(" I_3: Identifier ")" ⟧ = newLink (I_1, I_2, I_3).

- newLink (_, _, _) :: Identifier, Identifier, Identifier → JCall-Operation.

(2) newLink (I_1: Identifier, I_2: Identifier, I_3: Identifier) → ⟦ I_3 "." "newLink" "(" I_1 "," I_2 ")" ⟧.

- TS _ :: Link-Deletion → JCall-Operation.

(3) TS ⟦ "unlink" I_1: Identifier "→ " I_2: Identifier "(" I_3: Identifier ")" ⟧ = deleteLink(I_1, I_2, I_3).

- deleteLink (_, _, _) :: Identifier, Identifier, Identifier → JCall-Operation.

(4) deleteLink (I_1: Identifier, I_2: Identifier, I_3: Identifier) = I_3 "." "deleteLink" "(" I_1 "," I_2 ")".

Event Generation

- TS _ :: Event-Generation → JMethod-Invocation.

(1) TS ⟦ "send-event" I_1: Identifier "→ " I_2: Identifier ⟧ =
⟦ I_2 "." "stateTransmitted" "(" "EIDs" "." I_1 ")" ⟧.

- TS _ :: State-Transition ">>" Identifier.

(2) TS ⟦E: Expression ">>" I: Identifier ⟧ = □ .

Object Query

- TS _ :: Object-Selection → JStatement*.

(1) TS ⟦ "select-one" I: Identifier "of " C: Identifier ⟧ =
⟦ I "=" C "." "objectList.getFirst(); " ⟧.

(2) TS ⟦"select-one" I: Identifier "of" C: Identifer "(" E: Expression ")"⟧ =
⟦ "Iterator⟨ ALObject ⟩ iterator = " C "." "objectList.iterator(); "
"while(iterator.hasNext()){"
"ALObject selected = iterator.next();"
"if(" TE(E) ") {" I "= selected; break;}"
"}" ⟧.

(3) TS ⟦"select-many" I: Identifier "of" C: Identifer⟧ =
⟦ "Iterator ⟨ ALObject ⟩ iterator = " C "." "objectList.iterator(); "
"while(iterator.hasNext()){"
"ALObject selected = iterator.next();"
I "." "add(selected);"
"}" ⟧.

(4) TS ⟦"select-many" I: Identifier "of" C: Identifer "(" E: Expression ")"⟧ =
⟦ "Iterator⟨ ALObject ⟩ iterator = " C "." "objectList.iterator(); "
"while(iterator.hasNext()){"
"ALObject selected = iterator.next();"
"if(" TE(E) ") {" I "." "add(selected); }"
"}" ⟧.

- TS _ :: Link-Navigation → JAssignment.

(5) TS ⟦ I_1: Identifier "=" I_2: Identifier "→ " I_3: Identifier⟧ = ⟦ I_1 "=" linkedObject (I_2, I_3) ⟧.

In the actual Java implementation, there is appropriate type cast prefixed to 'linkedObject'.

- linkedObject (_, _) :: Identifier, Identifier → JCall-Operation.

(6) linkedObject (I_1: Identifier, I_2: Identifier) = ⟦ I_1 "." "getLinkedObject" "(" "RIDs" "." I_2 ")" ⟧.

- TS _ :: Link-Navigation → JAssignment.

(7) TS ⟦ I_1: Identifier "=" I_2: Identifier "→ *" I_3: Identifier⟧ = ⟦ I_1 "=" linkedObjects (I_2, I_3) ⟧.

- linkedObjects (_, _) :: Identifier, Identifier → JCall-Operation.

(8) linkedObjects (I_1: Identifier, I_2: Identifier) = ⟦I_1 "." "getLinkedObjects" "(" "RIDs" "." I_2 ")"⟧.

The translation of the conditional Link-Navation is similar to the conditional Object-Selection and thus is ignored here.

H.2.3 Declarations

needs: Statements, Expressions.

Class Declaration

Each ALx class is translated into a Java class and a Java interface. The translations of classes with constructors and the constructors themselves are intuitive and ignored to save space.

- TD _ :: Class-Declaration → JCompilation-Unit.

(1) TD ⟦ "class" I: Identifier "{"
F: Field-Declaration*
M: Method-Declaration* "}"
S: State-Machine-Declaration⟧ ⟧ =
⟦ ImportStms GeneratedClass(I, F, M) GeneratedInterface(I_1, F, M) GeneratedStateMachine⟧.

(2) TD ⟦ "class" I_1: Identifier "extends" I_2 "{"
F: Field-Declaration*
M: Method-Declaration* "}" ⟧ =
⟦ ImportStms GeneratedClass(I_1, I_2, F, M) GeneratedInterface(I_1, F, M)⟧.

The generated Java classes need three imported system classes. This is defined by the translation function 'ImportStms'.

- ImportStms : ⟨ JImport-Declaration, JImport-Declaration, JImport-Declaration ⟩.

(3) ImportStms = ⟨ "import library.*;", "import java.util.Iterator;", "import java.util.LinkedList;" ⟩.

The translation function 'GeneratedClass' has two versions: one has three parameters, and the other has four. The former is intended for the classes without super classes, and the latter for the classes with super classes.

- GeneratedClass (_, _, _) :: Identifier, Field-Declaration*, Method-Declaration* → JClass-Declaration.

(4) GeneratedClass(I: Identifier, F: Field-Declaration, M: Method-Declaration*) =
⟦"public" "class" I "extends" "ALObject" "{"
⟨ Field-ObjectList, RTF(F), RTF2GetMtd(F), RTF2SetMtd(F) ⟩
⟨ Method-NewInstance(I), Method-RecordObject4NoSuperclass TD(M) ⟩
"}"⟧.

- GeneratedClass(_, _, _, _):: Identifier, Field-Declaration*, Method-Declaration* → JClass-Declaration.

(5) GeneratedClass(I_1: Identifier, I_2: Identifier F: Field-Declaration, M: Method-Declaration*) =
⟦"public" "class" I_1 "extends" "ALObject" "{"
⟨ Field-ObjectList, DelegateField(I_2), RTF(F), RTF2GetMtd(F), RTF2SetMtd(F) ⟩
⟨ Method-NewInstance(I_1), Method-RecordObject4HasSuperclass TD(M) ⟩
"}".

For the ALx classes that have superclasses, a delegate object is created to represent the features of the superclass.

- DelegateField _ :: Identifier → JField-Declaration.

(6) DelegateField I: Identifier =
⟦ "private" I "parent_delegator" "=" "new" I "(" ")" ";"⟧.

- Method-NewInstance _ :: Identifier → JMethod-Declaration.

(7) Method-NewInstance I: Identifier =
⟦ "public" "static" I "newinstance" "(" ")" "{"
I "temp" "=" "new" I "(" ")" ";"
"recordObject" "(" "temp" ")" ";" "return" "temp" ";"
}⟧.

Object-recording methods vary depending on whether the class has super class or not. If it has, the newly-created object would be added to the superclasses as well.

- Method-RecordObject4NoSuperclass : JMethod-Declaration.

(8) Method-RecordObject4NoSuperclass = ⟦
"public" "static" "void" "recordObject" "(" "ALObject" "alo" ")" "{"
"objectList" "." "addObject" "(" "alo" ")" ";"
"}" ⟧

- Method-RecordObject4HasSuperclass _ :: Identifier → JMethod-Declaration.

(9) Method-RecordObject4HasSuperclass I = ⟦
"public" "static" "void" "recordObject" "(" "ALObject" "alo" ")" "{"
"objectList" "." "addObject" "(" "alo" ")" ";"
I "." "recordObject" "(" "alo" ")" ";" "}" ⟧

- Field-ObjectList: JField-Declaration.

(10) Field-ObjectLis = ⟦ "static" "ObjectList" "objectList" "=" "new" "ObjectList" "(" ")" ";" ⟧ .

- GeneratedInterface (_, _, _) :: Identifier, Field-Declaration, Method-Declaration → JInterface-Declaration.

(11) GeneratedInterface ... = □ . *Translating an ALx class to a Java interface is straightforwad and is ignored here.*

- RTF _ :: Field-Declaration* → JField-Declaration*

(12) RTF ⟨ ⟩ = ⟨ ⟩.

(13) RTF ⟦T: Type I: Identifier ";"⟧ = ⟦"public" TT(T) I ";"⟧

(14) RTF ⟨ F: Field-Declaration F_1: Field-Declaration$^+$ ⟩ = ⟨ RTD (F), RTD(F_1) ⟩.

Each attribute causes the generation of a method for setting its value.

- RTF2SetMtd _ :: Field-Declaration* → JMethod-Declaration*.

(15) RTF2SetMtd ⟨ ⟩ = ⟨ ⟩.

(16) RTF2SetMtd ⟦T: Type I: Identifier ";"⟧ = ⟦"public void" setMtd(I) "(" TT(T) value ")" ""⟧

(17) RTF2SetMtd ⟨ F: Field-Declaration F_1: Field-Declaration$^+$ ⟩ = ⟨ RTD2SetMtd(F), RTD2SetMtd(F_1) ⟩.

Each attribute causes the generation of a method for reading its value.

- RTF2GetMtd _ :: Field-Declaration* → JMethod-Declaration*.

(18) RTF2GetMtd ⟨ ⟩ = ⟨ ⟩.

(19) RTF2GetMtd ⟦T: Type I: Identifier ";"⟧ = ⟦"public void" setMtd(I) "(" TT(T) value ")" ""⟧

(20) RTF2GetMtd ⟨ F: Field-Declaration F_1: Field-Declaration$^+$ ⟩ = ⟨ RTF2GetMtd(F), RTF2GetMtd(F_1) ⟩.

- TD _ :: Method-Declaration* → JMethod-Declaration*.

(21) TD ⟨ ⟩ = ⟨ ⟩.

(22) TD ⟦ "void" I: Identifier "(" F: Formal-Parameters ")" "" B: Block-Statements""⟧ =
⟦ "void" I "(" RTF(F) ")" "{" TS(B) "}"⟧.

(23) TD ⟦ T: Type I: Identifier "(" F: Formal-Parameters ")" "" B: Block-Statements""⟧ =
⟦ TT(T) I "(" RTF(F) ")" "{" TS(B) "}"⟧.

(24) TD ⟨ M_1: Method-Declaration, M_2: Method-Declaration$^+$ ⟩ =
⟨ TD(M_1), TD(M_2) ⟩.

Relation Declaration

- TD _ :: Relation-Declaration → JClass-Declaration

(1) TD ⟦ "relation" I_1: Identifier I_2: Identifier "→ " I_3: Identifier ";" ⟧ =
⟦ "public" "class" I_1 "extends" "ALRelation" "{"
Field-Rid (I_1)
Field-OneEnd (I_2)
Field-AnotherEnd (I_3)
Method-GetOneEnd
Method-GetAnotherEnd
Method-GetRid
Method-newLink
"}" ⟧.

- Field-Rid _:: Identifier → JField-Declaration.

(2) Field-Rid I: Identifier =
⟦ "public" "static" "int" "rid" "=" "RIDs" "." I ";" ⟧.

- Field-OneEnd _:: Identifier → JField-Declaration.

(3) Field-Rid I: Identifier =
⟦ "public" "static" "int" "oneEnd" "=" "RIDs" "." I ";" ⟧.

- Field-AnotherEnd _:: Identifier → JField-Declaration.

(4) Field-Rid I: Identifier=
⟦ "public" "static" "int" "anotherEnd" "=" "RIDs" "." I ";" ⟧.

- Method-GetOneEnd : JMethod-Declaration.

(5) Method-GetOneEnd =
⟦ "public" "static" "int" "getOneEnd" "(" ")" "{"
"return" "oneEnd" ";"
"}" ⟧.

- Method-GetAnotherEnd : JMethod-Declaration.

(6) Method-GetOneEnd = ⟦ "public" "static" "int" "getAnotehrEnd" "(" ")" "{"
"return" "anotherEnd" ";"
"}" ⟧.

- Method-GetRid : JMethod-Declaration.

(7) Method-GetRid =
⟦ "public" "static" "int" "getAnotehrEnd" "(" ")" "{"
"return" "rid" ";"
"}" ⟧.

- Method-newLink : JMethod-Declaration.

(8) Method-newLink =
⟦ "public" "static" "ALLink" "newLink" "(" "ALObject" "o1" "," "ALObject" "o2" ")" "{"
"ALLink" "temp" "=" "new" "ALLink" "(" "rid" "," "o1" "," "o2" ")" ";"
"o1" "." "addLink" "(" "temp" ")" ";"
"o2" "." "addLink" "(" "temp" ")" ";"
"return" "temp" ";"
"}" ⟧.

State Machine Declarations

- GeneratedStateMachine _ : → JClass-Declaration.

(1) GeneratedStateMachine =
⟦ "state-machine" "{"
S: State-Declaration$^+$
"initial-state: " I: Identifier
"transition-table" "{" T: Transition-Entries "}"
"}" ⟧ =
⟦ "class" "SM" "extends" "StateMachine" "{"
SM-Fields (S)
SM-Constructor (T)
SM-States (S)
"}" ⟧.

- SM-Fields _ :: State-Declaration $^+$ → JField-Declaration$^+$.

(2) SM-Fields ⟦ "state" I: Identifier "{"
"entry" "{" B_1: Block-Statements "}"
"exit" "{" B_2: Block-Statements "}"
"}" ⟧ =
⟦ "public" "state" I "=" "new" I "(" ")" ";"⟧.

(3) SM-Fields ⟨ F_1: SM-Fields F_2: SM-Fields$^+$ ⟩ = ⟨ SM-Fields (F_1), SM-Fields (F_2) ⟩.

- SM-States _ :: State-Declaration$^+$ → JClass-Declaration$^+$.

(4) SM-States ⟦ "state" I: Identifier "{"
"entry" "{" B_1: Block-Statements "}"
"exit" "{" B_2: Block-Statements "}"
"}" ⟧ =
⟦"public" "class" I "extends" "State" "{"
"public" "void" "entry" "{" TS(B_1) "}"
"public" "void" "exit" "{" TS(B_2) "}"
"}"⟧.

(5) SM-States ⟨ S_1: State-Declaration S_2: State-Declaration$^+$ ⟩ =
⟨ SM-States S_1, SM-States S_1 ⟩.

- SM-Construtor (_, _) :: Identifier, Transition-Entries → JConstructor-Declaration.

(6) SM-Construtor (I: Identifier, T: Transition-Entries) =
⟦ "public" "SM" "(" ")" "{"
"currentState" "=" "this" "." I ";"
AddEntryFields (T)
"}" ⟧.

- AddEntryFields _ :: Transition-Entry ⟨ ";" Transition-Entry ⟩* → JField-Declaration$^+$.

(7) AddEntryFields I_1: Identifier "," I_2: Identifier "," I_3: Identifier =

⟦ "this" "." "addEntry" "(" I_1 "," "EIDs" "." I_2 "," I_3 ")" ";" ⟧.

(8) AddEntryFields T_1: Transition-Entry ⟨ ";" T_2: Transition-Entry ⟩$^+$

⟨ AddEntryFields(T_1), AddEntryFields(T_2) ⟩.

Misc

- RTF _ :: Formal-Parameters → JFormal-Parameters.

(1) RTF $\langle\ \rangle = \langle\ \rangle$.

(2) RTF F: Formal-Parameter = TF(F).

(3) RTF $\langle\ F_1$: Formal-Parameter "," F_2: Formal-Parameter$^+\ \rangle = \langle$ TF(F_1) "," RTF(F_2)$\rangle$.

- TF _ :: Formal-Parameter → JFormal-Parameter.

(4) TF $[\![\ T$: Type I: Identifier $]\!] = [\![$ TT(T) $I]\!]$.

- TV _ :: Variable-Declaration → JVariable-Declaration.

(5) TV $[\![\ T$: Type I: Identifier $]\!] = [\![$ TT(T) I ";"$]\!]$.

(6) TV $[\![\ T$: Type I: Identifier "=" E: Expression $]\!] = [\![$ TT(T) I "=" TE(E) $]\!]$.

- TT _:: Type → JType.

  The type of ALx is designed to be a subset of Java types, so the type translation is intuitive and ignored here.

Appendix I

Sample xUML Models

This appendix presents the sample xUML models for various systems, which are used to test the two xUML-to-Java translators.

I.1 Taxi-Booking System

This is a fictitious simple taxi-booking and planning system intended for a not very contemporary company AATaxi. There are operators receiving phones calls from customers to arrange a taxi booking, or customers can directly surf to the company's website to make reservations.

The system holds some basic information of the purchased vehicles, employed drivers and customers. Furthermore, the company categorizes the customers into two kinds — *contract customers* and *casual customers*. The contract customers are pre-registered and are analogous to members of clubs enjoying some discount when they consume the services, while the casual customers normally take rides casually so they are not interested in the discount exclusive to contract customers, or they are not willing to spend some time in registration. We model this system in xUML as Figure I.1 shows.

The system allows operators to configure the initial state of the system, such as entering basic data and setting up policies including the discount for contract customers and the normal charge per mile. The system also provides an interface for operator to update and manage basic data: The operators can enter a vehicle, driver or customer into the system, or delete one of them from the system. However the core business of the system is to help in booking taxis for customers; The typical booking procedures

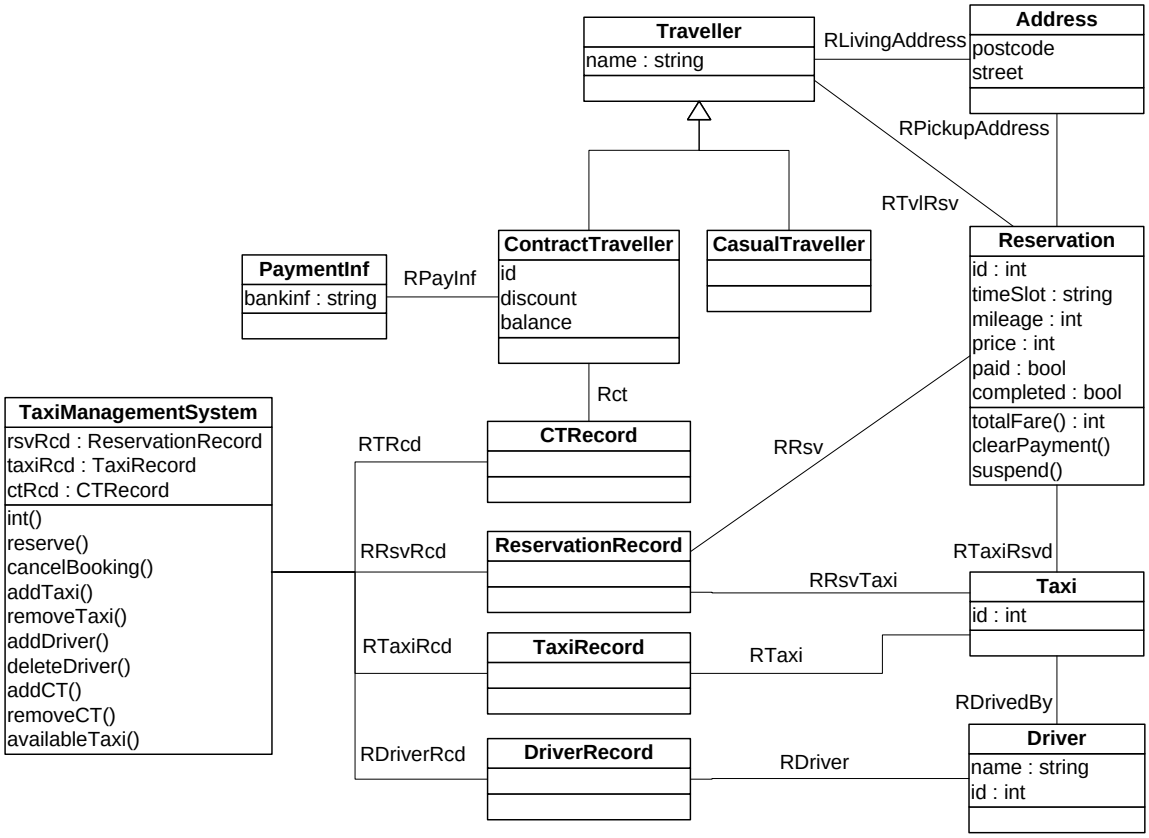


Figure I.1: Class diagram of the taxi-booking system.

are stated as follows:

1. The operator receives a phone call from the customer who would like to make a reservation.

2. The operator asks when and where the customer needs the service.

3. The operator views available taxis at the time that the customer requires.

4. The operator chooses one taxi and reserves this taxi for the customer.

This typical booking scenario will be mimicked in the main method of the ALx code of the system.

After a reservation is made, this reservation is in a 'Created' state and awaits being served. After the driver has delivered the customer to the right place at right time as the reservation requires, the payment is made. In respect to contract customers, they do not need to make the payment on the spot and the incurred fare would be added to the balance of their account, which may be cleared in period. For casual customers, they must make the payment on the spot. Anyway, the driver would inform the operator

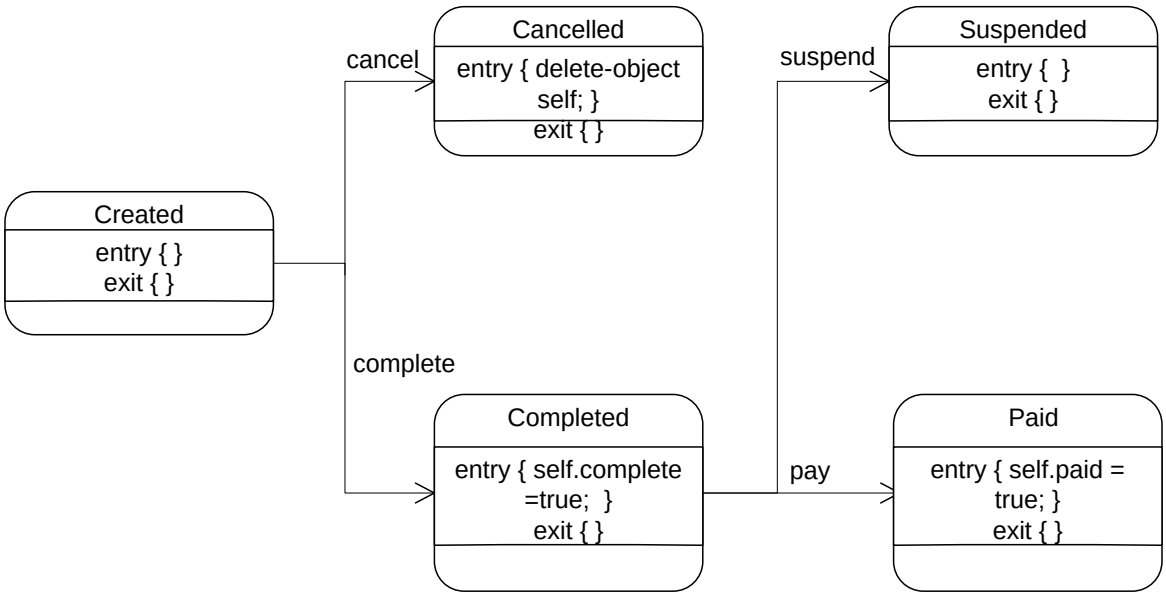


Figure I.2: State chart of 'Reservation' of the taxi-booking system.

of the payment situation, either paid or suspended, using radio communication. The state-chart of reservation is composed in Figure I.2;

```
class TaximanagementSystem {
    ReservationRecord rsvRcd;
    TaxiRecord taxiRcd;
    CTRecord ctRcd;
    DriverRecord driverRcd;
    int rsv_id;
    void init(){
        rsvRcd = self → RRsvRcd;
        taxiRcd = self →  RTaxiRcd;
        ctRsv = self → RTRcd;
        driverRcd = self → RDriver;
        rsv_id = 0;
    }
    void reserve(int taxiID, String slot, Traveller traveller){
        Taxi taxi = self → RTaxiRcd → Rtaxi (selected. Id == taxiID);
        create-object rsv of Reservation;
        rsv.id = rsv_id + 1;
        rsv.timeSlot = slot;
        link rsv → traveller (RTvlRsv);
        link rsv → taxi (RTaxiRsvd);
        link rsv → rsvRcd(RRsvRcd);
    }
    cancelBooking(int rsvId){
        Reservation rsv;
        rsv = rsvRcd → RRsv (selected.id = rsvId);
        rsv >> cancelled;
    }
    addTaxi(Taxi taxi){
        link taxi → taxiRcd (RTaxi);
    }
    remove(Taxi taxi){
        if ( (taxi → RTaxiRsvd) != null ){
              prints("error: cannot removed");
              return;
        }
        if ( (taxi → RDriver) != null ){
              prints("error: cannot removed");
              return;
        }
        delete-object taxi;
    }

    addDriver(Driver driver){
        link driver → driverRcd (RDriver) ;
    }
    removeDriver(Driver driver){
        delete-object driver;
    }
    addCT(ContractTraveller ct){
        link ct → ctRcd (RTRcd) ;
    }
    removeCT(ContractTraveller ct){
        delete-object ct;
    }
    availableTaxi(String time){
        select-many taxis of Taxi;
        for(int i=0; i< taxis.length; i++){
              Taxi taxi = taxis.get(i);
              Reservation rsvs = taxi → RTaxiRsvd(! selected.timeSlot.
                    equals(time));
              if (rsvs !=null)
                    prints("Taxi" + id +"available");
        }
    }
}
```

```
class CTRecord { }
class ReservationRecord { }
class TaxiRecord { }
class DriverRecord { }

class Reservation {
    boolean paid = false;
     int id;
    String timeSlot;
     int mileage;  int price;

    int totalFare() { return price * mileage; }
    clearPayment(int mileage){
        self >> Paid;
    }
    suspend () { self >> Suspended; }
}

class Taxi { int id;  }

class Driver{
    String name;
    int id;
}

class Traveller{
     String name;
 }

class ContractTraveller extends Traveller{
    int id;
    int discount;
     int balance;
}

class CasualTraveller extends Traveller { }
class Address{ String postcode;  String street; }
class PaymentInf{ String bankInf; }

// Relation declarations.
relation RTRcd TaxiManagementSystem →CTRecord;
relation RTRcd TaxiManagementSystem →
ReservationRecord;
relation RTaxiRcd TaxiManagementSystem → TaxiRecord;
relation RDriverRcd TaxiManagementSystem →
DriverRecord;

relation Rct CTRecord → ContractTraveller;
relation RPayInf ContractTraveller → PaymentInf;
relation RLivingAddress Traveller → Address;
relation RPickupAddress Reservation → Address;

relation RTvlRsv Reservation → Traveller;
relation RTaxiRsvd Taxi → Reservation;
relation RRsvTaxi ReservationRecord → Taxi;
relation RRsv Reservation→ Taxi;
relation RdrivedBy Taxi → Driver;
relation RDriver DriverRecord → Taxi;

// Event declarations
event cancel TaxiManagementSystem  → Reservation;
event complete Driver → Reservation;
event suspend Driver → Reservation;
event pay Driver → Reservation;
```

Figure I.3: ALx code of the taxi-booking system (Part 1).

```
state_machine_of Reservation{
    initial-state: Created
    state Created{
        entry {  }
        exit {  }
    }
    state Suspended  {
        entry { }
        exit { }
    }
    state Completed{
        entry { self.completed = true;   }
        exit {  }
    }
    state Paid {
        entry {self.paid = true;  }
        exit { }
    }
    state Cancelled{
        entry { delete-object self;  }
        exit { }
    }

        transition-table{
        Created,  complete,  Completed;
        Completed,  pay,  Paid;
        Completed, suspend, Suspended;
        Created,  cancel, Cancelled;
    }
}

// The main method of various use scenarios.
main(){
// configure data store;
    TaxiManagementSystem tsm;
    CTRecord ctRcd;
    ReservationRecord rsvRcd;
    TaxiRecord taxiRcd;
    DriverRecord driverRcd;

    create-object tsm of TaxiManagementSystem;
    create-object ctRcd of CTRecord;
    create-object rsvRcd of ReservationRecord;
    create-object taxiRcd of TaxiRecord;
    create-object driverRcd of DriverRecord;

    link tsm → ctRcd (RTRcd);
    link tsm → rsvRcd (RRsvRcd);
    link tsm → TaxiRcd (RTaxiRcd);
    link tsm → DriverRecord (RDriverRcd);

 // configure 8 example ContracTravelelrs.
    ContractTraveller ct0;  ContractTraveller ct1;  ...
    ContractTraveller ct5;

    create-object ct0 of ContractTraveller ;
    create-object ct1 of ContractTraveller;
    …
    ct0.id = 0; ct0.discount = 0.75; ct0.balacne = 0;
    ct1.id =1; ct1.discount = 0.80; ct1.balance = 100;
    …
    PaymentInf p0; PaymentInf p1; …; PaymentInf p7;
    create-object p0 of PaymentInf;
    create-object p1 of PaymentInf;
    …
    Address a0; Address a1; …; Address A7;
    create-object a0 of Address;
    create-object a1 of Address;
    …
    link ct0 → ctRcd (Rct);
    link ct0 → p0 (RPayInf);
    link ct0 → a0 (RLivingAddress);
    link ct1 → ctRcd (Rct);
    link ct1 → p1 (RPayInf);
    link ct1 → a1 (RLivingAddress);
    …
// configure 8 taxis
    Taxi t0; Taxi t1; …;  Taxi t7;
    create-object t0 of Taxi;
    create-object t1 of Taxi;
    …
    t0.id = 0; t1.id =2; …; t7.id = 7;
    link t0 → rsvRcd (RRsvTaxi);
    link t1 → rsvRcd (RRsvTaxi);
    …
    link t0 → taxiRcd (RTaxi);
    link t1 → taxiRcd (RTaxi);
    …

// configure 8 drivers
    Driver d0; Driver D1; …; Driver D7;
    create-object d0 of Driver;
    create-object d1 of Driver;
    ...
    d0.name = “Smith”; d0.id = 0; d1.name = “Jack”; d1.id = 1; …;
    link d0 → t0 (RDrivedBy);
    link d1 → t1 (RDrivedBy);
    ...
    link d0 → driverRcd (RDriver);
    link d1 → driverRcd(Rdriver);
    ...
    // Make reservation.
      // 1) create a reservation.
    Reservation rsv ;
    create-object rsv of Reservation;
    rsv.id = 0;
     // 2) view available taxi;
        tsm.availabeTaxi();
        //3) operator chooses a taxi for the reservation.
        // For example, t4 is available.
        link rsv → t4 (RTaxiRsvd);
        rsv.timeSlot =”30.15.24.09.2008”;
        rsv.price = 2;
        // If it is a contract traveller, the traveller says his ID no.
        // suppose id = 3;

        Traveller traveller = tms → RTRcd → Tct (selected.id=3);
        link rsv → traveller (RTaxiRsvd);
        // passenger is delivered, then he makes payment.
        rsv.clearPayment();
    }
```

Figure I.4: ALx code of the taxi-booking system (Part 2).

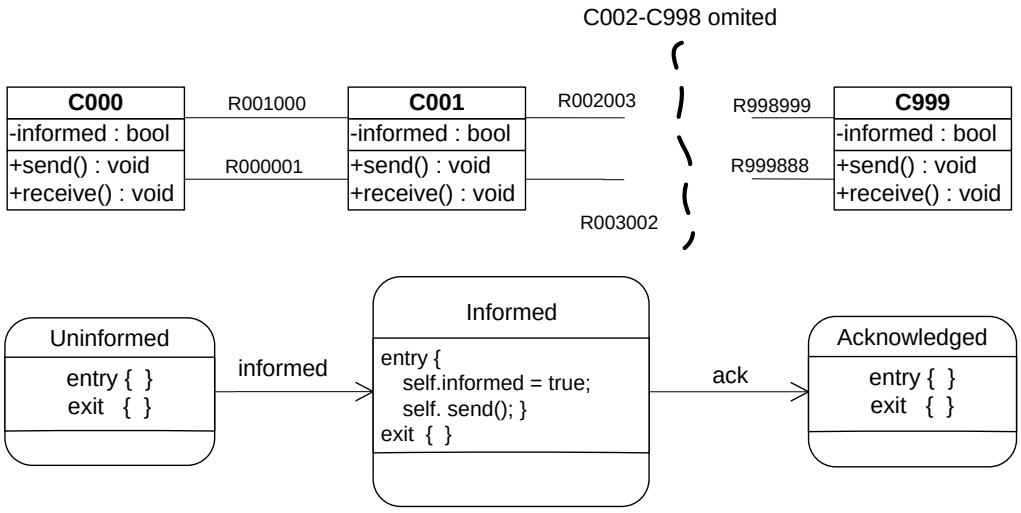


Figure I.5: xUML model of the toy message relay system.

I.2 Toy Message Relay System

The toy message relay system is a hypothetical system that models the relaying of a message between 1000 persons. A person relays a message to the next person, who subsequently acknowledges this message, updates its state to be 'informed' and then relays the message to the next person. This process will go on until all persons are informed of this message. The xUML model for this system is large-scale: the system has 1000 classes and each class has a state machine. The xUML model is illustrated in Figure I.5. The XMI file for this model is generated automatically. The corresponding ALx code of this system is illustrated in Figure I.6.

I.3 Traffic Light System

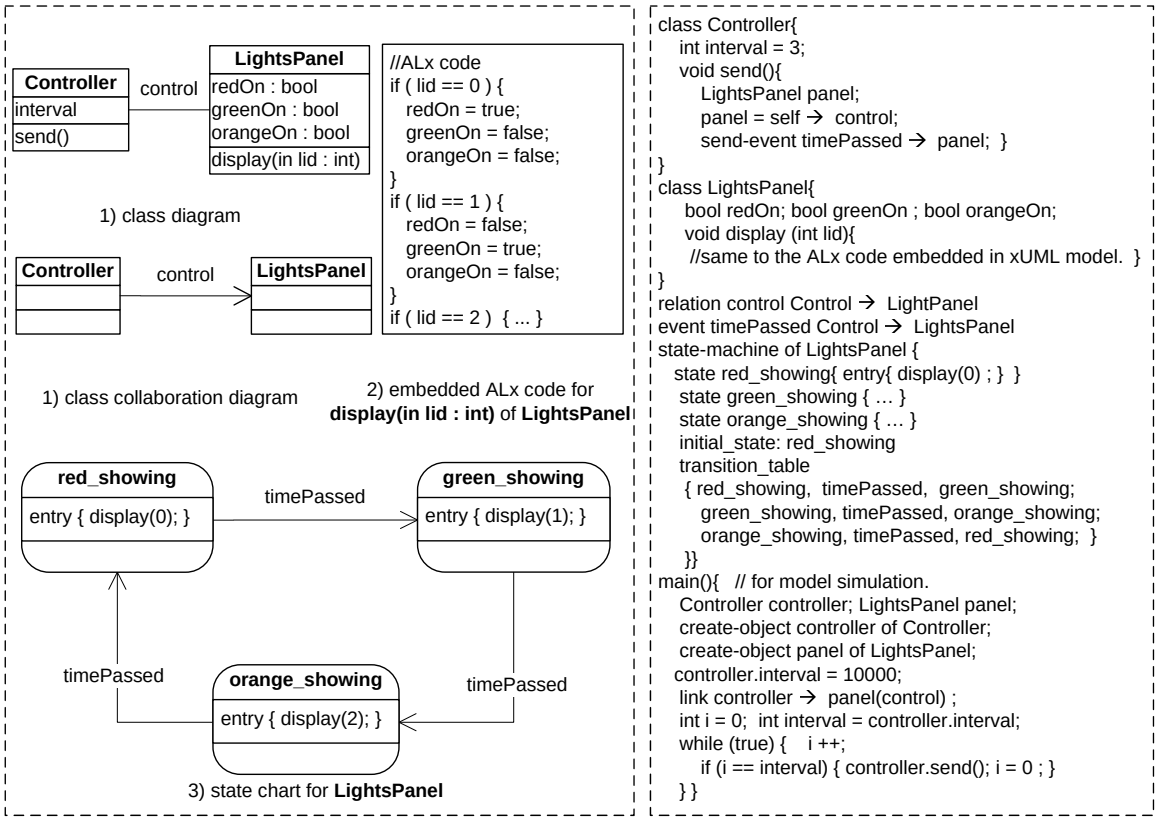
The traffic light system is made up of two objects: a Controller and a LightsPanel. The former, which can be regarded as a timer, controls the latter by sending a timing message on a regular basis. For instance, when one minute passes, the controller sends an event called 'timePassed'. And the LightsPanel responds to the event by switching on a light as desired after turning off the current illuminated one. The xUML model, ALx code and generated Java code of this system are shown in Figure I.7.

```
class C000{
   boolean informed;
   String msg = “msg”;
   void send(){
      C001 next = self → R0000001;
      next.receive(msg);
      prints(“C000 completes sending.”);
   }
   void receive(String msg){
      self.msg = msg;
      self >> informed;
       prints(“C000 competes receiving.”);
   }
}

class C001{
   boolean informed;
   String msg = “msg”;
   void send(){
      C002 next = self → R0010002;
      next.receive(msg);
      prints(“C001 complete sending.”);
   }
   void receive(String msg){
      self.msg = msg;
      C000 previous = self → R001000;
      self >> informed;
      send-event ack0 →  previous;
       prints(“C001 competes receiving.”);
   }
}

class C002{
   boolean informed;
   String msg = “msg”;
   void send(){
      C003 next = self → R002003;
      next.receive(msg);
      prints(“C002 completes sending.”);
   }
   void receive(String msg){
      self.msg = msg;
      C001 previous = self → R002001;
      self >> informed;
      send-event ack1 →  previous;
       prints(“C002 competes receiving.”);
   }
}
```

```
// The  C003 ... C 998 are
//  omitted here.

class C999{
   boolean informed;
   String msg = “msg”;
   void send(){
   }
   void receive(String msg){
      self.msg = msg;
      C998 previous = self → R999998
      self >> informed;
      send-event ack998 → previous;
       prints(“C999 competes receiving.”);
   }
}

// The state machine of C000.

state_machine_of C000{
   state Uninformed {
      entry{
        self.informed = true:
        send();
      }
      exit { }
   }

   state Informed{
      entry {}
      exit {}
   }

   state Acknowledged {
      entry{
        prints(“acknowledged”);
      }

      exit { }
   }
   initial_state: Ignorant
   transition_table{
      Uninformed, informed, Informed;
      Informed, ack0, Acknowledged
   }
}
// The state charts  of C001-C999 are
// ignored here.
```

```
// Relation declarations.

relation R000001 C000 → C001;
relation R001000 C001 → C000;
relation R001002 C001 → C002;
relation R002001 C002 → C001;
…
relation R998999 C998 → C999;
relation R999998 C999 → C998;

// Event declarations.

event ack0 C001→ C000;
event ack1 C002 → C001;
…
event ack998 C999 → C998;
….

// The main method of this system.

main(){
   C000 c000;
   C000 c001;
   C000 c002;
   C000 c003;
   …
   C999 c999;

   create-object c000 of  C000;
   create-object c001 of  C001;
   create-object c002 of  C002;
   …
   create-object c999 of  C999;

   link c000 →  c001 (R000001);
   link c001 →  c000 (R001000);
   link c001 →  c002 (R001002);
   link c002 →  c001 (R002001);
   ...
   link c998 →  c999 (R998999);
   link c999 →  c998 (R999998);

   c000.send();
}
```

Figure I.6: ALx code of the toy message relay system.



a) xUML model of traffic light system (input model)

b) ALx code generated

```
class Controller extends ALObject
                implements IALObject, IController{
    int interval;
    void send () {
        LightsPanel panel;
        panel = this.getLinkedObject(RIDs.Rcontrol);
        panel.stateTransitted (EIDs.EtimePassed);
    }
}

Class LightsPanel extends ALObject
                implements IALObject, ILIghtsPanel{
    bool redOn;
    bool greenOn;
    bool orangeOn;
    void display(int id){
        if ( lid = 0 ) {
            redOn = true;
            greenOn = false;
            orangeOn = false;
        }
        if ( lid = 1 ) {
            redOn = false;
            greenOn = true;
            orangeOn = false;
        }
        if ( lid =2 ) {
            redOn = false;
            greenOn = false;
            orangeOn = true;
        }
    }
}

// The main class for model simulation.
public class Main {
    public static void main(String[] argus){
        Controller controller;
        LightsPanel panel;
        controller = Controller.newinstance();
        panel = LightsPanel.newinstance();
        controller = 10000;
        Rcontrol.newLink(controller, panel);
        int i = 0;
        int interval = controller.interval;
        while (true) { i ++;
            if (i == interval) { controller.send(); i = 0 ; }
        }
    }
}
```

c) Java code generated

Figure I.7: Models and code of the traffic light system.

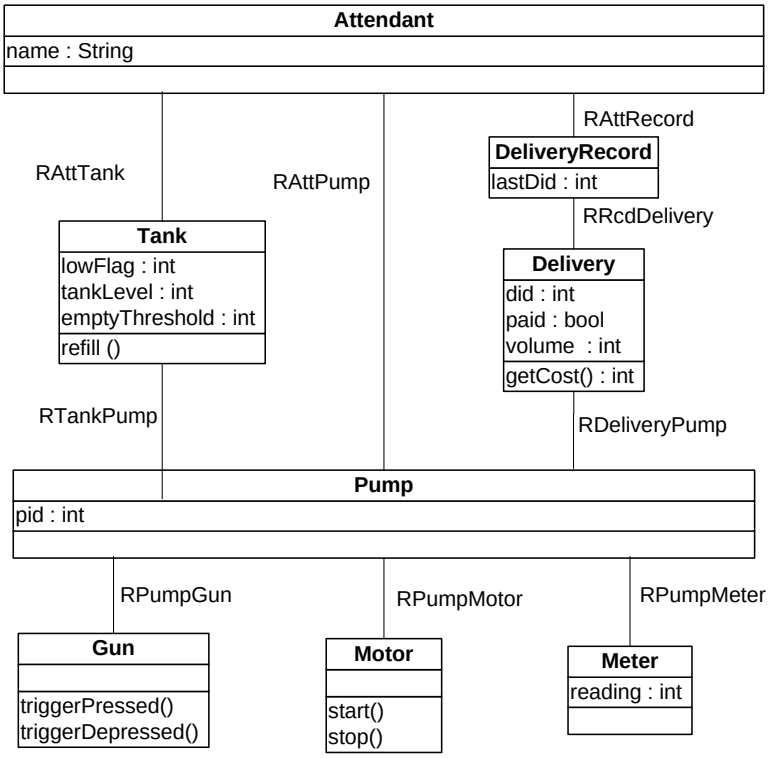
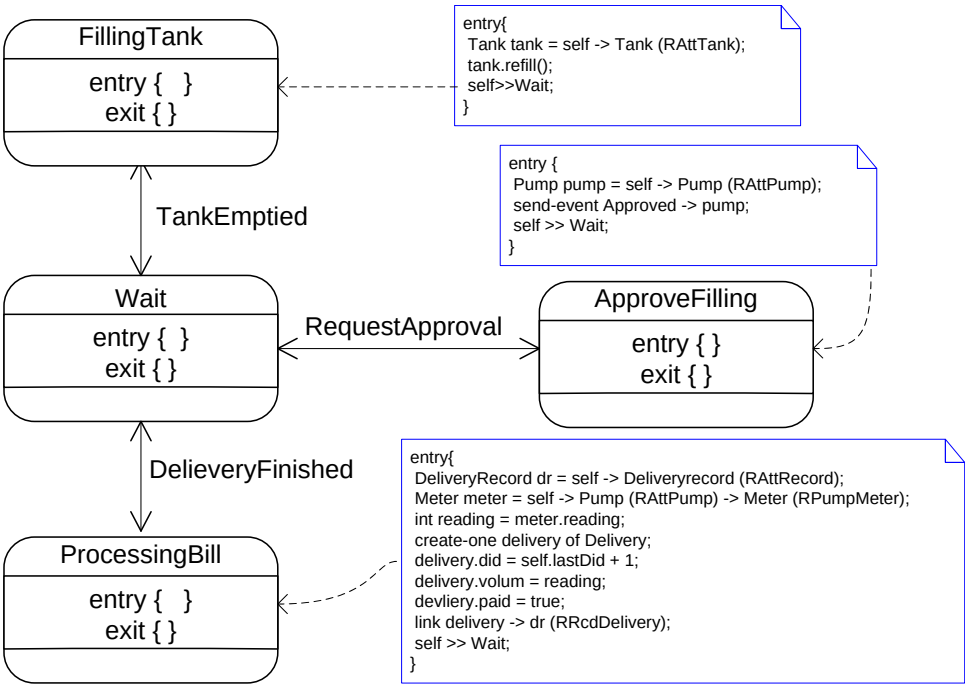


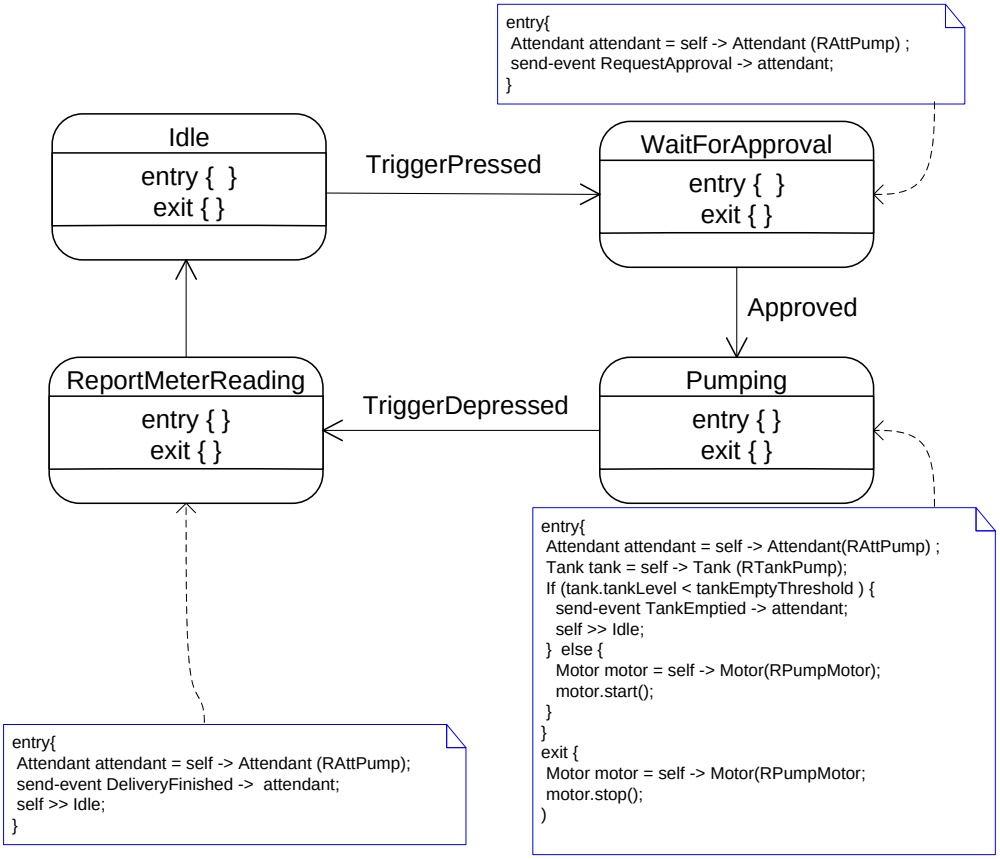
Figure I.8: Class diagram of the gas station system.

I.4 Gas Station System

The gas station system is intended to simulate a filling station, where there are an attendant watching the fuelling process of customers, and a pump, equipped with a filling gun, a motor and a meter, which is controlled by the filling gun. When a customer enters the gas station, takes off the filling gun and presses the trigger, then a message is sent to the attendant for the approval of starting the fuelling process. After the attendant approves it, the pump starts its motor and then gas is pumped into the customer's car. When the customer releases the trigger, this means the fuelling is completed and the motor is commanded to stop. When fuelling is started, the system checks whether the tank has enough fuel. If not, a message is sent to the attendant to refill the tank.



(a) The State chart of Attendant



(b) The State chart of Pump

Figure I.9: State charts of the gas station system.

```
class Attendant {
    String name;
}
class Tank {
    boolean emptyFlag;
    int emptyThreshold;
    int tankLevel;
    void init(){
        lowFlag = false;
        tankLevel = 1000;
    }
    void refill(){
        tankLevel = 1000;
    }
 }
class Pump {
    int pid = 1;
 }
class Gun {
    void triggerPressed(){
        Pump pump = self → Pump (RPumpGun);
        send-event triggerPressed →  pump;
    }
    void triggerDepressed(){
        Pump pump = self → Pump (RPumpGun);
        send-event triggerDepressed →  pump;
    }
 }
class Motor {
    boolean running = false;
    void start(){ running = true; }
    void stop(){ running = stop; }
}
class Meter { }
class DeliveryRecord{ }
class Delivery {
    int Did;
    boolean paid = false;
    int volume;
    int price = 1;

    int getCost() { return price * volume;  }
    suspend () { self >> Suspended; }
}

// Relation declarations are omitted.
// Event declarations are omitted.

state_machine_of Pump{
    initial-state: Idle

    state Idle  {
        entry {self.paid = true;  }
        exit { }
    }
    state WaitForApproval{
         entry{
              Attendant attendant = self -> Attendant (RAttPump) ;
              Send-event RequestApproval -> attendant;
        }
        exit {  }
    }
    state Pumping  {
        entry{
              Attendant attendant = self -> Attendant(RAttPump) ;
              Tank tank = self -> Tank (RTankPump);
               If (tank.tankLevel < tank.emptyThreshould ) {
                    send-event TankEmptied -> attendant;
                    self >> Idle;
                } else {
                    Motor motor = self -> Motor(RPumpMotor);
                    motor.start();
        }
        exit {
              Motor motor = self -> Motor(RPumpMotor;
               motor.stop();
        }
}
    }
    state ReportMeterReading  {
         entry{
 Attendant attendant = self -> Attendant (RAttPump);
 send-event DeliveryFinished ->  attendant;
 self >> Idle;
  }
        exit {  }
    }
        transition-table{}
    }
}
// The state machine of Attendant is omitted here.
// The main method.
main(){
// configure the initial state of the system.
    Attendant attendant;  Tank tank;  DeliveryRecord drcd;
    Pump pump;  Gun gun;   Motor motor;  Meter meter;

    create-object attendant of Attendant;
    create-object tank of Tank;
    tank.init();
    create-object drcd of DeliveryRecord;
    create-object pump of Pump;
    create-object gun of Gun;
    create-object motor of Motor;
    create-object meter of Meter;

    link attendant → pump (RAttPump);
    link attendant → tank (RAttTank);
    link attendant → drcd (RAttRecord);
    link tank → pump (RTankPump);
    link pump → gun (RPumpGun);
    link pump → motor (RPumpMotor);
    link pump → meter (RPumpMeter);

 // configure 8 example ContracTravelelrs.
    gun.triggerPressed();
    // hold the gun trigger for some time, we using the following
    // code to simulate the fuelling.
    int i = 0;
    while ( I < 10000 ) {
        meter. reading = meter.reading + 1;
        tank.tankLevel  = tank.tankLevel -1;
    }
    gun.triggerDepressed();
}
```

Figure I.10: ALx code of the gas station system.

I.5 Elevating System

In Chapter 4, we have presented the xUML model and the corresponding ALx code of the elevating system. Here the generated Java code of this system is provided.

```
/*****************************************/
// From the ALx class 'Controller'.

package elevator;
import library.*;

public interface IController extends IALObject {
  public void upButPressed();
  public void downButPressed();
  public void doorButPressed();
  public void stopButPressed();
}


package elevator;
import library.*;
import java.util.LinkedList;
import java.util.Iterator;

public class Controller extends ALObject implements IController {
  public static ObjectList objectList = new ObjectList();
  public static Controller newinstance(){
     Controller temp = new Controller();
     recordObject(temp);
       return temp;
  }

  public static void recordObject(ALObject alo){
    objectList.addObject(alo);
  }

  public static void destroyInstance(ALObject alo){
    objectList.deleteObject(alo);
  }

  public void upButPressed() {
    Elevator elevator;
    elevator = (Elevator)this.getLinkedObject(RIDs.RR4);
    elevator.stateTransmitted(EIDs.EMoveUp);
  }

  public void downButPressed(){
    Elevator elevator;
    elevator = (Elevator)this.getLinkedObject(RIDs.RR4);
```

```
    elevator.stateTransmitted(EIDs.EMoveDown);
  }

  public void doorButPressed() {
    Elevator elevator;
    elevator = (Elevator)this.getLinkedObject(RIDs.RR4);
    elevator.stateTransmitted(EIDs.ESwitchDoor);
  }

  public void stopButPressed() {
    Elevator elevator;
    elevator = (Elevator)this.getLinkedObject(RIDs.RR4);
    elevator.stateTransmitted(EIDs.EStop);
    LinkedList<IALObject> mbs = this.getLinkedObjects(RIDs.RR2);
    ((IMovingButton)(mbs.get(0))).deIlluminate();
    ((IMovingButton)(mbs.get(1))).deIlluminate();
  }
}

/*******************************************/
// From the ALx class 'Door'.

package elevator;
import library.*;

public interface IDoor extends IALObject {
  public void open();
  public void close();
}

package elevator;
import library.*;

public class Door extends ALObject implements IDoor{

  public static ObjectList objectList = new ObjectList();

  public static Door newinstance(){
     Door temp = new Door();
     recordObject(temp);
       return temp;
  }

  public static void recordObject(ALObject alo){
    objectList.addObject(alo);
  }
  public static void destroyInstance(ALObject alo){
    objectList.deleteObject(alo);
  }

  public void open(){
    System.out.println("The door is called to open");
```

```
  }

  public void close(){
    System.out.println("The door is called to close");
  }

}

/*******************************************/
// From the ALx class 'DoorSwitchButton'.

package elevator;
import library.*;

public interface IDoorSwitchButton extends IALObject {
  public void pressed();
}

package elevator;
import library.*;

public class DoorSwitchButton extends ALObject
                            implements IDoorSwitchButton {
  public static ObjectList objectList = new ObjectList();
  public static DoorSwitchButton newinstance(){
     DoorSwitchButton temp = new DoorSwitchButton();
     recordObject(temp);
       return temp;
  }

  public static void recordObject(ALObject alo){
    objectList.addObject(alo);
  }

  public static void destroyInstance(ALObject alo){
    objectList.deleteObject(alo);
  }

  public void pressed(){
    Controller controller;
    controller = (Controller)this.getLinkedObject(RIDs.RR1);
    controller.doorButPressed();
  }
}

/*******************************************/
// From the ALx class 'DownButton'.

package elevator;
import library.*;
```

```
public interface IDownButton extends IMovingButton, IALObject{
  public void pressed();
}

package elevator;
import library.*;
public class DownButton extends ALObject
                  implements IDownButton, IMovingButton {
  public static ObjectList objectList = new ObjectList();

  // The reference to an object for the super class mechanism.
  // This object is used to delegate the properties of
  // the superclass. Note we use 'new' to create such an object
  // instead of using newinstance() because the later will add
  //  this object to the object list, the former not.

  private MovingButton parent_delegator = new MovingButton();

  public static DownButton newinstance(){
     DownButton temp = new DownButton();
     recordObject(temp);
       return temp;
  }
  public static void recordObject(ALObject alo){
    objectList.addObject(alo);
    // Only the class of a super class has the next line.
    MovingButton.recordObject(alo);
  }


  public static void destroyInstance(ALObject alo){
    objectList.deleteObject(alo);
    MovingButton.destroyInstance(alo);
  }
  public void pressed(){
    Controller controller;
    controller = (Controller)this.getLinkedObject(RIDs.RR2);
    controller.downButPressed();
    this.setIlluminated(true);
  }

  public void deIlluminate() {
    parent_delegator.deIlluminate();

  }

  public void illuminate() {
    parent_delegator.illuminate();
  }

  public boolean getIlluminated() {
    return parent_delegator.getIlluminated();
```

```
  }

  public void setIlluminated(boolean value) {
    parent_delegator.setIlluminated(value);
  }

}

/*****************************************/
//From the ALx class 'Elevator'.
package elevator;
import library.*;

public interface IElevator extends IALObject {
  public void move(boolean direction);
  public void stop();
}

package elevator;
import library.*;

public class Elevator extends ALObject
                      implements IALObject, IElevator {
  public static ObjectList objectList = new ObjectList();
  public static Elevator newinstance(){
     Elevator temp = new Elevator();
     recordObject(temp);
       return temp;
  }

  public static void recordObject(ALObject alo){
    objectList.addObject(alo);
  }

  public static void destroyInstance(ALObject alo){
    objectList.deleteObject(alo);
  }

  public Elevator(){
    sm = this.new SM();
  }

  public void move(boolean direction) {
    if (direction == true)
      System.out.println("The elevator is moving up");
    if (direction == false)
      System.out.println("The elevator is moving down");
  }

  public void stop() {
    System.out.println("The elevator is stopped.");
  }
```

```
/* The following is for simulating the state machine */
class SM extends StateMachine {

  public State StoppedWithDoorOpened =
                                  new StoppedWithDoorOpened();
  public State StoppedWithDoorClosed =
                                   new StoppedWithDoorClosed();
  public State MovingUp = new MovingUp();
  public State MovingDown = new MovingDown();

  public SM(){
    this.addEntry(StoppedWithDoorOpened,
                    EIDs.ESwitchDoor, StoppedWithDoorClosed);
    this.addEntry(StoppedWithDoorClosed,
                    EIDs.EMoveUp, MovingUp);
    this.addEntry(StoppedWithDoorClosed,
                    EIDs.EMoveDown, MovingDown);
    this.addEntry(StoppedWithDoorClosed,
                    EIDs.ESwitchDoor, StoppedWithDoorOpened);
    this.addEntry(MovingUp,
                    EIDs.EStop, StoppedWithDoorClosed);
    this.addEntry(MovingDown,
                    EIDs.EStop, StoppedWithDoorClosed);
    currentState = this.StoppedWithDoorOpened;
  }

  class StoppedWithDoorOpened extends State{
    public void entry() {
      System.out.println("Enter State: StoppedWithDoorOpened");
      Door door;
      door = (Door)(Elevator.this.getLinkedObject(RIDs.RR5));
      door.open();
    }
    public void exit(){
      Door door;
      door = (Door)(Elevator.this.getLinkedObject(RIDs.RR5));
      door.close();
      System.out.println("Exit State: StoppedWithDoorOpened");
    }
  }

  class StoppedWithDoorClosed extends State{
    public void entry(){
      System.out.println("Enter State: StoppedWithDoorClosed");
    }
    public void exit(){
      System.out.println("Exit State: StoppedWithDoorClosed");
    }
  }

  class MovingUp extends State{
```

```
      public void entry(){
        System.out.println("Enter State: MovingUp");
        move(true);
      }
      public void exit(){
        System.out.println("Exit State: MovingUp");
        stop();
      }
    }
    class MovingDown extends State{
      public void entry(){
        System.out.println("Enter State: MovingDown");
        move(false);
      }
      public void exit(){
        System.out.println("Exit State: MovingDown");
        stop();
      }
    }

  }
}

/*******************************************/
//  From the ALx class 'StopButton'.

package elevator;
import library.*;

public interface IStopButton extends IALObject {
  public void pressed();
}
package elevator;

import library.*;

public class StopButton extends ALObject
                                implements IStopButton {
  public static ObjectList objectList = new ObjectList();

  public static StopButton newinstance(){
     StopButton temp = new StopButton();
     recordObject(temp);
       return temp;
  }

  public static void recordObject(ALObject alo){
    objectList.addObject(alo);
  }

  public static void destroyInstance(ALObject alo){
    objectList.deleteObject(alo);
```

```
  }

  public void pressed(){
    Controller controller;
    controller = (Controller)this.getLinkedObject(RIDs.RR3);
    controller.stopButPressed();
  }
}

/*******************************************/
// From the ALx class 'UpButton'.
package elevator;
import library.*;

public interface IUpButton extends IMovingButton, IALObject {
  public void pressed();
}

package elevator;
import library.*;
import library.ObjectList;

public class UpButton extends ALObject
            implements IALObject, IUpButton, IMovingButton {
  public static ObjectList objectList = new ObjectList();
  private MovingButton parent_delegator = new MovingButton();
  public static UpButton newinstance(){
     UpButton temp = new UpButton();
     recordObject(temp);
     return temp;
  }

  public static void recordObject(ALObject alo){
    objectList.addObject(alo);
    MovingButton.recordObject(alo);
  }

  public static void destroyInstance(ALObject alo){
    objectList.deleteObject(alo);
    MovingButton.destroyInstance(alo);
  }

  public void pressed(){
    Controller controller;
    controller = (Controller)this.getLinkedObject(RIDs.RR2);
    controller.upButPressed();
    illuminate();
  }

  public void deIlluminate() {
    parent_delegator.deIlluminate();
  }
```

```
  public void illuminate() {
    parent_delegator.illuminate();

  }

  public boolean getIlluminated() {
    return parent_delegator.getIlluminated();
  }

  public void setIlluminated(boolean value) {
    parent_delegator.setIlluminated(value);
  }
}


/*******************************************/
// The generated classes for relations.

package elevator;
import library.*;

import library.ALObject;

public class RR1 extends ALRelation {
  public static int rid = RIDs.RR1;
  public static int oneEnd = CIDs.DoorSwitchButton;
  public static int anotherEnd = CIDs.Controller;

  public static int getAnotherEnd(){
    return anotherEnd;
  }

  public static int getOneEnd(){
    return oneEnd;
  }

  public static int getRid(){
    return rid;
  }

  public static ALLink newLink(ALObject o1, ALObject o2){
    ALLink temp = new ALLink(rid, o1, o2);
    o1.addLink(temp);
    o2.addLink(temp);
    return null;
  }
}

package elevator;
import library.*;
```

```
public class RR2 extends ALRelation {
  public static int rid = RIDs.RR2;
  public static int oneEnd = CIDs.MovingButton;
  public static int anotherEnd = CIDs.Controller;

  public static int getAnotherEnd(){
    return anotherEnd;
  }

  public static int getOneEnd(){
    return oneEnd;
  }

  public static int getRid(){
    return rid;
  }

  public static ALLink newLink(ALObject o1, ALObject o2){
    ALLink temp = new ALLink(rid, o1, o2);
    o1.addLink(temp);
    o2.addLink(temp);
    return null;
  }
}
package elevator;
import library.*;

public class RR3 extends ALRelation {
  public static int rid = RIDs.RR3;
  public static int oneEnd = CIDs.StopButton;
  public static int anotherEnd = CIDs.Controller;

  public static int getAnotherEnd(){
    return anotherEnd;
  }

  public static int getOneEnd(){
    return oneEnd;
  }

  public static int getRid(){
    return rid;
  }

  public static ALLink newLink(ALObject o1, ALObject o2){
    ALLink temp = new ALLink(rid, o1, o2);
    o1.addLink(temp);
    o2.addLink(temp);
    return null;
  }
}
```

```
package elevator;
import library.*;

public class RR4 extends ALRelation {
  public static int rid = RIDs.RR4;
  public static int oneEnd = CIDs.Controller;
  public static int anotherEnd = CIDs.Elevator;

  public static int getAnotherEnd(){
    return anotherEnd;
  }

  public static int getOneEnd(){
    return oneEnd;
  }

  public static int getRid(){
    return rid;
  }

  public static ALLink newLink(ALObject o1, ALObject o2){
    ALLink temp = new ALLink(rid, o1, o2);
    o1.addLink(temp);
    o2.addLink(temp);
    return null;
  }
}

package elevator;
import library.*;

public class RR5 extends ALRelation {
  public static int rid = RIDs.RR5;
  public static int oneEnd = CIDs.Elevator;
  public static int anotherEnd = CIDs.Door;

  public static int getAnotherEnd(){
    return anotherEnd;
  }

  public static int getOneEnd(){
    return oneEnd;
  }

  public static int getRid(){
    return rid;
  }

  public static ALLink newLink(ALObject o1, ALObject o2){
    ALLink temp = new ALLink(rid, o1, o2);
    o1.addLink(temp);
    o2.addLink(temp);
```

```
    return null;
  }
}

/*******************************************/
// The folling 'ID' classes are generated
// based on the information obtained in the
// first-pass scan of the ALx code of the system.
package elevator;

public class CIDs {
  // Class ID
  public final static int Controller = 11001;
  public final static int Elevator = 11002;
  public final static int Door = 11003;
  public final static int MovingButton = 11004;
  public final static int UpButton = 11005;
  public final static int DownButton = 11006;
  public final static int DoorSwitchButton = 11007;
  public final static int StopButton = 11008;
}


package elevator;

public class RIDs {
  // Relation Id.
  public final static int RR1 = 22001;
  public final static int RR2 = 22002;
  public final static int RR3 = 22003;
  public final static int RR4 = 22004;
  public final static int RR5 = 22005;
}


package elevator;

public class EIDs {

  //Event ID.
  public final static int ESwitchDoor = 33001;
  public final static int EMoveUp = 33002;
  public final static int EMoveDown = 33003;
  public final static int EStop= 33004;

}



/*******************************************/
/*              Main Class                 */
/*******************************************/
```

```
package elevator;

public class Main {

  /**
   * @param args
   */
  public static void main(String[] args) {

    DoorSwitchButton doorSwitchButton;
    UpButton upButton;
    DownButton downButton;
    StopButton stopButton;
    Controller controller;
    Elevator elevator;
    Door door;

    doorSwitchButton= DoorSwitchButton.newinstance();
    upButton = UpButton.newinstance();
    downButton = DownButton.newinstance();
    stopButton = StopButton.newinstance();
    controller = Controller.newinstance();
    elevator = Elevator.newinstance();
    door = Door.newinstance();


    RR1.newLink(doorSwitchButton, controller);
    RR2.newLink(upButton, controller);
    RR2.newLink(downButton, controller);
    RR3.newLink(stopButton, controller);
    RR4.newLink(controller, elevator);
    RR5.newLink(elevator, door);

    System.out.println("1. One building staff" +
                       "enters the elevator...");
    System.out.println("2. He clsoed the door...");
    doorSwitchButton.pressed();

    System.out.println("3. He pressed the upButton" +
                        "to go up... ");
    upButton.pressed();

    System.out.println("4. For some reason, he stops" +
                  " the elevator in the middle way...");
    stopButton.pressed();


    System.out.println("5. He presses the down" +
                            " button to move down...");
    downButton.pressed();

    System.out.println("6. He stops somewhere...");
```

```
    stopButton.pressed();

    System.out.println("7. He opens the door" +
                                    "of the elevator...");
    doorSwitchButton.pressed();

    System.out.println("8. He steps out of the elevator.");
  }

}
```

Appendix J

Excerptions of ATL Transformations

This appendix presents an excerpt of the xUML-to-ALx transformation rules for two purposes: to give the reader some idea of ATL and to show that xUML is intuitively mapped to ALx. For a complete manual of ATL, the reader is referred to [12].

```
-------------------------------------------------------------------------
--                      State Machine Translation
-------------------------------------------------------------------------


rule State{
from
    input:xumlmm!UState
to
    output:alxmm!AStateDeclaration(
     entry <- input.entry.specification,
exit <- input.exit.specification,
    name <- input.name
    )
}
rule StateMachine{
from
    input:xumlmm!UStateMachine
```

```
to
    output:alxmm!AStateMachineDeclaration(
     state <- input.states,
ofclass <- input.ofclass,
initialState  <- input.getInitialState()
    )
do{
   output.transitionTable <- thisModule.
    NewTransitionTable(input.transitions);
}
}
-- called rule.
rule NewTransitionTable(transitions:Set(xumlmm!UTransition)){
to
output: alxmm!ATransitionTable(

)
do{
output.entries <- transitions->
collect(t|thisModule.NewTransitionEntry(t));
output;  --  this line is necessary.
}
}
-- called rule
rule NewTransitionEntry(transition:xumlmm!UTransition){
to
output: alxmm!ATransitionEntry(
sourceState <- transition.source.name,
targetState <- transition.target.name,
-- the name of a transition is the triggering event name.
triggerEvent <- transition.name
)
do{
```

```
output;  -- this line is necessary.
}
}

-- This helper looks for the name of the initial state.
-- CONTEXT: xumlmm!UStateMachine
-- RETURN: String
helper context xumlmm!UStateMachine def: getInitialState() : String =
let states:Set(xumlmm!UState) = self.states in
  states->any(state|state.initial = true).name
```

Bibliography

[1] Official Apache Ant Project Website. Http://ant.apache.org/. Inspected on 20/05/09.

[2] Official Eclipse M2M Project Website. Http://www.eclipse.org/modeling/m2m/. Inspected on 20/05/09.

[3] Official Eclipse M2T Project Website. Http://www.eclipse.org/modeling/m2t/. Inspected on 20/05/09.

[4] Official Eclipse Modeling Framework Project (EMF) Website. Http://www.eclipse.org/modeling/emf/. Inspected on 20/05/09.

[5] Official Eclipse Website. Http://www.eclipse.org/. Inspected on 20/05/09.

[6] Official JavaCC Website. Http://javacc.dev.java.net/. Inspected on 20/05/09.

[7] Official RAT Project Website. Http://www.cin.ufpe.br/~rat/. Inspected on 20/05/09.

[8] H. Abelson and J. Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill, 2 edition, 1996.

[9] S. Abramsky and A. Jung. Domain Theory. *Handbook of Logic in Computer Science*, 3:1–168, 1994.

[10] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*. Addison-Wesley Reading, Mass, 1977.

[11] J. M. Álvarez, T. Clark, A. Evans, and P. Sammut. An Action Semantics for MML. In M. Gogolla and C. Kobryn, editors, *Proceedings of The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Con-*

ference, UML 2001., volume 2185 of *Lecture Notes in Computer Science*, pages 2–18, Toronto, Canada, October 2001. Springer.

[12] ATLAS group. *ATL: Atlas Transformation Language, ATL User Manual, Version 0.7*, 2006. Available at http://www.eclipse.org. Inspected on 20/05/09.

[13] F. Belina and D. Hogrefe. The CCITT-specification and description language SDL. *Computer Networks and ISDN Systems*, 16(4):311–341, 1989.

[14] J. A. Bergstra, J. Heering, and P. Klint. *Algebraic Specification*. ACM Press New York, NY, USA, 1989.

[15] A. Bondorf and J. Palsberg. Compiling Actions by Partial Evaluation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 308–317. ACM New York, NY, USA, 1993.

[16] A. Bondorf and J. Palsberg. Generating Action Compilers by Partial Evaluation. *Journal of Functional Programming*, 6(02):269–298, 2008.

[17] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 2nd edition, 2005.

[18] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In R. Teodor, editor, *Proceedings of Algebraic Methodology and Software Technology. 8th International, Conference, AMAST 2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 293–308, Iowa City, Iowa, USA,, 2000. Springer.

[19] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele, editors, *Proceedings of Abstract State Machines, Theory and Applications, International Workshop, ASM 2000*, volume 1912 of *Lecture Notes in Computer Science*, pages 223–241, Monte Verita Switzerland, 2000. Springer.

[20] D. F. Brown, H. Moura, and D. A. Watt. Actress: An Action Semantics Directed Compiler Generator. In U. Kastens and P. Pfahler, editors, *Proceedings*

of Compiler Construction, 4th International Conference on Compiler Construction, CC'92, volume 641 of *Lecture Notes in Computer Science*, pages 95–109, Paderborn, Germany, October 1992. Springer.

[21] D. F. Brown and D. A. Watt. JAS: A Java Action Semantics. In *Proceedings of 2nd International Workshop on Action Semantics.*, pages 43–56. University of Aarhus, Denmark: BRICS NS, 1999.

[22] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[23] R. G. Cattell and D. K. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[24] N. Chomsky. Three Models for the Description of Language. *Information Theory, IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[25] T. Clark, A. Evans, and S. Kent. The Metamodeling Language Calculus: Foundation Semantics for UML. In H. Hußmann, editor, *Proceedings of Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001*, volume 2029 of *Lecture Notes in Computer Science*, pages 17–31, Genova, Italy, 2001. Springer.

[26] W. Damm, B. Josko, A. Pnueli, and A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-time UML. In F. Boer, M. Bonsangue, S. Graf, and W. Roever, editors, *Proceedings of Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Revised Lectures*, volume 2582 of *Lecture Notes in Computer Science*, pages 72–99, Leiden, The Netherlands, November 2003. Springer.

[27] T. Dinh-Trong, S. Ghosh, and R. France. JAL: Java like Action Language. Department of Computer Science, Colorado State University, 2006.

[28] G. Engels, R. Heckel, and S. Sauer. UML: A Universal Modeling Language. In M. Nielsen and D. Simpson, editors, *Proceedings of Application and Theory of Petri Nets 2000, 21st International Conference, ICATPN 2000*, volume 1825

of *Lecture Notes in Computer Science*, pages 24–38, Aarhus, Denmark, 2000. Springer.

[29] A. Evans and S. Kent. Core Meta-Modeling Semantics of UML: the pUML Approach. In R. B. France and B. Rumpe, editors, *Proceedings of The Unified Modeling Language - Beyond the Standard, Second International Conference, UML'99*, volume 1723 of *Lecture Notes in Computer Science*, pages 140–155, Fort Collins, CO, USA, 1999. Springer.

[30] R. Farrow. LINGUIST-86: Yet Another Translator Writing System Based on Attribute Grammars. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction, SIGPLAN '82*, pages 160–171, New York, NY, USA, 1982. ACM.

[31] S. Flake and W. Müller. An ASM Definition of the Dynamic OCL 2.0 Semantics. In T. Baar, A. Strohmeier, M. D. Moreira, and S. J. Mellor, editors, *Proceedings of The Unified Modelling Language: Modelling Languages and Applications, 7th International Conference, UML 2004*, volume 3273 of *Lecture Notes in Computer Science*, pages 226–240, Lisbon, Portugal, 2004. Springer.

[32] R. France, A. Evans, K. Lano, and B. Rumpe. The UML as a Formal Modeling Notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

[33] E. Gammar, R. Helm, R. Johnson, and J. Vlissides. *Design Pattern*. Addison-Wesley, 1995.

[34] T. Gehrke, U. Goltz, and H Wehrheim. The Dynamic Models of UML: Towards a Semantics and its Application in the Development Process, November 1998. Institut für Informatik Universität Hildesheim.

[35] D. W. Gonzalez. *Ada Programmer's Handbook*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[36] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, the Java Series* . Addison-Wesley Professional, 2nd edition, 2005.

[37] D. Grune, C. Jacobs, K. Langendoen, and H. Bal. *Modern Compiler Design*. John Wiley & Sons, Inc. New York, NY, USA, 2000.

[38] B. S. Hansen and J. U. Toft. The Formal Specification of ANDF: An Application of Action Semantics. In *Proceedings of the First International Workshop on Action Semantics*, number NS-94-1 in BRICS Notes, pages 34–42, Edinburgh, Scotland, April 1994.

[39] D. Harel and S. Maoz. Assert and Negate Revisited: Modal Semantics for UML Sequence Diagrams. *Software and System Modeling*, 7(2):237–252, 2008.

[40] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer*, 37(10):64–72, 2004.

[41] S. Haustein and J. Pleumann. OCL as Expression Language in an Action Semantics Surface Language. In O. Patrascoiu, editor, *Proceedings of OCL and Model Driven Engineering*, pages 99–113. University of Kent, 2004.

[42] R. M. Herndon and V. A. Berzins. The Realizable Benefits of a Language Prototyping Language. *IEEE Transactions on Software Engineering*, 14(6):803–809, 1988.

[43] M. Ibrahim, A. Fedorec, and K. Rennolls. Executable UML – UML 2 – the Need for xUML. Available at http://www.kc.com. Inspected on 20/05/09, Feb. 2003.

[44] S. C. Johnson. Yacc: Yet Another Compiler Compiler. *UNIX Programmer's Manual, AT&T Bell Laboratories*, 2:353–387, 1979.

[45] F. Jouault and J. Bézivin. KM3: A DSL for Metamodel Specification. In R. Gorrieri and H. Wehrheim, editors, *Proceedings of Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference, FMOODS 2006.*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185, Bologna, Italy, June 2006. Springer.

[46] F. Jouault and I. Kurtev. Transforming Models with ATL. In J. Bruel, editor, *Proceeding of Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Montego Bay, Jamaica, October 2006. Springer.

[47] Kabira Technologies, Inc. Kabira Action Semantics, 2004. Available at http://www.kabira.com. Inspected on 20/05/09.

[48] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer, New York., 1982.

[49] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: the Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2005.

[50] K. Koskimies, O. Nurmi, and J. Pakki. The Design of a Language Processor Generator. *Software Practice and Experience*, 18(2):107–135, 1988.

[51] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In M. Gogolla and C. Kobryn, editors, *Proceedings of The Unified Modeling Language, Modeling Languages, Concepts, and Tools, 4th International Conference, UML 2001*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256, Toronto, Canada, 2001. Springer.

[52] M. Kyas, H. Fecher, F. Boer, J. Jacob, J. Hooman, M. Zwaag, and H. Kugler. Formalizing UML Models and OCL Constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47, 2005.

[53] S. B. Lassen, P. D. Mosses, and D. A. Watt. An Introduction to AN-2: The Proposed New Version of Action Notation. In D. P. Mosses and H. Moura, editors, *Proceedings of the Third International Workshop on Action Semantics, AS 2000*, BRICS Notes Series, pages 19–36, Recife, Brazil, May 2000. Dept. of Comput. Sci. of Aarhus.

[54] L. Lavagno, G. Martin, and B. V. Selic. *UML for Real: Design of Embedded Real-Time Systems*. Springer, 1st edition, 2003.

[55] X. Li, Z. Liu, and J. He. A Formal Semantics of UML Sequence Diagrams. In *Proceedings of Australian Software Engineering Conference, ASWEC'2004*, pages 168– 177, Melbourne, Australia, 2004. IEEE Computer Sciety.

[56] S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343. ACM New York, NY, USA, 1995.

[57] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[58] W. E. McUmber and H. C. Cheng. A General Framework for Formalizing UML with Formal Languages. In *Proceedings of the 23rd International Conference on Software Engineering, ICSE '01*, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society.

[59] S. J. Mellor and M. J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, 2003.

[60] S. J. Mellor and S. Tockey. Action Semantics for UML. Response to OMG RFP ad/98-11-01 OMG ad/1002-08-04. Project Technology, Inc, 2001.

[61] S. J. Mellor, S. Tockey, R. Arthaud, and P. Leblanc. Software-Platform-Independent, Precise Action Specifications for UML. Project Technology, Inc, 2000.

[62] J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group (OMG), 2003. Available at http://www.omg.org. Inspected on 20/05/09.

[63] R. Monson-Haefel and A. K. Weissinger. *Enterprise JavaBeans*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2003.

[64] P. D. Mosses. Unified Algebras and Modules. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL' 89*, pages 329–343. ACM, 1989.

[65] P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.

[66] P. D. Mosses. Unified Algebras and Abstract Syntax. In H. Ehrig, editor, *Recent Trends in Data Type Specification, 9th Workshop on Specification of Abstract Data Types, Caldes de Malavella, 1992*, volume 785 of *Lecture Notes in Computer Science*, pages 280–294, Caldes de Malavella, Spain, October 1994. Springer-Verlag.

[67] P. D. Mosses. Formal Semantics of Programming Languages: An Overview. *Electr. Notes Theor. Comput. Sci.*, 148(1):41–73, 2006.

[68] P. D. Mosses and D. A. Watt. Pascal Action Semantics, 1993. University of Aarhus. Available at ftp://ftp.brics.dk/Projects/AS/Papers/MossesWatt93DRAFT/pas-0.6.ps.Z. Inspected on 13/07/09.

[69] H. Moura. An Implementation of Action Semantics (Summary). In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming, PLILP '92*, Lecture Notes in Computer Science, pages 477–478, London, UK, 1992. Springer-Verlag.

[70] H. Moura. *Action Notation Transformations*. PhD thesis, University of Glasgow, 1993.

[71] H. Moura and L. Menezes. The Abaco System: An Algebraic Based Action Compiler. In P. D. Mosses and D. A. Watt, editors, *Proceedings of The Second International Workshop on Action Semantics, number NS-99-3 in BRICS Notes Series*, pages 143–154, March 1999.

[72] T. J. Mowbray and R. Zahavi. *The Essential CORBA: Systems Integration Using Distributed Objects*. John Wiley & Sons, Inc. New York, NY, USA, 1995.

[73] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, J., 1992.

[74] I. Ober. More Meaningful UML Models. In *Proceedings of TOOLS Pacific 2000: 37th International Conference on Technology of Object-Oriented Languages and Systems*, pages 146–157, Sydney, Australia, 2000. IEEE Computer Society.

[75] Object Management Group (OMG). *Common Ware House (CWM) Specification*, 2001. Available at http://www.omg.org. Inspected on 20/05/09.

[76] Object Management Group (OMG). *Object Constraint Language Version 2.0*, 2005. Available at http://www.omg.org. Inspected on 20/05/09.

[77] Object Management Group (OMG). *Meta Object Facility (MOF) Core Specification, Version 2.0*, 2006. Available at http://www.omg.org. Inspected on 20/05/09.

[78] Object Management Group (OMG). *Unified Modeling Language: Infrastructure, version 2.0*, 2006. Available at http://www.omg.org. Inspected on 20/05/09.

[79] Object Management Group (OMG). *Unified Modeling Language: Superstructure, version 2.0*, 2006. Available at http://www.omg.org. Inspected on 20/05/09.

[80] Object Management Group (OMG). *MOF 2.0/XMI Mapping, Version 2.1.1*, 2007. Available at http://www.omg.org. Inspected on 20/05/09.

[81] Object Management Group (OMG). *Business Process Definition MetaModel (BPDM), Common Infrastructure: OMG Adopted Specification*, 2008. Available at http://www.omg.org. Inspected on 20/05/09.

[82] Object Management Group (OMG). *Unified Modeling Language Specification. Version 1.5 formal/03-03-01*, March 2003. Available at http://www.omg.org. Inspected on 20/05/09.

[83] P. Ørbæk. OASIS: An Optimizing Action-Based Compiler Generator. In P. Fritzson, editor, *Proceedings of Compiler Construction, 5th International Conference, CC'94.*, volume 786 of *Lecture Notes in Computer Science*, pages 1–15, Edinburgh, April 1994. Springer.

[84] J. Palsberg. An Automatically Generated and Provably Correct Compiler for A Subset of Ada. In J. R. Cordy and M. Barbacci, editors, *Proceedings of the 1992 International Conference on Computer Languages, ICCL'92,*, pages 117–126, Oakland, California, April 1992. IEEE.

[85] T. Pittman and J. Peters. *The Art of Compiler Design: Theory and Practice.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[86] G. D. Plotkin. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[87] Project Technology, Inc. *BridgePoint Action Language (AL) Manual*, 2005. Available at http://www.projtech.com. Inspected on 20/05/09.

[88] R. W. Rasmussen. *A Framework for the UML Meta Model.* PhD thesis, Insititute for Informatics, University of Bergen, April 2000. Page: 26–28.

[89] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting Its Multiview Approach. In H. Hußmann, editor, *Proceedings*

of Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, volume 2029 of *Lecture Notes in Computer Science*, pages 171–186, Genova, Italy, 2001. Springer.

[90] M. Richters and M. Gogolla. On Formalizing the UML Object Constraint Language OCL. In T. W. Ling, S. Ram, and M. L. Lee, editors, *Proceedings of Conceptual Modeling , 17th International Conference on Conceptual Modeling Proceedings, ER '98*, volume 1507 of *Lecture Notes in Computer Science*, pages 449–464, Singapore, 1998. Springer.

[91] C. Rossi, M. Enciso, and I. P. de Guzmán. Formalization of UML State Machines Using Temporal Logic. *Software and System Modeling*, 3:31–54, 2004.

[92] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[93] B. Selic. On the Semantic Foundations of Standard UML 2.0. In M. Bernardo and F. Corradini, editors, *Proceedings of Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004*, volume 3185 of *Lecture Notes in Computer Science*, pages 181–199, Bertinoro, Italy, September 2004. Springer.

[94] J. L. Sierra and A. Fernandez-Valmayor. A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars. *Electronic Notes in Theoretical Computer Science*, 164(2):19–36, 2006.

[95] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[96] H. Störrle. Semantics and Verification of Data Flow in UML 2.0 Activities. *Electr. Notes Theor. Comput. Sci.*, 127(4):35–52, 2005.

[97] H. Storrle and J. H. Hausmann. Towards a Formal Semantics of UML 2.0 Activities. In P. Liggesmeyer, K. Pohl, and M. Goedicke, editors, *Proceedings of Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 64 of *LNI*, pages 117–128, Essen, November 2005. GI.

[98] J. E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. MIT Press Cambridge, MA, USA, 1977.

[99] International Telecommunication Union. Specification and Description Language (SDL). Technical Report Z.100, ITU, 1999.

[100] M. van den Brand, J. Iversen, and P. D. Mosses. An Action Environment. *Sci. Comput. Program.*, 61(3):245–264, 2004.

[101] A. van Deursen, J. Heering, and P. Klint. Language Prototyping: An Algebraic Specification Approach. *Language*, 5:307–322.

[102] A. van Deursen and P. D. Mosses. ASD: The Action Semantic Description Tools. In M. Wirsing and M. Nivat, editors, *Proceedings of Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96.*, volume 1101 of *Lecture Notes in Computer Science*, pages 579–582, Munich, Germany, July 1996. Springer.

[103] V. Vitolins and A. Kalnins. Semantics of UML 2.0 Activity Diagram for Business Modeling by Means of Virtual Machine. In *Proceedings of Ninth IEEE International Enterprise Distributed Object Computing Conference, EDOC 2005*, pages 181–194, Enschede, The Netherlands, 2005. IEEE Computer Society.

[104] D. A. Watt. An Action Semantics of Standard ML. In *Proceedings of the 3rd Workshop on Mathematical Foundations of Programming Language Semantics*, pages 572–598, London, UK, 1988. Springer-Verlag.

[105] D. A. Watt. JOOS Action Semantics. Version 1. Available at http://www.dcs.gla.ac.uk/~daw/publications/JOOS.ps. Inspected on 20/05/09, Oct. 1997.

[106] D. A. Watt and M. Thomas. *Programming Language Syntax and Semantics*. Prentice Hall, Hertfordshire, UK., 1991.

[107] I. Wilkie, A. King, M. Clarke, C. Weaver, C. Raistrick, and P. Francis. *UML ASL Reference Guide: ASL Language Level 2.5*. Kennedy Carter, Ltd., revision d edition, 2005.

[108] M. Yang, G. Michaelson, and R. Pooley. Semantics for a UML Action Langauge. In *Proceedings of XII Brazilian Symposium on Programming Languages*, pages 129–142. Brazilian Computer Society, August 2008.