Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 101
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 101
Espoo 2006 HUT-TCS-A101

JUMBALA — AN ACTION LANGUAGE FOR UML STATE MACHINES

Jori Dubrovin

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 101
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 101
Espoo 2006 HUT-TCS-A101

JUMBALA — AN ACTION LANGUAGE FOR UML STATE MACHINES

Jori Dubrovin

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK
Tel. +358-9-451 1
Fax. +358-9-451 3369
E-mail: lab@tcs.tkk.fi
URL: http://www.tcs.tkk.fi

ISBN 951-22-8102-3
ISSN 1457-7615

Multiprint Oy
Helsinki 2006

ABSTRACT: UML 2.0 is a language for modeling complex software systems. A UML model may describe the dynamic aspects of software as well as the static structure. We concentrate on models of reactive systems, such as embedded controllers or telecommunications switches. The behavior of such systems is modeled using UML state machines. Although UML defines the structure of state machines, it leaves open the choice of an action language, which is the language used to specify how the transitions of a state machine affect the configuration of the underlying model.

A UML action language named Jumbala is introduced. The language has been designed as part of a project where the goal is to formally analyze behavioral UML models. Jumbala is based on the Java programming language. It has nearly the same syntax and semantics for statements and expressions as Java. Some new programming constructs have been added to facilitate state machine modeling. Jumbala also supports object-oriented programming with classes and inheritance.

An interpreter that parses and executes Jumbala programs has been developed. The interpreter will be part of a prototype tool set for analyzing the behavior of reactive computer systems modeled in UML.

KEYWORDS: action language, behavioral modeling, interpreter, Java, object-oriented, UML

CONTENTS

List of Figures

1 INTRODUCTION

Software systems are among the most complex systems ever built by humans. Handling that complexity is one of the challenges when aiming to improve the quality of software products. A proposed solution is not to think in terms of the software system itself but in terms of a *model* of the system [18, 28, 30]. A model is a representation that is simpler than the system but still contains the details that we consider essential. The purpose is to raise the level of abstraction and allow thinking in high-level concepts instead of technicalities.

This still leaves open the questions of how to decide what is essential, and how to represent the model. The last question has one answer that has been widely adopted: the Unified Modeling Language (UML) [29]. It is the result of combining the leading development and modeling concepts of the past decades, backed up by an industrial consortium known as the Object Management Group (OMG). The current version of the language is UML 2.0.

UML has a graphical notation that aims to help people think and communicate about the model. The notation is based on different kinds of diagrams that are suitable for expressing a variety of aspects such as customer requirements, test cases, and collaboration of software units. The overall structure of modules and the associations between them are represented hierarchically using package and class diagrams. Besides structural matters, UML can express dynamic aspects with behavioral diagrams such as state machines and activity diagrams.

The problem area that we concentrate on is the behavior of reactive, embedded software systems such as those found in medical devices, telecommunications switches, and railway signaling systems [15]. A system is reactive if it does not have all its input ready when the system is started. Instead, the system receives input from its environment on the fly during operation. The inherent characteristics of a reactive system are parallelism and continuous interaction with the environment. The evolution of the system in time depends not only on the current state of the system but also on the inputs, which may be unpredictable. This phenomenon is known as *nondeterminism*. Another source of parallelism, besides the environment, is concurrent execution within the system. A reactive system often contains several software modules that run under a real-time operating system or that may even be distributed physically. The mutual order of their execution is difficult or impossible to resolve, so it is effectively nondeterministic. A consequence of nondeterminism is that the number of possible executions of the system may be immense [35].

The traditional approaches to analyzing behavior are simulation and testing, which mean running the implementation in a simulated or real environment and examining the results. However, the presence of nondeterminism reduces the capacity of these methods. Errors may go undetected because of the large state space [33], and even if a failing execution is found, it may be difficult to repeat in a nondeterministic environment.

A more ambitious scheme is to use *formal verification* [10]. Formal verification means explicit or implicit mathematical analysis of the possible executions of the system to prove that it satisfies a defined property. The object under verification must have a well-defined set of executions. It is possible to derive such an object from the final implementation of a piece of software by introducing formal semantics for the programming language used. Another possibility is to verify the properties of an abstraction, the model. The latter approach has a number of benefits. The model exists before an implementation is written. An error found in the implementation code can be tens of times more expensive to fix than an error found at the design phase in the model, especially in a large project [7]. Verification of the model may be possible even if the model is incomplete because the missing parts can be replaced by abstract versions. Also, the chance of successfully verifying properties of the model is better because the model is mathematically simpler than the implementation.

UML has elements for modeling reactive behavior, in particular, *state machines* that communicate with signals. A behavioral UML model is suitable for being formally verified. However, there are requirements that the model must meet: it must be represented formally and its set of executions must be well defined. UML as such is a loose specification. The language has to be trimmed and elaborated to create a subset that is still useful for modeling software behavior and that has unambiguous semantics for execution.

The most fundamental building block of behavior in UML is called an *action*. A single action might, for example, destroy an object instance or assign a new value to some variable. Actions in a UML state machine define the effect of firing a transition in the state machine. In state machine diagrams, actions are expressed in textual form written in a suitable language. The twist is that UML does not define a language for writing actions; the choice is left to the user or the manufacturer of a modeling tool. It is legal to describe actions in an existing programming language such as C++, or a natural language such as ancient Greek, or anything the user finds appropriate. This is one of the conscious choices that the designers of UML have made to keep the standard as widely applicable as possible, without enforcing the use of any particular technology.

When the intention is to derive a machine-executable subset of UML, a formal action language is indispensable. The purpose of this work is to design a UML action language and an interpreter for it. The language is named Jumbala. It has been developed in the Laboratory for Theoretical Computer Science at Helsinki University of Technology in the framework of a project where the goal is to develop efficient verification techniques for industrial UML models.

Jumbala has syntax and semantics very close to the Java programming language [12]. Jumbala is considerably simpler than the current version of Java, and it has additional constructs to support state machine models. Java has been chosen as the baseline because the language is well known in the industry and because it is based on the same object-oriented philosophy as UML. Java also has relatively clean and well-defined semantics, which is a good starting point for formal verification.

The use of an action language that resembles a programming language makes it possible to analyze models where the level of abstraction varies. Generally a model describes a system at high level, but we may also be interested in verifying models that have some components close to the implementation level. Assuming that Java is used as the implementation language, an action language like Jumbala makes it more straightforward to obtain such models.

In our framework, actions in state machines are specified using Jumbala statements and expressions. The idea is that the user should feel like writing actions in Java. Therefore we support almost all statements and expressions of Java, including dynamic allocation of objects. We have omitted some features, for example implicit type conversions and exceptions, because they are not required in the framework and they would have involved heavy implementation efforts. A new construct, the `send` statement for sending signals, has been added to support asynchronous communication between objects. Jumbala is a statically typed language with integer and Boolean primitive types, strings, arrays, and user-defined classes and interfaces. The type system has direct support for fundamental object-oriented concepts such as fields, methods, and polymorphism. If a UML model contains non-reactive classes or class hierarchies, their behavior can be implemented as Jumbala methods. Certain advanced features of the Java type system, e.g. inner classes and generic types, have been left out.

The Jumbala interpreter has been implemented in Python and it is integrated as a functional part of a UML simulator and model checker. The interpreter performs syntax and type checking, and produces human-readable error messages. Internally, Jumbala source programs are translated to a simple code language, which is then executed in a virtual machine that is a part of the interpreter. This feature is currently used for simulating executable UML models. To aid verification, the interpreter provides an interface for accessing the data structures representing the Jumbala statements given as input. The interface is being used by a preliminary version of a model checker that can verify UML models using a subset of Jumbala as the action language.

The report is organized as follows. In Chapter 2 we present the parts of UML that are necessary for being able to see where an action language is used and what it does. In Chapter 3 we describe the Jumbala action language, its features, and the design decisions behind them. The model designer who specifies actions uses only a subset of the features of Jumbala, most notably statements and expressions. Other features, such as class declarations, are needed for building the tools that interface the interpreter. Chapter 4 combines the topics of the previous chapters to explain at a general level how the statements and expressions of Jumbala relate to UML model elements.

Our implementation of the Jumbala interpreter and the programmatic interfaces it offers are covered in Chapter 5. Chapter 6 contains a quick survey of other existing UML action languages besides Jumbala. Chapter 7 concludes the report with some final remarks.

2 UML

The Unified Modeling Language (UML) is a visual language for specifying, constructing, and documenting the artifacts of a software-intensive system [29].

UML defines a model as a set of interconnected *model elements*, which are abstractions drawn from the system. What the user normally sees is a UML *diagram*, which is a graphical view to the model and shows a subset of the model elements. Different diagrams provide different perspectives to the model with various levels of accuracy. The diagrams are largely independent and complement each other to form a general view that can be easily understood. Modeling tools are responsible for ensuring that these views consistently represent the same model.

UML is a vast language. Anyone who wishes to write a model in UML must decide which parts of the modeling language are the best for expressing the essential aspects of the particular type of system. UML, by itself, does not give an answer. It gives the specification of a model but it does not tell how to build one.

In this chapter we describe a set of UML constructs that can be used as a basis for modeling reactive, distributed software. The focus is on complex discrete-event systems such as communication networks. It is not our purpose to establish a complete design framework but to allow us to see the context and requirements of an *action language*, which is the language for specifying the low-level dynamics of models. The presented subset of UML is rather minimal but contains the basic elements for modeling reactive behavior. We have omitted some types of model elements that are well suited to the problem domain, not because we consider them unimportant but because they would not have much impact on the role of an action language.

For a more comprehensive view of UML, consult the abundance of literature, e.g. [8, 11, 32].

UML takes an object-oriented view on the partitioning of the system. The static structure and relationships of objects are modeled with classes (Section 2.1). A UML class diagram shows a graphical view of the class structure.

The objects that are capable of initiating execution without an explicit request from another object are called *active objects*. Active objects communicate asynchronously using *signals*. The dynamic behavior of active objects is modeled with UML *state machines*, which are based on Harel Statecharts [14]. State machines are a natural formalism for modeling reactive, event-driven software, where the focus is on handling unpredictable external events rather than on heavy computation. We discuss state machines in Section 2.2.

A state machine specifies the behavior of a single object. When we examine an execution of the system as a whole, we need to consider the states of all objects. A *global configuration* is a collection of all the information that an instance of a model contains at a point of execution. It is the subject of Section 2.3. An object diagram, a special case of the class diagram, shows the relationships and attributes of object instances, giving a snapshot of the global configuration of the system.

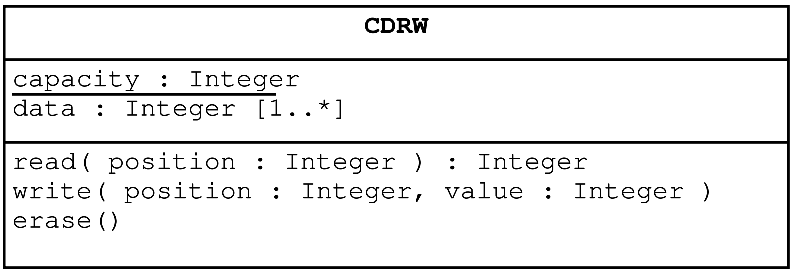

Figure 2.1: The `CDRW` class.

A transition in a state machine may have an associated *activity* that is executed when the transition is fired. Statements written in the action language appear at this level to specify activities. In UML, activities are decomposed into atomic pieces of behavior called *actions*. Activities and actions are covered in Section 2.4.

2.1 OBJECTS AND CLASSES

As UML is a language for object-oriented modeling, our abstraction of the system is based on the concept of an *object*. An object is an identifiable, bounded entity that encapsulates state and behavior. An object may represent an abstract or a concrete thing. A *classifier* describes a set of objects that have features in common. There are several kinds of classifiers in UML, including classes, interfaces, and signals.

2.1.1 Attributes and Operations

The most general kind of a classifier is *class*. It defines a set of objects with two kinds of features: *attributes* and *operations*. For example, a class named `CDRW`, representing a rewritable compact disc, might have the attributes `capacity` and `data`, and operations `read`, `write`, and `erase`. Figure 2.1 shows the class graphically as a rectangle subdivided in three compartments. The topmost compartment contains the name of the class, the middle compartment shows the attributes, and the lowest compartment lists the operations.

Additional information about the features of the class is shown in the figure. The attribute `capacity` has type `Integer`, expressed by the colon and type name after the name of the attribute. The square bracket notation `[1..*]` on the next line denotes *multiplicity*, in this case a range from one to infinity. Multiplicity declares the minimum and maximum number of values assigned to the attribute. If omitted, multiplicity defaults to `[1..1]`. Thus, the attribute `capacity` always has exactly one `Integer` value, and `data` consists of one or more `Integer`s. The chosen representation of `data` stands for a low-level view of the contents of a compact disc. In the end, the data stored on a disc is a sequence of numbers, maybe just ones and zeros.

The attribute `capacity` is a *class attribute*, which means that at run time there exists only one value for `capacity` that is shared by all objects of class

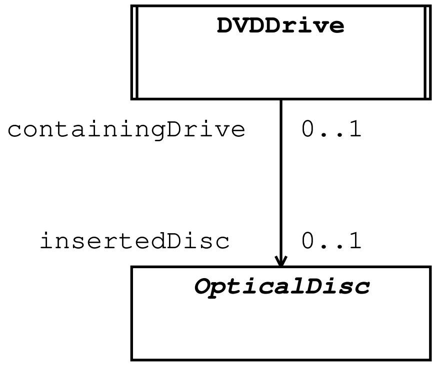

Figure 2.2: Two classes with an association.

CDRW. Class attributes are tagged in the diagram by underlining them. The other attribute, data, is not underlined, so it is an *instance attribute*. Every CDRW object has its own value for data, and these values are generally distinct between objects. Conceptually, objects of a class have slots corresponding to all instance attributes defined by the class, and these slots must be filled with values that match the types and multiplicities of the attributes. An object that has no value associated with the attribute named data cannot be a well-formed CDRW instance. Neither can an object whose data slot is filled with the string "backup", which is not a sequence of Integers.

An operation specifies a service offered by an object. As the CDRW class declares the operation erase, anyone who has access to a CDRW instance at run time can *call* the operation on that instance to invoke a behavior. The declaration of an operation places a requirement that such a behavior exists, but it does not specify the behavior itself. A procedural implementation for the behavior may be given in a programming language. Such an implementation is called a *method*. Operations may involve parameters, expressed in parentheses after the name of the operation (see the operation write in Figure 2.1). Parameters have types, and each call to the operation must be accompanied by values for all parameters. An operation may have one or more return values that the behavior must supply back to the caller at run time. Textually, return values are specified by listing their types after a colon following the parenthesized parameter list. The operation read of the CDRW class returns an Integer. The three operations declared in the example are all *instance operations*. The behavior of an instance operation is associated with a particular object that is determined when the operation is called. It would also be possible to declare *class operations* by underlining them in the diagram. Class operations are called without an instance of the class.

2.1.2 Associations

A picture of a class in isolation does not give a holistic view of a system. A class diagram usually contains several classes, and *associations* between them. Semantically an association means that objects of the classes are somehow connected. If there is no association between two classes, say, DVDDrive and

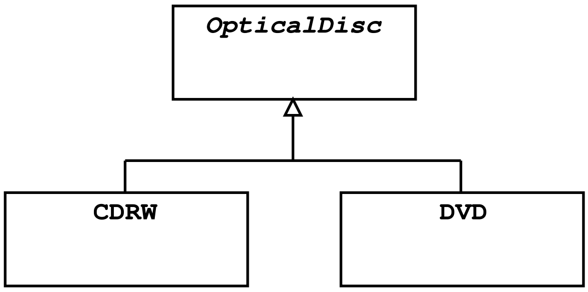

Figure 2.3: A class diagram that demonstrates generalization.

`OpticalDisc`, then there is no way a DVD drive (an instance of `DVDDrive`) can access an optical disc (an instance of `OpticalDisc`) directly, without the intervention of a third class acting as a bridge.

In its simplest form, an association is drawn as a solid line between two classes in a class diagram. By default, an association is bidirectional. An open arrowhead in one end of the line denotes navigability only in that direction. For example, the class `DVDDrive` in Figure 2.2 can access an associated `OpticalDisc`, but not vice versa.

An association has two *association ends*. An end is often labeled by a *role name*, which tells the role played by the class attached to that end. An end also has multiplicity, which indicates how many objects participate in the relationship. If omitted, the multiplicity defaults to 1. As shown in the figure, a `DVDDrive` instance can contain zero or one optical discs, and an `OpticalDisc` instance can be inserted in zero or one DVD drives at a time.

A run-time instance of an association is called a *link*. A link is a connection between objects, which must be instances of the classes corresponding to the ends of the association.

2.1.3 Generalization and Interfaces

Another kind of a relationship between classifiers is *generalization*, which captures the concept of inheritance in object-oriented design. Generalization relates a specific classifier to a general classifier. Each instance of the specific classifier is also an instance of the generic classifier, and features specified for the general classifier are implicitly present in the specific classifier.

A generalization is visualized by a solid line that ends in a hollow triangle at the end of the general classifier. The class `OpticalDisc` in the class diagram of Figure 2.3 is a generalization of `CDRW` and also a generalization of `DVD`. This captures the fact that every rewritable CD is an optical disc, and every DVD is also an optical disc. The name of the class `OpticalDisc` is printed in italics, which means that the class is *abstract*. Such a class cannot be directly instantiated. To create an optical disc, one must be more specific and choose which kind of a disc to create: a CDRW, a DVD, or perhaps some other concrete specialization of `OpticalDisc` not shown in the diagram.

An *interface* is a classifier that is similar in concept to an abstract class.

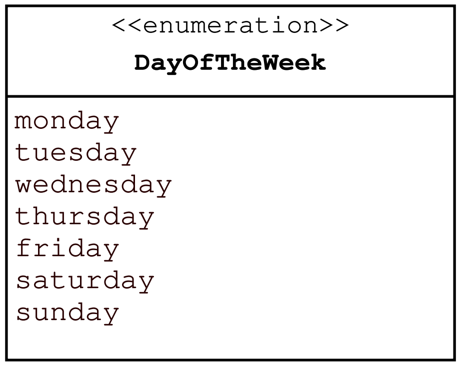

Figure 2.4: A class diagram with an enumeration.

Neither can have direct instances but the difference is that an abstract class can have an implementation for some of its operations. An interface is purely a specification of services provided by an object. A class whose objects fulfill the specification is said to *realize* the interface.

2.1.4 Enumerations

An *enumeration* is a classifier that has a finite, predefined set of instances. The instances, or *enumeration literals*, have no internal structure or data, but they can be compared for equivalence. The graphical representation of an enumeration is similar to a class. The topmost compartment is marked with the keyword «`enumeration`» above the name, and the second compartment contains a list of enumeration literal names. Figure 2.4 shows a class diagram with an enumeration whose literals denote the seven days of the week.

2.1.5 Active Objects

Active objects model concurrent execution. An active object has its own thread of execution. An object that is not active is passive. A passive object executes behavior only in the scope of another object, and only when requested to do so by that object. All actions are ultimately initiated by active objects. An *active class* is a class whose instances are active objects. In a class diagram, double vertical lines on the sides of a class emphasize the fact that it is active. In Figure 2.2, `DVDDrive` is an active class.

Active objects support asynchronous communication using *signals*. Either an active or a passive object can send a signal instance to an active object. Sending a signal is like mailing a postcard. The sender does not have to stand still while the signal or its consequences are being processed, and it is not specified exactly when the recipient will receive the signal or react to it.

We make the assumption that if an active object has instance operations, they can only be called by the object itself. This is because we do not want to allow bypassing the signal mechanism by synchronous operation calls. To perform synchronous communication between active objects, two signals have to be used, one in each direction.

Signals cannot be sent to passive objects because they are not able to react asynchronously. However, a passive object can have operations that may be called by any active or passive object. The execution of the caller is suspended for the duration of the operation. The behavior associated with the operation may call further operations or send signals.

2.2 STATE MACHINES

A *state machine* is a mechanism for specifying behavior using a finite state-transition system. UML defines two kinds of state machines. *Behavioral state machines* are used to specify behavior of model elements such as class instances. We assume that the behavior of all active objects is modeled using behavioral state machines. *Protocol state machines* are used to express the allowed usage protocols of ports and interfaces. A protocol state machine defines the legal sequences of events that may occur in the context of a classifier. For example, an interface could define two operations, `open` and `read`, and a protocol state machine could be used to define that `read` must not be called before `open` has been called. As protocol state machines do not contain actions, they are not discussed further here.

2.2.1 States and Transitions

A state machine is a graph of *states* and *transitions*. At any moment during run time, one or more of the states are *active* in an active object. The set of active states is called the *active state configuration*, and the object is said to be in those states. The states that are not active are *inactive*. The active state configuration is changed by firing transitions. When a transition is fired, the source state becomes inactive and the target state becomes active.

We restrict our treatment to flat state machines, where no state is nested in another state. Exactly one state is active at a time in a flat state machine. UML also defines hierarchical state machines [32] that allow states to contain substates. Several substates may be active at the same time.

Figure 2.5 contains a simple state machine that specifies the behavior of an active class, call it `OverheatProtection`. The state machine contains three states (a rounded rectangle), five transitions (an arrow from source state to target state) and an initial pseudostate (a filled circle). Transitions have text labels of the form *trigger* `[`*guard*`]` `/`*effect*. Any of the three parts may be omitted.

The initial pseudostate indicates the starting conditions of the state machine. When an object of class `OverheatProtection` has been created, its active state configuration consists of state `Operating`.

Transition Triggers

The *trigger* of a transition specifies an *event*. The transition may only fire if the event occurs. In Figure 2.5, the trigger of the transition leaving from state `Operating` is a *signal event*, denoting the reception of the signal `measured`. The signal comes when a new temperature measurement is ready. When that happens, the object switches from state `Operating` to state `Checking`.

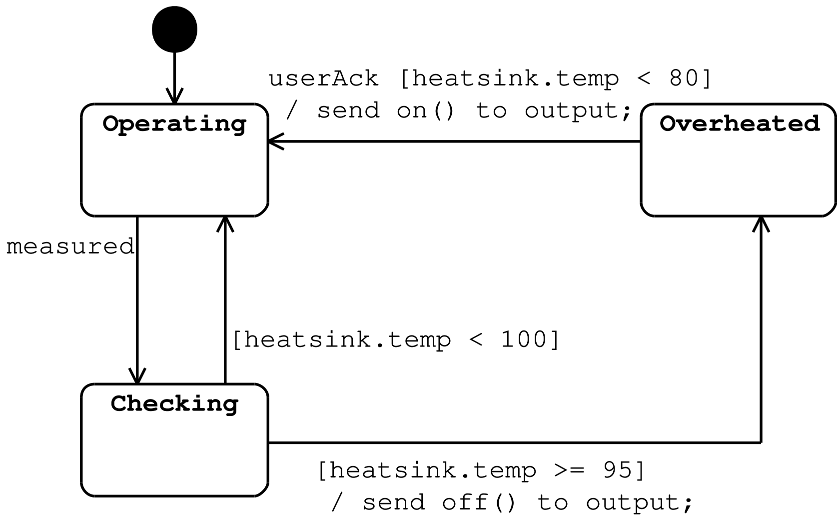

Figure 2.5: A state machine diagram for `OverheatProtection`.

Other kinds of events, which we do not consider, are time events for real-time modeling (e.g. `after (2 s)`), change events that occur when a condition changes (e.g. `when (altitude < 100)`), and call events that represent incoming operation calls.

The transitions leaving state `Checking` have no explicit triggers. Such a transition is called a *completion transition* and it is triggered by an implicit *completion event* that is generated as soon as all internal activity in the source state is finished. In our simple framework there are no states with any internal activities. Therefore a completion event is generated whenever a state becomes active.

If an event occurs that is not the trigger of any transition leaving an active state, the event is implicitly consumed. For instance, if the signal `measured` is received in state `Overheated`, it is simply discarded.

Guards

Three transitions in Figure 2.5 have *guards* enclosed in square brackets. A guard is a Boolean expression acting as a precondition for firing the transition. When the trigger event occurs, the expression is evaluated. If the result is true, the transition may be fired. If the result is false, the transition is not fired and the guard is not evaluated again until a new trigger event occurs. If the guard is omitted, it is the same as having a guard that always evaluates to true. The language for writing guards is not fixed in UML, but we use Boolean expressions of the action language. A guard must not have side effects. In other words, the model is ill-formed if evaluating a guard changes the global configuration.

The transition from `Overheated` to `Operating` has both a a trigger, the signal event `userAck`, and a guard, `heatsink.temp < 80`. The signal arrives when the user presses a button to acknowledge the overheat alarm, but the transition fires only if the heatsink temperature is below 80 degrees. In particular, if the user presses the button while the temperature is still over 80,

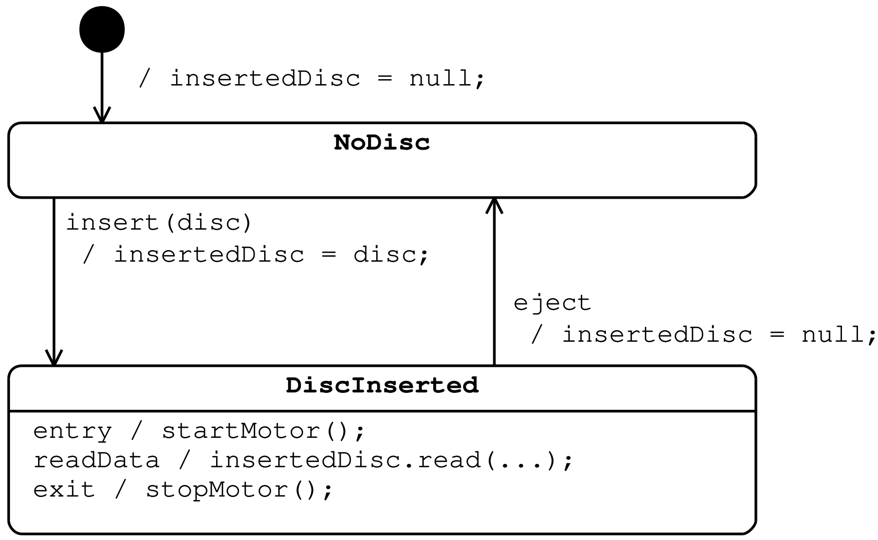

Figure 2.6: A state machine diagram for `DVDDrive`.

it has no effect. Generally, if a guard prevents a transition from being fired, the triggering event is discarded, and a new trigger event has to occur before the transition is considered again.

Both transitions leaving state `Checking` have guards. If the temperature of the heatsink is less than 100 degrees, the active object may return to state `Operating`. If the temperature is 95 degrees or more, the transition to `Overheated` may be fired. Because one of these conditions is always true, the object cannot deadlock in state `Checking`. But what if the temperature is, say, 96 degrees? Either transition may fired, and the choice may be arbitrary. A concrete system might always choose one transition in favor of the other, or it might make the choice depending on the time derivative of the temperature. Thus, there are many possible systems, and the state machine is a model of all of them. When we execute the model, we are faced with nondeterminism, i.e. choices that could go one way or the other.

Effects

Two transitions in Figure 2.5 have an *effect*. The effects are '`send off() to output;`' and '`send on() to output;`'. An effect is an activity that is executed when the transition is fired. In this case, the activities send a signal named `off` or `on` with no parameters to an active object `output`. Activities are expressed in the action language. In our approach, activities are lists of action language statements.

Signal Parameters

The state machine for class `DVDDrive` is shown in Figure 2.6. The trigger of the transition leaving `NoDisc` is `insert(disc)`, which denotes a signal event with a parameter. In UML, signals are classifiers that are allowed to have typed parameters, and we assume that `insert` has one parameter of type `OpticalDisc`. Parameter values can be accessed from transition effects. The purpose of the effect '`insertedDisc = disc;`' is to establish a link between the `DVDDrive` object and the `OpticalDisc` received as a parameter. The link is an instance of the association shown in Figure 2.2.

Entry and Exit Activities and Internal Transitions

State `DiscInserted` in Figure 2.6 has internal structure. The line that begins with `entry` specifies an *entry activity* for the state. The activity calls the operation `startMotor`, which we assume is an instance operation of `DVDDrive`. The entry activity is executed at the moment the state becomes active, after the effect of the incoming transition has been executed. Correspondingly, the *exit activity* '`stopMotor();`' is executed when the state becomes inactive, before executing the effect of the outgoing transition.

The second line, beginning with `readData`, denotes an *internal transition* in state `DiscInserted`, triggered by an ordinary signal event. Internal transitions may have guards and effects like ordinary (*external*) transitions, but they do not affect the active state configuration. Notice that `entry` and `exit` are UML keywords but `readData` is just a signal name. There is a difference between an internal transition and an external transition whose source and target states are the same. An internal transition does not leave the state, and therefore any exit or entry activities are not executed.

UML defines still another kind of an activity associated with a state. A *do activity* of a state is continuous behavior that is executed all the time when the state is active. We omit do activities because our approach only supports execution in discrete steps.

2.2.2 Behavior of Active Objects

Active objects communicate asynchronously by sending signals. The sending object creates an instance of a signal and sends it to another, explicitly identified object. The reception of a signal produces an occurrence of a signal event. Signals may have associated parameters, whose values are passed to the receiving object together with the signal event occurrence.

An active object stores all event occurrences in a temporary storage, the *event pool*. Every active object has its own event pool. The process of handling one event occurrence is called a *run-to-completion* step. First the event occurrence is removed from the event pool. The order in which events are removed is not specified by UML. However, completion events have priority over other events. All transitions in the state machine of the object are then examined to find those that are eligible for firing. A transition is eligible if its source state is currently active, its trigger matches the chosen event occurrence, and its guard condition is true. If no transition is eligible, the event occurrence is simply discarded and the run-to-completion step ends. Otherwise one of the eligible transitions is fired. The choice is nondeterministic.

The firing of a transition consists of removing the source state from the active state configuration, executing sequentially the exit activity of the source state, the effect of the transition, and the entry activity of the target state, and adding the target state to the active state configuration. If the target state has no do activity and no substates, a completion event is placed in the event pool.

If an event occurs during a run-to-completion step, the occurrence has no immediate effect and it is placed in the event pool. No event occurrences are removed from the event pool of the object until the step has finished and the object is in the target state.

UML guarantees that a run-to-completion step of an active object cannot be interrupted by an external event. We make further assumptions by demanding that only a single active object in the system executes a run-to-completion step at a time. Therefore a run-to-completion step is not only uninterruptible in the scope of an active object but it is an atomic step in the entire system. This assumption simplifies the definition of global configuration and reduces the set of possible interleaving executions, making it easier to analyze the behavior of a model.

2.3 GLOBAL CONFIGURATION

At a given moment in time, the system is in a state that, as a whole, is an instance of the model. We call this state the global configuration of the system. With the framework presented so far, we can conclude that a global configuration consists of the following items.

- A list of active and passive objects that exist.
- The values of all class attributes.
- All the links between objects.
- For each active or passive object, the values of its instance attributes.
- For each active object, the active state configuration of its state machine.
- For each active object, the contents of its event pool.

Because of the assumptions of Section 2.2.2, we can say that an execution step from one global configuration to the next is exactly one run-to-completion step of one of the active objects that exist. We do not need to consider the possibility of several active objects executing at the same time. In a global configuration, no activity is being executed. Consequently, there is no need to bundle the global configuration with information such as call stacks or local variables of operation calls.

2.3.1 Object Diagrams

An important part of the global configuration is the knowledge of what objects exist, what their attribute values are and how the objects are linked to each other. This information can be visualized using an object diagram, which shows a snapshot of objects. An object diagram is similar to an ordinary class diagram. Objects are represented as rectangles, like classes. The difference is that the string in the name compartment is underlined. The string is of the form *objectname : classname*, where the object name is optional. The attribute compartment contains values for instance attributes,

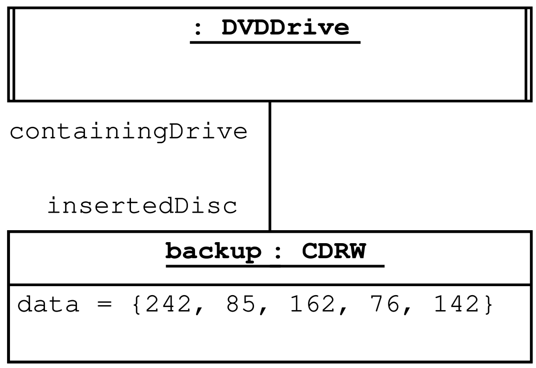

Figure 2.7: An object diagram with two objects.

and the solid lines between objects denote links. A simple example is shown in Figure 2.7. The figure depicts a global configuration where a disc named `backup` is inserted in an unnamed DVD drive.

An object diagram does not capture all the information contained in the global configuration. Nevertheless, an object diagram is useful, for instance, for specifying the initial configuration of the system.

2.4 ACTIONS AND ACTIVITIES

As we saw in Section 2.2, activities are expressed in UML diagrams as strings of text written in an action language. This is a practical perspective to exposing the details of object behavior. The action language typically has constructs that make it easy for humans to read and write. The implementation source-level language may be used as an action language to enable straightforward code generation from the model.

The current version of UML also offers a way to define activities using pure model elements, without resorting to external languages. An activity can be decomposed into a set of *activity nodes* that are connected with *flows*. An activity node can represent a point of flow control, e.g. a branch, a nested activity, or an *action*. Actions are the fundamental building blocks of behavior, similar to the machine instructions of a microprocessor. An action has input and output *pins* that can hold values at run time. When all the input pins hold a value, the action may be executed. The action performs some computation and places the result in the output pins. The output pins of actions are connected to the input pins of other actions by flows to form a network that is capable of complex computation.

The semantics for executing actions and activities is given informally in the UML specification [29]. The semantics is not complete, so there are points where UML does not dictate precisely what an action does. One reason for the existence of action semantics is to allow exchanging models between different tools that use different action languages, without losing the execution semantics. This part of UML also defines what input data is available to an activity, and what is possible or not possible to do by an activity.

Action class	Purpose
CallOperation	Invokes an operation of an object.
SendSignal	Sends a signal instance to an object.
CreateObject	Creates an instance of a classifier.
DestroyObject	Destroys an object.
ReclassifyObject	Changes the classifiers of an object.
ReadLink	Gets the object that a link points to.
ReadStructuralFeature	Reads the value of an attribute.
AddStructuralFeatureValue	Sets the value of an attribute.
CreateLink	Adds a link between objects.
DestroyLink	Removes a link.
RemoveStructuralFeatureValue	Removes a value of an attribute.

Table 2.1: Examples of different kinds of actions.

To make the idea more concrete, Table 2.1 lists some examples of the kinds of actions defined in UML. As can be seen, the list contains no control flow constructs, like "if" or "goto" statements in programming languages. This is because control flow is handled at a higher level in the context of activities using control nodes. Also, one would expect to see actions for primitive numerical operations such as addition, subtraction and comparison of numbers. There are none. By design, UML does not define primitive functions for handling numbers, strings, or Boolean values because it would be impossible to fix the semantics without being inconsistent with some existing programming languages. Nevertheless, there is a mechanism that allows tool manufacturers to provide primitive functions as specialized kind of actions.

UML offers a graphical notation for actions and activities, namely activity diagrams. However, the notation is too low-level to be concise or human-friendly if it is applied to programming language constructs. Unlike textual descriptions of activities, activity diagrams cannot be embedded in state machine diagrams. For these reasons, we do not see the actions and activities defined in UML as a replacement for an action language, but as a complement.

3 THE JUMBALA ACTION LANGUAGE

As we saw in Chapter 2, defining an action language is a crucial prerequisite of modeling behavior rigorously in UML. In this chapter we introduce an action language named Jumbala. It has been developed for the needs of an industrial project called SMUML, whose goal is to formally analyze behavioral aspects of reactive computer systems modeled in UML.

The requirements for Jumbala come from two sources. First, the design of an UML action language must meet the general requirements placed by the UML framework. Second, the tool architecture in the SMUML project dictates further capabilities that the language must have. We discuss these points in Sections 3.1 and 3.2.

Sections 3.3 through 3.9 describe the features of Jumbala and the design rationale behind them. The key design principle has been to make Jumbala resemble the Java programming language [12]. Java is a widely used language with relatively clean and well-defined semantics, and it has features that map well to our requirements. Most of the elements in Jumbala are taken directly from Java. In fact, Jumbala is almost "simplified Java", with only a few features added to better fit the problem domain.

The execution of Jumbala programs is discussed in Section 3.10. We conclude the chapter by listing the most notable differences between Jumbala and Java in Section 3.11.

3.1 REQUIREMENTS FOR AN ACTION LANGUAGE

UML does not define an action language or specify explicitly what features an action language must have. However, the various model elements, such as classes, attributes, and signals, and the UML action semantics imply a set of minimum requirements that a practical action language must meet.

First of all, we require that the language is precise enough to be executed. This rules out solutions like natural languages or pseudocode. The language must be effectively a programming language with precise syntax and semantics.

There are several places in a UML diagram where the action language may turn up. The most apparent are the effects of internal and external transitions and the entry and exit activities of states. Guard conditions of transitions are specified in the action language. We would also like to write the methods of classes using the same language. It is not an absolute requirement that a single language is used for all these purposes but it is an assumption we make to avoid unnecessary complexity. It follows that the language must be capable of expressing Boolean constraints to be usable in guard conditions, and it must be possible to access parameter values and pass back return values in method definitions.

Various kinds of actions, as discussed in Section 2.4, need to be supported. Namely, it must be possible

- to read and write the values of attributes of objects,

- to navigate and modify links between objects,
- to send signal instances, possibly with parameters, to active objects,
- to call the operations of objects,
- to create and destroy objects.

We also require that the language has a basic set of capabilities found in almost every programming language. These include arithmetic operations and comparison of integers, which correspond to UML actions that perform primitive functions, and flow control constructs for branching and looping, corresponding to control nodes in UML activities.

Finally, because one purpose of an action language is to serve as a tool that helps people design and understand complex systems, we want the language to be simple and easy to learn and to support rapid development.

3.2 THE SMUML SETUP

The Jumbala language has been developed in the framework of a project named SMUML (Symbolic Methods for UML Behavioral Diagrams) at the Laboratory for Theoretical Computer Science in Helsinki University of Technology. The aim of the project is to build a prototype tool set for analyzing the behavior of industrial UML models using state-of-the-art symbolic model checking techniques [6, 9, 25, 26]. From the design perspective of the action language, the tool set contains the following three entities.

- **A UML simulator.** The function of the simulator is to execute a UML model, i.e. to produce an instance of the behavior of the model in order to examine the space of possible executions. The simulator must somehow fix nondeterministic choices during execution, for example by asking the user interactively every time a choice has to be made. The goal is that the user learns something from the model or the particular execution, e.g. by discovering a path that reveals a bug in the system.

- **An action language interpreter.** Whenever the simulator takes a step that requires executing a user-defined activity, it invokes the interpreter. Thus, the simulator needs not be aware of the syntax or semantics of the action language. The interface between the simulator and the interpreter must be rich enough to let the simulator know the effects of activities on the instances of UML model elements. In the case of a logical error in one of the activities, the interpreter gives back an error message that the simulator shows to the user.

- **A UML model checker.** The model checker performs formal verification by examining the set of possible executions of the model. The aim is to prove or disprove that the model has a predefined property. The interpreter is not used as an execution engine during verification because that would be too ineffective. Instead, the model checker forms an efficient internal representation for the activities in the model and therefore needs to know the semantics of the action language. The model checker may support only a part of the range of language constructs.

The design of the interpreter is influenced by the observation that even though the simulator depends on the services of the interpreter, the interpreter does not necessarily need the simulator. The interpreter is designed to be a self-contained module whose input is a text string called the *program* and outputs are invocations of callback functions and possibly an error message if the program contains errors. The callback functions and their parameters are declared in the input program. When a callback function is invoked at run time, its implementation, which exists in the simulator, is called with the given parameter values. This allows transferring data from the interpreter to the simulator.

The simulator maps the UML elements defined in the model, such as classes and attributes, to action language constructs that are embedded in the input program given to the interpreter. When the interpreter executes user-defined activities, it manipulates these action language constructs without knowing anything about their origin, the UML model.

This has implications on the action language. It must have enough features to support the various UML model elements, at least the most commonly used ones. In effect, the action language must be an object-oriented programming language. It must support execution in parts because the simulator invokes the interpreter with unforeseeable inputs according to the choices made along the execution. The language must have a callback mechanism to reflect the changes in action language constructs in the interpreter back to the model constructs in the simulator. To support formal verification, it must be possible to derive a formal semantics for at least some parts of the language.

The precise way in which the simulator maps model elements to the action language is part of the design of the simulator and outside the scope of this work. However, the fundamental points in this subject are brought out in Chapter 4.

It turns out that there are two different views of the language. The simulator sees a complete object-oriented language and takes advantage of Jumbala classes and inheritance to map a UML model into a Jumbala program. The person who constructs the model sees a language of statements that send signals or change the values of an object's attributes. The semantics of the language from the modeler's point of view, i.e. how the action language statements eventually relate to the model elements, depends on how the simulator maps UML to the action language.

3.3 DESIGN CHOICES

A Subset of Java

A decision has been made from the beginning to make the Jumbala action language have the look and feel of the Java programming language. A programmer who is familiar with Java should immediately feel comfortable writing activities in UML diagrams. The choice also makes it possible, although not necessary, to have models with low-level parts that closely resemble the implementation of the software system, assuming that Java is used as the implementation language.

Jumbala is roughly a subset of Java. Our baseline is version 5.0 of the Java language, as defined in [12]. Basic flow control constructs, integer arithmetic operations, and handling of types and variables have been incorporated with minimal changes. Other features have been adopted if they were considered crucial from modeling or verification point of view. Features that would have required heavy specification or implementation efforts without much gain have been excluded. New features have been introduced sparingly where it was required to have a capability that is absent in Java.

Static Typing

One of the key features of Java is *type safety*. It means that a program cannot treat a value as a type to which it does not belong. The language is *statically typed*, meaning that variables are annotated with types in the source code and violations of type safety are caught at compile time before execution. These aspects are also included in Jumbala.

From formal verification point of view, static typing is almost a necessity. Performing verification is extremely complicated, especially using symbolic model checking techniques, if the system operates on objects whose types are not known until run time.

Compile-Time and Run-Time Checks

As a consequence of static typing, a Jumbala interpreter has to perform a sophisticated analysis of the entire program before execution. Errors found at this stage are called *compile-time* errors and they include syntax errors, type violations and references to undefined entities. A program that conforms to the language specification (that has no compile-time errors) may be executed. Errors encountered during execution, such as division by zero or following a null reference, are called *run-time* errors. Execution halts immediately after a run-time error occurs. There is no way to catch or recover from a run-time error within a program (cf. exceptions in Java).

Incremental Programs

The simulator-interpreter interface requires that Jumbala programs can be executed in parts. The language has been designed so that when a valid program has been executed, the state of execution is saved. The program can later be incremented by a new program that has access to the entities defined in the original program (we discuss incremental execution in Section 3.10). This is a simplified version of dynamic class loading found in Java. To support incrementality, the notion of a *top-level statement* (Section 3.4.1) is introduced.

Features Left Out

Although Jumbala and Java share a similar structure, they have very distinct sets of design goals. Java is directed at a wide audience and it has been designed to support writing large-scale applications, to run on multiple computer platforms with sufficient performance, and to offer security in a networked environment. Jumbala has a much more restricted range of use. Consequently, Jumbala lacks a number of features that are an integral part of Java.

- **Generic types.** A new feature in Java 5.0 improves type safety but at the same time adds to the complexity of the type system.
- **Annotations.** This is another new feature that offers a unified way to attach metadata to Java source code. Annotations provide information for various Java-related tools but do not contribute to the language semantics.
- **Access modifiers.** The modifiers `public`, `protected` and `private` play an important role in Java by restricting access to implementation details. We assume that the simulator is responsible for prohibiting UML models that access data illegally. In Jumbala, everything is implicitly `public`.
- **Exceptions.** Although UML defines exceptions and exception handlers in the level of activities, it is unclear what the state of an active object should be if an exception is thrown in the middle of firing a transition. We could have allowed throwing exceptions in activities under the constraint that they are all caught and handled inside the context of a transition. The feature was not considered useful enough to compensate for added complexity in the language. We assume that exceptional conditions are modeled using higher-level elements such as signals. Many situations that would throw an exception in Java are considered run-time errors in Jumbala.
- **Reflection.** The ability of a program to observe or modify its own structure was not considered a desirable feature in the context of UML models or verification.
- **Threading.** We assume that there is only a single point of execution control at a time in a program. To achieve the effect of multiple active objects running concurrently, their executions have to be explicitly interleaved. Thus, the simulator has total control over the order of execution.
- **Class libraries.** The Java API contains hundreds of classes for solving more or less generic programming tasks. Jumbala, by default, only supplies the classes that are necessary for the operation of the language itself, such as `Object` and `String`. The architecture makes it possible for the simulator to define a class library and place it at the modeler's disposal.

3.4 PROGRAM STRUCTURE

A Jumbala *program* is a string of text with a structure similar to a Java source file. The program resides in a single string and may not be split into several compilation units. There is no notion of package in Jumbala.

A program consists of identifiers, keywords, literals, separators, operators, comments, and white space. Apart from the slightly different set of keywords and simplifications due to the restricted character set and type system, the program elements are the same as in Java. The amount of white space (indentation, line breaks and space between words) is not significant. Comments can be enclosed between character sequences `/*` and `*/`, or they can be placed at the end of a line after the characters `//`.

Existing conventions for names and program layout in Java can be directly applied to Jumbala.

The structure of a program is hierarchical. The top level contains statements and type declarations. Type declarations are used to add user-defined class, interface, and enum types to the type system. They may contain nested type declarations and declarations of fields, methods, and constructors for the type. The declaration of a method includes the method body, which contains statements that define the behavior of the method.

3.4.1 Top-Level Statements

The statements that appear on the top level, not within type declarations, are the first to be executed. They are meant to be used to set up an initial configuration of objects and to invoke more dedicated behavior that is implemented in classes. Top-level statements are effectively a replacement for the `main` method in Java programs, whose purpose is to act as a point of entry. Simple programs can be written using top-level statements alone, without declaring types.

Top-level statements are particularly useful with incremental execution. An incremental part might consist of just a single statement that relies on definitions in preceding parts. Without top-level statements, every increment would need to have a `main` method or a similar construct of its own, making the language interpreter very cumbersome to use.

3.5 TYPES

The type system in Jumbala is a simplified version of the type system in Java. Jumbala has two primitive types, user-definable class and interface types with inheritance, simple enumerated types (enums) and arrays. Generic types are not supported by Jumbala. The language is strongly and statically typed. Every variable and expression has a type that is determined at compile time. A variable cannot hold a value that is in conflict with the type of the variable.

Types fall into three categories. Primitive types represent Boolean or integer values. Reference types include classes, interfaces, enums, and arrays. The third category contains only the special null type, which is the type of the `null` expression. A programmer rarely needs to think about the null type.

3.5.1 Primitive Types

In Jumbala there are two simple *primitive* types. The `int` type represents signed 32-bit integers, i.e. the numbers -2147483648 to 2147483647. The `boolean` type represents the Boolean values `false` and `true`. The values of primitive types are all disjoint (e.g. 0 is not `false`) and they are not objects.

Floating point types are not supported. It would not be too difficult to implement them in an interpreter, but accurate specification of floating point arithmetic and operations is troublesome. There is also no character type as the language is not intended for low-level manipulation of text. Characters may be simulated using strings of length 1 or integers.

As usual, integer values can be represented in the source code as literals using decimal, hexadecimal or octal notation. The `boolean` values can be represented using the reserved words `false` and `true`.

A variable of a primitive type can only hold a value of that exact type.

Java provides boxed types, such as `Integer` and `Boolean`. They are classes that encapsulate primitive types. In some situations a Java compiler performs implicit conversions between primitive and boxed types. In Jumbala the definition of boxed types is up to the user, and implicit conversions are not supported.

3.5.2 Reference Types

All class, interface, enum, and array types are called *reference* types. Every object has exactly one type that is a concrete class, an enum, or an array type.

The most common kind of a reference type is *class*. A class specifies the state and behavior of a set of objects by declaring *fields* and *methods*. Fields are variables associated with a class or an object, and methods are procedural descriptions of behavior that can be invoked at run time. While a class both specifies objects and describes their implementation, an *interface* is purely an abstract specification. Classes and interfaces are discussed in more detail in Sections 3.9.1 and 3.9.2.

An *enum* denotes a type that has only a finite set of values that are named at compile time. Enum types are covered in Section 3.9.3.

New reference types can be added by writing class, interface, or enum declarations. Most reference types are defined by the user. In the context of the UML simulator, the user defines classifiers in UML class diagrams, and the simulator automatically translates the classifiers to declarations of types in Jumbala.

A variable of a reference type holds either a reference to an object of that exact type or one of its subtypes, or the special value `null`. Note that the value of a variable is never an object but a reference. Creating a variable does not create an object. Two or more variables can hold a reference to the same object. If no variable refers to an object, it is unreachable from the program and, effectively, does not exist.

3.5.3 Subtypes

Reference types form a hierarchy induced by the *subtype* relationship. Every reference type is a subtype of the primordial class `Object`. A class type may *extend* at most one other class type and *implement* any number of interface types. An interface type may extend zero or more interfaces. The class or interface is a subtype of the reference types it extends or implements. An enum type has no subtypes besides itself.

Subtyping stands for *substitutability*. If `S` is a subtype of `T` and an object of type `T` is required somewhere, then an object of type `S` can be supplied instead.

3.5.4 Strings

Objects of the class `String` represent strings of text. Strings exist in the language mainly to make testing and debugging tasks easier. For example, it might be useful to periodically print the value of a variable as a string during the simulation of a model. Strings are not expected to have much significance in actually modeling the behavior of reactive systems.

The `String` type resembles primitive types in that it is a built-in type and its values can be represented using literals in the source code. `String` literals are strings of text enclosed in double quotes. However, `String` is a class type, not a primitive type. For example, a variable of type `String` can hold the value `null`.

As in Java, strings in Jumbala are immutable, i.e. their contents do not change after creation.

3.5.5 Arrays

An *array* represents a sequence of values. An object of an array type consists of an ordered sequence of variables known as *components*. All components have the same type, which is the component type of the array. The component type may be a primitive type or a reference type, even an array type. The number of components is determined when the array object is created and cannot be changed afterwards.

Array types are not explicitly declared. If `T` is any type except the null type, it implies the existence of type `T[]`, which is an array type whose component type is `T`. Multidimensional arrays can be simulated using arrays of arrays.

The components of an array are not named but they can be accessed using the square bracket notation also found in Java. If `arr` is a variable whose value is a reference to an array that has at least five components, then the expression `arr[4]` refers to the fifth component of the array. Array indexing is zero-based.

Array Subtyping

An array type may be a subtype of another array type. If `S` and `T` are reference types and `S` is a subtype of `T`, then also `S[]` is a subtype of `T[]`.

Strictly speaking, this violates the notions of type safety and substitutability. Assume that `arr` is a variable whose type is `T[]`. Now it is legal to execute the assignment `arr = new S[12]` because `arr` can hold a reference to an object of type `S[]`, a subtype of `T[]`. It is also reasonable to allow the expression `arr[4] = new T()`, which assigns the fifth component of `arr` a value that is a reference to an object of type `T`. However, the component happens to be a variable whose type is `S`, and a variable of type `S` cannot hold a reference to an object of type `T`. The anomaly depends on the run-time type of the object referred by `arr` and cannot be detected at compile time. At this point, Java would raise an ArrayStoreException to prevent the illegal assignment. Jumbala does not have exceptions, so the situation causes a run-time error.

The cause of the problem is that an array of `S` is not a kind of array of `T` in every respect even if `S` is a kind of `T`. The problem can only arise when assigning to a component of an array, not when reading the value of a component.

3.6 LIFE CYCLE OF OBJECTS

Objects may be dynamically created during program execution using `new` expressions. The expression returns a reference to a fresh object of a given type. Classes may declare constructors to ensure that a newly created object is in a valid state when the execution of a `new` expression completes.

A program manipulates objects through references, so it is often not possible to determine the scope of existence of an object at compile-time. An object has to exist at least as long as it is reachable by following the references. An object may become unreachable if, for example, a variable that holds a reference to the object is assigned a new value or falls out of scope. An unreachable object cannot affect the execution of a program, so it can be removed. The process of removing unreachable data is called *garbage collection*. We do not define when or how it happens. The language has no mechanism for observing garbage collection via measuring time or the amount of free memory, for example, and there are no destructors or finalization methods for objects like in Java. Therefore garbage collection is purely an implementation concern.

Along similar lines, out-of-memory situations are outside the scope of the language. Specifically, execution of a `new` expression always succeeds. In practice, an implementation might run out of memory at any point during execution. It is always an exceptional situation that most probably terminates the execution. In contrast, Java allows an OutOfMemoryError to occur only at certain points in the program, and the error may be caught and handled within the program.

3.7 EXPRESSIONS

Expressions denote computation of values. Expressions are evaluated at run time to perform primitive operations, such as integer arithmetic, or more demanding operations, such as creating new objects. The requirement of basic arithmetic and comparison operations is fulfilled by expressions.

Evaluation results in a value. For example, the value of the expression `24 * 60` is the integer 1440. A more complex example is the assignment expression `minutes = 24 * 60`, where `24 * 60` appears as a *subexpression*. When evaluated, the assignment expression results in the value 1440, and more importantly, it produces a *side effect*: the value is assigned to the variable `minutes`.

The decomposition of expressions that contain multiple operators, such as `100 + 2 * 2`, is governed by operator precedence rules. The rules are the same as in the Java language and follow mathematical conventions. Therefore, the value of `100 + 2 * 2` is 104, not 204.

Expressions have types at compile-time and they are subject to static type checking. Illegal type combinations, such as `100 + false * 2`, are reported as compile-time errors.

Guard conditions in UML state machines can be written as Jumbala expressions of `boolean` type. UML states that a model is ill-formed if the evaluation of a guard can produce side effects. It is straightforward to use the interpreter to statically check that a guard indeed produces a `boolean` value, but it is not easy to determine statically whether a guard is free from side effects if it invokes methods. However, it is possible to instrument the simulator to perform checks at run time by comparing the global configuration before and after guard evaluation.

3.7.1 Evaluation Order

When expressions are assembled from simpler subexpressions, the evaluation order of subexpressions is fully specified in Jumbala, like in Java. Generally, the order is from left to right.

Evaluation order is a distinct concept from precedence. Consider the expression *x* `+` *y* `*` *z*. Operator precedence rules state that *y* times *z* is added to *x* because multiplication precedes addition. Evaluation ordering rules state that subexpression *x* is to be evaluated first, then *y*, and finally *z*.

The order of evaluation makes a difference when the subexpressions have side effects. Because we have fixed the evaluation order, the semantics of complex expressions is well defined. This prepares the ground for defining formal semantics. Even insane expressions, such as `i += ++i + (i = 7)`, have a well-defined meaning. The example assigns to the variable `i` three times and reads its value twice. A more subtle setting might involve calling several methods in one expression, each method having access to the same object.

3.7.2 Variables

Variables are accessed by their names. A name can be a single identifier or a qualified name consisting of identifiers separated by periods.

A single identifier can refer to a local variable, a method or constructor parameter, or a field in an enclosing type. A qualified name always denotes a field. For example, the expression `system.version` means either the field `version` in the object referred to by the variable `system`, or a class variable named `version` in a class or an interface type `system`, or an enum constant `version` declared in an enum type `system`. The disambiguation process is the same as in Java.

The components of an array are variables that are accessed using the notation *array*`[`*index*`]`.

3.7.3 Arithmetic and Bitwise Operators

The set of arithmetic and bitwise operators is copied directly from Java.

The arithmetic operators are unary plus (`+`) and minus (`-`), multiplication (`*`), division (`/`), remainder (`%`), addition (`+`), and subtraction (`-`). They operate on integer operands using 32-bit modulo arithmetics just like in Java. Bitwise operators on integers, namely bitwise complement (`~`), shift left (`<<`), signed shift right (`>>`), unsigned shift right (`>>>`), AND (`&`), XOR (`^`), and OR (`|`), work as in Java except when using the shift operators with unusual shift distances (see below).

The AND, XOR, and OR operators work as logical operators when their operands are of the `boolean` type. The unary logical complement operator (`!`) inverts the truth value of its `boolean` operand.

The prefix and postfix increment (`++`) and decrement (`--`) operators offer shorthand notation for adding or subtracting 1 from an integer variable. The prefix expressions `++x` and `--x` modify the value of `x` and yield the new value, while the postfix expressions `x++` and `x--` modify `x` and return the original value of `x`.

Shift Operators

The shift operators are binary operators with two integer operands. The left operand is the value to be shifted, and the right operand is the shift distance. The value of `n << s` is `n` shifted left by `s` bit positions. The value of `n >> s` or `n >>> s` is `n` right-shifted `s` bit positions with sign-extension or zero-extension, respectively. In each case, the bits that flow over the 32-bit boundary are discarded.

Shift operators, often combined with bitwise AND and OR operators, can be viewed as a low level mechanism that allows quick but restricted multiplication and division of integers, as well as packing of several pieces of data into the different bits of a single integer. They are valuable for efficient direct handling of hardware registers, and shift operators have direct counterparts in the instruction set of many microprocessors. Perhaps for this reason the Java language allows an efficient implementation by not specifying run-time checks for the amount of shift distance. Instead, when the value to be shifted is a 32-bit `int`, the shift distance is forced in the range 0 to 31 by only taking its 5 lowest-order bits into account.

The Jumbala specification consciously deviates from this behavior by offering a more intuitive approach. If the shift distance evaluates to a value less than zero, regardless of which shift operator is used, execution interrupts in a run-time error. If the shift distance is greater or equal to 32, then, conceptually, all bits of the original value are shifted out. Therefore, assuming that `s` ≥ 32, the value of `n << s` or `n >>> s` is 0. The value of `n >> s` is 0 if `n` is non-negative, and -1 otherwise, because of sign-extension.

String Concatenation

The binary `+` operator is used not only for adding integers but also for concatenating `String`s. The latter functionality is invoked when the compile-time type of the left and right operand are both `String`. This is a more restricting condition than in Java, where only one of the operands needs to be a `String` and the other is implicitly converted by string conversion. Thus, the handy notation `"" + x` sometimes used in Java for converting an arbitrary value `x` to a `String` is useless in Jumbala and will result in a compile-time error.

3.7.4 Comparison Operators

The numerical comparison operators less than (`<`), less or equal (`<=`), greater than (`>`), greater or equal (`>=`), equal (`==`), and not equal (`!=`) compare two integer values with the obvious semantics.

The equal and not equal operators can also be used to compare `boolean` values or references. Two references are considered equal if they are both `null` or refer to the same object. References are not equal if they refer to different objects with the same contents. This coincides with the semantics in Java.

All comparisons result in a `boolean` value.

3.7.5 Conditional Operators

The conditional AND (`&&`) and conditional OR (`||`) operators differ from their logical counterparts (Section 3.7.3) by having short-circuit evaluation rules. If the value of the left-hand operand determines the outcome of the operator, the right-hand operand is not evaluated at all. As a practical example, the expression `(y != 0 && x / y > 100)` never results in division by zero. If `y` equals `0`, the subexpression `x / y` is not evaluated. In contrast, if the conditional AND (`&&`) is replaced by a logical AND (`&`), a run-time error occurs if `y` is zero.

The ternary conditional operator (`? :`) is used to define a conditional branch inside an expression. It can be viewed as a short-hand replacement for an `if` statement (Section 3.8.3). For example, the statement

```
sign = (x >= 0) ?  1 :  -1;
```

is equivalent to the more verbose form

```
if (x >= 0) sign = 1; else sign = -1;
```

3.7.6 Assignments

A simple assignment expression has the form *variable* `=` *expression*. It assigns the value of the *expression* to the *variable*, which must have a compatible type.

A compound assignment expression *variable* *op*`=` *expression* is equivalent to the expression *variable* `=` *variable* *op* *expression*, except that *variable* is evaluated only once. The operator *op* may be any binary arithmetic, bitwise, or logical operator or the string concatenation operator.

Assignment never makes copies of objects. Assigning to a variable of a reference type only makes a new reference to an existing object.

An assignment is an expression, not a statement, so it may appear as a subexpression in a more complex expression. The result of an assignment is the value assigned to the variable. For example, the expression `x = y = 0` assigns the value 0 first to `y`, then to `x`.

3.7.7 Creation of Objects

A `new` expression is used to create new instances of a class type or new arrays. The resulting value is a reference to the new object. In the UML framework, a `new` expression is suitable for creating active as well as passive objects.

The form `new` *classname*`(`*arguments*`)` creates a new class instance. The class must have a constructor (Section 3.9.1) whose parameters match the given arguments. The expression invokes the constructor and returns with a reference to the new object. For example, `new String()` creates an empty string.

Array creation is illustrated below.

```
// Create an array of 3 integers, initially zeros.
int[] arr = new int[3];

// Create an array with the given initial values.
arr = new int[] { 1, 4, 9, 16 };
```

Java has a short-hand form for creating multidimensional arrays in one step, e.g. `arr = new int[2][3]`. We do not support such an implicit form for implementation reasons. In Jumbala, one must write the following to achieve the same effect.

```
// Create an array of array of int.
arr = new int[2][];
// Assign an array of int to each component.
arr[0] = new int[3];
arr[1] = new int[3];
```

3.7.8 Method Invocations

A method invocation, when evaluated, performs a synchronous call to a method declared in a program. It corresponds to CallOperationAction in UML (Section 2.4).

Resolving an invocation is a potentially complicated process that involves, at compile time, selecting a method that best matches the given name and arguments and, at run time, evaluating the arguments and executing the body of the method. Furthermore, if the method is associated with an object (it is an instance method), a reference to a target object is evaluated at run time and the actual implementation to call is chosen based on the class of the object. As a result, method invocations are as powerful and flexible as in Java. Methods are further discussed in Section 3.9.1.

A method invocation has the basic form *methodname*`(`*arguments*`)`. Arguments are given as a comma-separated list. The value of a method invocation expression is the value returned by the method.

A class name may appear before the name of the method, separated by a period, to invoke a specific class method. In the following example, we assume that `abs` is a class method in class `Math`. The method is invoked with the argument value -5.

```
int x = -5;
x = Math.abs(x);
```

Another form of invocation has an expression before the name of the method. The following example first invokes `toString` on the object that `obj` refers to. The return value is a reference to a `String`, whose method `length` is invoked to obtain an integer.

```
int len = obj.toString().length();
```

3.7.9 Type Testing

The `instanceof` operator may be used to test if an object is an instance of a given reference type. The run-time value of the expression *expr* `instanceof` *reftype* is `true` if *expr* is a reference to an object whose type is (a subtype of) *reftype*, and `false` otherwise. A related expression is a *cast*, which has the form `(`*reftype*`)` *expr*. A cast is used to assure the interpreter that *expr* refers to an object of whose type is *reftype* or its subtype. If the presumption proves wrong, execution terminates in a run-time error. Otherwise the value of the cast expression is a reference to the same object. These expressions function in the same way as in Java.

Below is an example of a common pattern, where a variable `obj` is cast to a type `Device` after checking that `obj` indeed refers to an instance of `Device`.

```
if (obj instanceof Device) {
  // It is safe to cast obj to Device.
  Device d = (Device) obj;
  // Use d as a Device reference.
  // ...
}
```

Another context in which casting is indispensable is when extracting elements out of a generic container. Assume that `List` is a container class for holding a list of generic references, i.e. references to `Object`. In the example below, we use a `List` for storing strings.

```
List messages = new List();
// No cast is needed here because
// a string is also an Object.
messages.add("Hello!");

String s;
// A cast is necessary because get returns an Object.
s = (String) messages.get(0);
```

Without casting we would need a separate class for a list of strings. The current version of Java offers a more elegant solution based on generic types. Jumbala does not support generic types due to their complexity.

3.8 STATEMENTS

The implementation of methods and constructors is defined by *statements*. When a method or constructor is invoked at run time, the statements are executed sequentially. Statements may also appear outside type declarations as top-level statements.

Statements in Jumbala have roughly the same role as activities in UML. An activity in a UML state machine (say, the entry activity of a state) might be expressed as a Jumbala statement such as '`powerLed = Led.on;`' or as a sequence of statements. Like activities, statements can be nested in each other. The kinds of statements that may contain substatements are branching statements (`if`, `switch`) and iteration statements (`while`, `do`, `for`). They are used to control the flow of execution, as required in Section 3.1.

The most frequently used statements, including expression statements, `if` statements, and iteration statements, have the same syntax and semantics as in Java. Local type declarations within methods are not allowed in Jumbala. There are no `try`, `throw`, or `synchronized` statements due to the lack of exceptions and threading. A `send` statement has been added to allow a clean syntax for sending signals defined in a UML model.

Statements can be grouped together into a *block* by surrounding them by a pair of curly braces. Blocks are mainly useful as substatements in branching and iteration statements.

3.8.1 Local Variable Declarations

Local variables can be declared wherever statements may occur, i.e. in the body of a method or on the top level. The scope of a local variable begins at its declaration and extends to the end of the innermost block or method body in which the declaration appears. Local variables declared on the top level can be accessed from top-level statements but not from methods or expressions that are enclosed in type declarations.

A local variable declared in an activity in a state machine diagram denotes a temporary storage location that exists only for the duration of that activity.

As a simple example, the following line declares two integer variables `x` and `y`. Variable `x` is initialized to 12.

```
int x = 12, y;
```

If an expression tries to read the value of an uninitialized variable as in `x = y`, a run-time error occurs. A Java compiler prevents the situation by examining the program flow statically and reporting possible problems as compile-time errors. We postpone the check until run time to make the implementation simpler.

3.8.2 Expression Statements

An expression statement evaluates an expression for its side effects. Examples include '`x = 4;`', '`x++;`', and '`list.append(x);`'. The terminating semicolon is part of the statement. The kinds of expressions whose purpose is to produce a value without a side effect cannot be used in an expression statement. For example, 'width() * height();' or 'this;' are not valid statements.

3.8.3 If Statements

An if statement makes a conditional choice based on a boolean test expression, precisely as in Java. In the example below, the assignment sign = +1 is evaluated if variable x is greater than or equal to zero. The block enclosed in braces is executed if x is less than zero.

```
if (x >= 0)
  sign = +1;
else {
  x = -x;
  sign = -1;
}
```

The else part of an if statement may be omitted.

3.8.4 Iteration Statements

The iteration statements include while, do, and for statements. They have the same functionality as in Java, with one omission: Jumbala does not support the enhanced for statements introduced in Java 5.0.

A while statement repeatedly executes a contained substatement, the *body*. The body is often a block of statements, sometimes just a single statement. Before each repetition, including the first one, a boolean condition expression is tested. The condition appears after the keyword while. If the result of the expression is true, the body is executed and the condition is tested again. If the result is false, the body is not executed and the while statement completes.

A do statement is similar to a while statement, but the condition is tested *after* each iteration. Syntactically, the condition is placed at the end of the statement. The body of a do statement is executed at least once.

A third variation is the for statement. It differs from while in that it has an initialization part, a condition, and an update part in its header. The initialization part is executed once before the iteration. It may declare local variables or it may be a comma-separated list of expressions with side effects. The condition is tested before each iteration. The update part, a list of expressions, is executed at the end of each iteration.

Break and Continue

There are two special statements for eliciting abnormal flow of control. A break statement breaks out of an iteration by immediately transferring control to the next statement following the iteration statement. A continue statement prevents further execution of the current iteration, transferring control to the beginning of the next iteration. A break or continue statement may only appear in the body of an iteration statement. A break is also allowed in a switch statement (Section 3.8.5).

The Java language allows a `break` or `continue` to have a labeled iteration statement as a target, making it possible to cut off several nested iterations at once. This cannot be done in Jumbala. A `break` or `continue` statement only affects the innermost iteration statement. Consequently, there are no labeled statements in Jumbala. A programmer who wants to break out of several levels of iteration has to make the modest effort of introducing a flag variable to simulate the effect. Apart from this restriction, the behavior of `break` and `continue` statements is identical in Jumbala and Java.

3.8.5 Switch Statements

A `switch` statement represents a choice with multiple, predetermined possibilities. It consists of the keyword `switch` followed by a parenthesized *switch expression* and a *switch body*. The switch body contains *case labels* (of the form `case` *expression*`:`), at most one *default label* (of the form `default:`), and statements following the labels, all enclosed in a pair of curly braces.

At run time, the switch expression is evaluated and its value is compared to each case expression in turn. Case expressions are the expressions contained in case labels. If a match is found, the statements following the case label are executed. If no match is found, the statements following the default label are executed. Below is an example.

```
switch (time) {
case 12:
  haveLunch();
  break;
case meeting.time:
  attend(meeting);
  break;
default:
  haveCoffee();
}
```

If `time` is equal to 12, the method `haveLunch` is invoked. Otherwise if `time` is equal to the field `meeting.time`, the method `attend` is invoked with `meeting` as the argument. If neither condition holds, `haveCoffee` is invoked. The `break` statements before case and default labels prevent execution from falling through past the label to the following statements.

Java restricts case expressions to have distinct constant values. This makes it possible for a Java compiler to implement the selection procedure as an efficient table lookup. Because Jumbala lacks the notion of compile-time constant expressions, this rule has been relaxed so that any expression of appropriate type is accepted as a case expression. As a result, the execution semantics of `switch` statements is slightly more general than in Java. Many case expressions may evaluate to the same value. A case expression may have side effects, and it may evaluate to `null`. A `switch` in Jumbala resembles a chain of `if` statements that compare the previously evaluated switch expression to each case expression in turn. When the first match is found, no further case expressions are evaluated.

In the example above, if the value of `time` is 12 and `meeting.time` also happens to be 12, only the first case expression is examined. The line '`attend(meeting);`' will not be executed. To avoid confusion, the programmer is encouraged only to write case expressions that have constant, distinct, non-null values and no side effects. Under these assumptions, the semantics in Java and Jumbala are equivalent.

3.8.6 Send Statements

The `send` statement is a construct that does not exist in Java at all. Its purpose is to model the transmission of a signal instance within an activity. Because the transmission of signals has a vital role even in the simplest state machine models, it is given a specialized syntax that is easily recognized. A `send` statement has the form

`send` *name*(*arguments*) `to` *object*;

The *name* of a signal is simply an identifier. The signal may have zero or more arguments whose values are specified as a comma-separated list. The expression *object* evaluates to a reference to the object that receives the signal.

Internally, a `send` statement is equivalent to the statement

(*object*)`.$$signal_`*name*(*arguments*)`;`

i.e. the invocation of a method whose name is the concatenation of the string '`$$signal_`' and the name of the signal. (Dollar signs are valid characters in identifiers.) This form also reveals the semantics of a `send` statement. It is verified at compile-time that the expression *object* evaluates to a reference to an object that has a method with the aforementioned name. Therefore the set of possible signals cannot be changed dynamically. The number and types of parameters for each signal are also fixed at compile-time. At run time, the value of each argument is evaluated to yield either a value of a primitive type, a reference to an object, or `null`.

A simulator that generates the Jumbala source code must ensure that the implementation of an active class has the appropriate methods to support sending signals to the object. The body of a signal method places an instance of the signal in the event pool of the object and performs other bookkeeping activities. The details depend on the implementation of the simulator.

3.8.7 Assertions

An assertion is a statement of the form `assert` *expression*`;`. At run time the *expression* is evaluated, and it is a run-time error if the result is `false`. This is a simplified version of assertions in Java, where an error message can be attached to an `assert` statement, and assertions may be disabled dynamically.

Assertions can be used in a model to explicitly define configurations that must not be reachable during execution. A verification tool may be able to automatically prove the validity of an assertion.

```
class CDRW extends OpticalDisc {
  static int capacity = 650;
  IntList data;
  CDRW() {
    // Initialize data with an IntList of length 1.
    data = new IntList(1);
  }
  int read(int position) {
    return data.get(position);
  }
  void write(int position, int value) {
    // ...
  }
  void erase() {
    // ...
  }
}
```

Figure 3.1: Declaration of a class in Jumbala.

3.9 TYPE DECLARATIONS

New reference types are introduced by adding a type declaration to the program. Class, interface, and enum types all have their own kinds of declarations. Array types are not declared since they exist implicitly.

All information about a type is localized in the declaration of the type and its supertypes. It is not possible to add later, for example, new methods to a class or new enum constants to an enum type.

A type declaration may be placed on the top level in a program to declare a top-level type, or inside a class or interface declaration to declare a member type. Type declarations may not be placed inside methods or expressions, so there are no local or anonymous classes like in Java.

3.9.1 Class Declarations

A class is a specification for objects that have similar properties. The declaration of a class defines the name, supertypes, members, and constructors of the class.

Classes in Jumbala are much like in UML. One use for Jumbala classes is to implement a passive UML class in a straightforward way. An example is given in Figure 3.1. The declared Jumbala class `CDRW` shows a sketch of a possible implementation of the UML class with the same name, shown as a class diagram in Figure 2.1.

Supertypes of a Class

The first line of the class declaration in Figure 3.1 says that `CDRW` extends (is a direct subclass of) class `OpticalDisc`, reflecting the generalization relationship between the corresponding UML classes (Section 2.1.3). If the `extends` part is omitted in a declaration, then class `Object` is implicitly extended.

An aspect of the language is that a class must be completely declared before it is extended. It is an error if the declaration of `CDRW` appears textually before or inside the declaration of `OpticalDisc`. The restriction also applies if a class implements an interface (using the syntax `class` *classname* `implements` *interfacename*) or if an interface extends another interface. The supertype must be declared before the subtype. In all other situations it is permitted to refer to types, fields, or methods before they are declared, as long as they are declared at some point in the program.

Java allows the declarations of a subtype and a supertype to appear in any order. The restriction is included in Jumbala to simplify implementation.

If a class is declared using the keyword `final`, it may not be extended. For example, the class `String` is final to prevent declaring a subclass that breaks the assumption that strings are immutable.

Members of a Class

The members of a class are types, fields, and methods. Members are declared in the body of the class declaration using the Java syntax. Members may also be *inherited* from a supertype of the class. The rules for inheritance are the same as in Java.

A member type can be a class, interface, or enum type. A class and its member type have no special relationship other than the former acting as a namespace for the latter, affecting the syntax to access the member type. In Java terms, all member types are implicitly `static`.

There is no way to restrict access to a member. Java has three access modifiers, "public", "protected", and "private", which do not add to the semantics of the language but are used to hide information from a client using the class. Access modifiers cannot be used in Jumbala as we do not consider it important to implement information hiding in the action language.

Field Declarations

A field can be either a *class variable* or an *instance variable*. They correspond to class and instance attributes in UML. A class variable is global in the sense that there exists only one copy of the variable at run time. For an instance variable, every object of the class has its own copy.

The second and third lines of Figure 3.1 declare two fields, `capacity` and `data`. The type of `capacity` is `int` and it is a class variable, hence the modifier `static`. The declaration of `capacity` ends with an *initializer*, which sets the variable to the value 650. Field initializers are optional. The field `data` is an instance variable and its type is `IntList`. We assume that `IntList` is a class representing a list of integers, declared somewhere in the program. A list is one way of implementing an UML attribute that has a variable multiplicity, in this case [1..*].

It is also possible to declare a field using the modifier `final`. Final fields are used to denote constant values. A final field must have an initializer, and its value may not be changed after initialization. Java allows final fields without initializers provided that they are assigned exactly once in every constructor. To avoid complexity, we require that all declarations of final variables contain an initializer.

Method Declarations

A *method* specifies behavior that can be invoked at run time. A method has a fixed number of parameters and at most one return value. The example class declares three methods. The declaration of method `read` is duplicated below.

```
int read(int position) {
  return data.get(position);
}
```

The method takes one parameter, namely `position`, of type `int` and returns a value of type `int`. The body of the method contains just one statement, a `return` statement, which is used to return a value from the method. The value is obtained by invoking a method named `get` of the object referred to by the field `data`. We assume that `get` is a member of the class `IntList` and returns the value of an element of the list.

If a method is declared to return a value, all execution paths should end in an appropriate `return` statement. Consider an alternative implementation of `read`:

```
int read(int position) {
  if (position < data.length())
    return data.get(position);
}
```

The new implementation attempts to defend against illegal access by checking the validity of the parameter `position`. However, if the check fails, execution falls off the end of the method body without reaching a `return` statement. Such an execution causes a run-time error in Jumbala. The situation cannot occur in Java because a Java compiler is required to analyze the flow of execution and reject methods that may lack a `return` statement. Because of its implementation overhead, compile-time flow analysis is not part of Jumbala.

The problem does not turn up with `void` methods, such as `erase` in Figure 3.1, which do not return a value.

All the methods in the figure are *instance methods*. An instance method is always invoked for a specific object. The method body has access to a reference to the object using the keyword `this` and to instance variables of the object, such as `data` in the example. Analogously to fields, a *class method* is not associated with a specific object at run time. The `this` reference or any instance variables may not be accessed in the body of a class method. A class method is declared with the modifier `static`.

Constructor Declarations

A class has one or more *constructors*, which are used to initialize objects of the class. When an object is created, one of the constructors is invoked. The object cannot be accessed from outside until the execution of the constructor has finished and the object is in a well-defined state. The example of

Figure 3.1 contains one constructor declaration:

```
CDRW() {
  // Initialize data with an IntList of length 1.
  data = new IntList(1);
}
```

Unlike a method, a constructor has no return type and it has the same name as the class. The sole constructor of `CDRW` takes no parameters. When an instance of the class is created, using the expression `new CDRW()`, the body of the constructor is executed. In the example, a reference to a new `IntList` object is assigned to the field `data`. Thus, when the `new` expression returns a reference to a `CDRW` object, the field `data` already contains a list of 1 integer, in accordance with the multiplicity of the attribute.

A class may have several constructors, which are distinguished by the number and types of parameters. In the beginning of a constructor, any superclass constructors are properly invoked explicitly or implicitly in the same way as in Java.

Abstract Classes and Polymorphism

An *abstract class* is a class that cannot be instantiated. It may have member types, fields, methods, and constructors like any class, and in addition, it may have *abstract methods*, i.e. methods without an implementation. Below is an example.

```
abstract class OpticalDisc {
  abstract int read(int position);
}
```

Notice that the body of the abstract method `read` is replaced by a single semicolon.

It is a compile-time error to try to create an instance of the class with the expression `new OpticalDisc()`. The point of an abstract class is that it may have a concrete (non-abstract) subclass that can be instantiated and that has implementations for all methods.

A concrete class that has an abstract superclass must *override* all abstract methods declared in the superclass. A method is overridden in a subclass by declaring another method with the same name and the same number and types of parameters. For example, the class `CDRW` in Figure 3.1 correctly overrides the abstract method `read` in `OpticalDisc`.

A variable of type `OpticalDisc` may be used to invoke the method `read`. The implementation to call is chosen at run time according to the class of the object that the variable refers to, as in the following example. The effect is known as *polymorphism*.

```
OpticalDisc d = new CDRW();
value = d.read(0);  // Invokes the method read in CDRW.
```

A method may be overridden even if it is not abstract. Overriding and polymorphism automatically applies to all instance methods, but not class methods. A method may be declared `final`, in which case it cannot be overridden. The semantics comes from Java.

An abstract class in Jumbala is like an abstract class in UML, and an abstract method is like an operation without implementation. Notice the incoherence in terminology: in UML, a method cannot be abstract because it is an implementation of an operation. In Java or Jumbala, the word method refers to both a specification and its implementation. UML does not define how an operation call resolves to a method. The resolution rules of Java are just one possibility.

Native Methods

An interpreter is not very useful if it just takes a program, computes a while and then halts, perhaps producing a run-time error. What is needed is a way to produce output from the program. Thus, the language has a callback mechanism, as outlined in Section 3.2, for supplying values back to the module that invoked the interpreter.

As in Java, the callback system is implemented using *native methods*. Any non-abstract method of a class may be declared native. Below is a declaration of a native method that prints an integer to the screen.

```
class PrintWriter {
  static native void print(int n);
}
```

Again, the method body is replaced by a semicolon. The method is not implemented in Jumbala but in a platform-specific language. When an invocation of `PrintWriter.print` in a Jumbala program is executed, the interpreter calls the native implementation. When the implementation returns, the interpreter resumes the execution of the program. The details depend on the implementation of the interpreter.

Native methods may also have a return value, so they can be used for input as well as output.

In the UML context the simulator invokes the interpreter to step into a new global configuration. After that the simulator may need to update its own data structures to reflect the new configuration. The simulator does this by invoking the appropriate native methods for all relevant data elements, retrieving the information piece by piece.

3.9.2 Interface Declarations

An interface is like an abstract class that has no implementation. All methods of an interface are abstract, and all fields of an interface are final class variables. An interface may not have constructors. Therefore an interface is merely a specification of a set of methods, possibly completed with related constant values. A concrete class that implements the interface must have non-abstract methods whose names, parameter types and return types match those of the abstract methods in the interface. Figure 3.2 shows a simple example.

Interfaces add to the power of the language by enabling a limited form of *multiple inheritance*. While a class may only extend one superclass, it may implement any number of interfaces. Interfaces may also extend other interfaces. Thus, the class hierarchy forms a tree rooted at `Object`, and classes and interfaces together form a directed acyclic graph.

```
interface Shape {
  int getArea();  // The method is implicitly abstract.
}

class Rectangle implements Shape {
  int getArea() {
    return width * height;
  }
  int width, height;
  // ...
}
```

Figure 3.2: Declaration of an interface and a class that implements it.

3.9.3 Enum Declarations

An enumerated type or enum has a finite set of values that is fixed at compile time. The values are called enum constants and they are listed in the declaration of the enum type. Below is a common example, the declaration of a type whose enum constants are the seven days of the week.

```
enum DayOfTheWeek {
  monday, tuesday, wednesday, thursday, friday,
  saturday, sunday
}
```

For simplicity, enums in Jumbala are less powerful than in Java. Enum constants are not allowed to have fields or methods. In fact, the only properties of an enum constant are its name and identity. Even as such, enums are expected to be useful in many cases, and they are much cleaner and more type safe than using fixed integer values to denote enumerated constants.

A variable whose type is an enum may hold a reference to an enum constant of that type or the value `null`. Values may be compared for identity using the operators `==` and `!=`. Below is a declaration of an enum variable, which is initialized to a reference to one of the enum constants.

```
DayOfTheWeek today = DayOfTheWeek.thursday;
```

An enum type may not explicitly extend classes or other enums or implement interfaces. All enum types implicitly extend a common superclass `Enum`.

3.10 EXECUTION OF PROGRAMS

A Jumbala program that contains no compile-time errors can be executed. Execution begins with the creation of all class variables declared in the program. After that, the initializers of class variables are evaluated and the results are assigned to the variables. Then the top-level statements are executed.

During execution, new variables and objects may be created, and methods may be invoked. The run-time environment must keep track of the values of variables, plus all the objects that are being referred to by variables of a reference type. For each method or constructor invocation, a stack frame must be created that contains the program counter and values of local variables for the invocation.

Unless a run-time error occurs, the execution of a program finishes when the last top-level statement has been executed. At this point all methods and constructors have returned and there are no stack frames. The variables that still exist are the class variables and the top-level local variables. The values of these variables are contained in what we call the *Jumbala configuration*. If any of the values are references to objects, those objects and their instance variables and any objects reachable from them are also included in the Jumbala configuration.

The Jumbala configuration contains exactly the information that is passed on when an incremental program is executed. When a new increment begins executing, the values of the class variables and top-level local variables that were declared in the original program are determined by the Jumbala configuration. The class variables declared in the new program are then created and initialized and the top-level statements executed. This leads to a new Jumbala configuration that contains values for all the class and top-level local variables declared so far.

The process may be repeated indefinitely. New incremental programs can be executed to make changes to the Jumbala configuration or to extract information from it using native methods.

3.11 DIFFERENCES BETWEEN JUMBALA AND JAVA

Jumbala is close to Java in many respects. The general handling of types, classes, and variables, the iteration and branching constructs, and integer arithmetic and other kinds of expressions are almost identical. However, there are points in Jumbala where the intuition of a Java programmer may be misleading. In the following we highlight some potential sources of confusion.

Excluded Features

The list of features is more modest in Jumbala than in Java. Most notably, primitive types are limited to `int` and `boolean`, there are no exceptions and there are no access modifiers. Consequently, Java keywords such as `float`, `throw`, and `public` have no special meaning. The interpreter treats them as ordinary identifiers, which may be confusing to an unaware programmer.

Java has a number of short-hand notations, such as one-step allocation of multidimensional arrays, that are not present in Jumbala. In many places a modern Java compiler performs implicit conversions between types, for example, from an `int` to a `String` or from an instance of `Integer` to an `int`. In Jumbala one must write all conversions explicitly using method or constructor invocations.

Added Features

Jumbala introduces two significant new features. The send statement (Section 3.8.6) is used to model the transmission of UML signals. A side effect is that `send` and `to` are keywords and may not be used as identifiers.

Another new construct is top-level statements (Section 3.4.1), which allow program code to be placed outside any class declarations. Variables may also be declared on the top level but they are only visible to other top-level statements. Thus, there are still no global variables and no global functions. Class variables and class methods may be used instead.

Less Compile-Time Checks

A Java compiler is required to perform a sophisticated analysis on the program to determine, for example, which expressions are compile-time constants, which statements are definitely unreachable, and which local variables might go around uninitialized. The standard has been relaxed in Jumbala to allow a simpler implementation.

As a consequence, certain problems that are caught at compile time in Java are not checked until run time in Jumbala. Reading an uninitialized local variable or forgetting a `return` statement cause run-time errors in Jumbala. The initializers of fields, i.e. class variables and instance variables, are evaluated in the order in which they appear in the program. No checks are made, not even at run time, to prevent an initializer from accessing a field that has not yet been initialized. This does not lead to a run-time error because uninitialized fields, unlike local variables, have a default value of 0, `false`, or `null` depending on the type.

The `switch` statement (Section 3.8.5) in Jumbala is more general than in Java. The case expressions are not checked to be constants but they may be any expressions with a proper type. Some caution is needed from the programmer to avoid problems following from too complicated case expressions.

4 JUMBALA IN THE SMUML FRAMEWORK

In Chapter 2 we outlined the execution semantics for a restricted subset of UML, assuming that the behavior of all active objects is specified using state machines. Chapter 3 described Jumbala as an object-oriented programming language whose semantics does not depend on a UML model. The purpose of this chapter is to connect the execution of Jumbala to the execution of UML.

The presented ideas are based on the implementation of the UML simulator in the SMUML project. As discussed in Section 3.2, the simulator maps a UML model to a Jumbala program to enable accessing model elements from state machine activities. The semantics of Jumbala actions are defined through that mapping. Because all details of the mapping have not yet been fixed in the simulator, we omit the more subtle parts of the semantics.

In Section 4.1 below we tell what Jumbala expressions in different parts of the model can do with respect to the UML model. In Section 4.2 we discuss the mapping from UML to Jumbala and illustrate how the simulator uses the Jumbala interpreter to execute models. Section 4.3 describes how the various UML model elements appear in Jumbala.

4.1 THE CONTEXTS OF ACTIONS

We identify three different contexts in a model where action language statements and expressions may appear. First, the *activities* in state machines are lists of Jumbala statements. These include the effects of transitions and the entry and exit activities of states. Second, the *guards* of transitions are written in the action language. Guards are not statements but Jumbala expressions because they must evaluate to `true` or `false`. It is the user's responsibility to make sure that guards have no side effects. Third, *methods*, i.e. implementations of operations, are specified as lists of Jumbala statements.

In accordance with the basic requirements for an action language, the following bullets describe what Jumbala actions can do and what effects they can have on the instances of UML elements.

- Guards, methods, and activities can perform basic computations, such as integer arithmetic.
- Guards, methods, and activities can call operations.
- Methods and activities can navigate, add, and remove links between objects. Guards can only navigate links.
- Methods and activities can read and write attribute values of objects reachable by navigating links. Guards can read attributes but must not change them.
- Methods and activities can send signals.
- Methods and activities can create new active and passive objects.

Naturally, there are things that cannot be done by executing actions. In the following we list some nontrivial restrictions that are placed on Jumbala actions either by the SMUML framework or by UML itself.

- The active state configuration of any state machine cannot be examined or modified.
- The event pool of any active object cannot be examined and event occurrences cannot be removed. Event occurrences can be added by sending signals.
- Objects can be accessed only if they are reachable by following links and attributes.
- It is not possible to multicast or broadcast signals without individually naming each target object.

4.2 THE MAPPING FROM UML TO JUMBALA

The simulator generates a Jumbala program based on the UML model. Every UML class is mapped to a Jumbala class. Attributes and associations are mapped to fields, and operations are mapped to methods. At run-time, the simulator keeps track of the correspondence between UML objects and Jumbala objects. Attribute values and links in UML are then equivalent to the values of fields in Jumbala.

Active classes require special attention because the action language expressions in their state machines must be included in the program. The Jumbala implementation of an active class has a method for every activity defined in the state machine of the class. The body of the method is just a copy of the activity text in the state machine diagram. A UML editor treats the activity as an uninterpreted string with no meaning. When the string becomes part of a Jumbala program, it gets a meaning with respect to the other entities defined in the program, and it can be executed by the interpreter. When those other entities are mapped back to UML objects and attributes by the simulator, the meaning of an activity is defined in the UML model.

Like activities, guards of transitions are also mapped to methods of active classes. The return type of such a method is `boolean` and the method body consists of a single `return` statement that contains the text of the guard as an expression. A guard can be an arbitrary `boolean` expression that has a meaning in the context of the active class. The user must make sure that guards are free from side effects.

The special methods representing guards and activities must not be invoked from Jumbala code written by the user.

4.2.1 Execution of UML and Jumbala

In Section 2.3 we defined the global configuration in UML as the collection of all the state information from the objects that exist in a system. In one execution step between global configurations, one state machine makes a run-to-completion step. The simulator keeps track of the global configuration and makes the choices regarding the scheduling of active objects.

In the beginning of a step, the simulator picks, possibly nondeterministically, one active object whose event pool is not empty. One event occurrence is chosen and removed from the event pool. The simulator examines the event and the active state configuration to find a transition that can be fired. A transition can only be eligible for firing if its guard condition is true. At this point, the simulator evaluates guards using the interpreter. If one or more eligible transitions are found, the simulator fires one of them. If there are exit activities, transition effects, or entry activities associated with the transition, the simulator executes them sequentially by invoking the interpreter. The run-to-completion step does not end until execution of all the activities has finished.

Guards and activities can access parts of the global configuration, namely objects, links, and attribute values. This information must be made known to the interpreter at run time. The Jumbala program that the simulator generates from the model defines a special array that contains references to all active objects. The array must not be accessed from the code written by the user. When the program is executed, the array is initialized according to the initial conditions specified in the model. The array is implemented as a class variable (a `static` field), so it is part of the Jumbala configuration (Section 3.10) that remains after the program has been executed. The array contains references to active objects, which contain fields that may in turn be references to other objects. The effect is that the Jumbala configuration contains all the objects that exist in the global configuration, and their attribute values and the links between objects.

The simulator uses incremental programs to execute run-to-completion steps. An incremental program can refer to the class definitions of the original program, and it may read and modify the Jumbala configuration. The simulator generates and executes three kinds of incremental programs.

- **Evaluating guards.** To evaluate a guard, the simulator executes a program that picks an active object from the array and invokes the instance method representing the guard. The simulator obtains the resulting `boolean` value by a native method. This process must not change the Jumbala configuration, or the model is ill-formed.

- **Executing activities.** Using the results of guard evaluation, the simulator may choose to fire a transition that has associated activities. To execute an activity in an active object, the simulator invokes the corresponding method for the object. This time, the Jumbala configuration may change because activities are allowed to modify objects. The constructors of active objects are instrumented so that if new active objects are created, they are automatically added to the array.

- **Updating the global configuration.** When all the activities of a run-to-completion step have been executed, a Jumbala configuration has been reached that reflects the new global configuration. The simulator is not yet aware of all the changes, so it makes a program to read all attributes and links into its own data structures.

4.3 MANAGING MODEL ELEMENTS WITH JUMBALA

A UML model contains model elements such as associations and operations, whereas a Jumbala program contains elements from a different realm, e.g. fields and methods. The following sections describe how the various kinds of model elements appear in Jumbala and how they must be handled in guards, methods, and activities to achieve a desired effect.

4.3.1 Associations

To obtain a reference to the object executing a guard, activity, or method, the expression `this` may be used. This is not allowed in a method that implements a class operation.

Links are run-time instances of associations. Because UML associations are mapped to fields in Jumbala, links are handled by field access expressions. We only consider associations whose ends have either multiplicity `1` or `[0..1]`.

An expression of the form *obj*`.`*rolename* navigates a link starting from *obj*, which must be an expression evaluating to an object reference. The *obj* part may be omitted if it is a self-reference, thus `this.`*rolename* is equivalent to *rolename*. The association end opposite to *obj* must be labeled with *rolename*, and the association must be navigable in that direction. The value of the expression is a reference to the object at the other end of the link.

The expression may be used as the left-hand side of an assignment to modify a link. The right-hand side must be a reference to an object whose type is compatible with the association. If the right-hand side is `null`, the link is removed. Guards must not modify links, neither directly nor indirectly through operation calls.

It is the user's responsibility to make sure that links respect the multiplicities of association ends. Associations that are navigable in both directions are mapped to two fields, one in both end. The user must keep the fields consistent with each other.

4.3.2 Attributes

Like links, attribute values are accessed with field access expressions of the form *obj*`.`*attribute*, where *attribute* is the name of the attribute. Again, *obj* and the period may be omitted when accessing an attribute of `this`. For class attributes, the form *classname*`.`*attribute* can be used as well.

To change the value of an attribute, the expression may be used as the left-hand side of an assignment or as the operand of an increment or decrement (`++` or `--`) expression. Guards must not modify attribute values.

Because Jumbala uses reference semantics for objects like Java, attribute values are never objects but references to objects. It means that if the type of attribute `attr` is a reference type, statements such as '`attr2 = attr;`' or '`operation(attr);`' do not make copies of the object referred to by `attr` but only make new references to the same object. The primitive types `int` and `boolean` act differently. They are always passed by value.

4.3.3 Creating Objects

A `new` expression creates an object of a given class and returns a reference to the new object. The reference can be stored in an attribute or assigned to an end of a link.

If the class in a `new` expression is an active class, the new active object starts its behavior automatically. However, we assume that two run-to-completion steps cannot be executing simultaneously. The step that created the object is first finished before the new object begins to dispatch events from its event pool.

Guards must not create objects.

4.3.4 Operation Calls

Guards, methods, and activities can call operations using method invocation expressions. However, there are restrictions that models must obey. First, the operations called by guards are not allowed to change the global configuration.

Second, to prevent interference between operation calls and the event mechanism as discussed in Section 2.1.5, any method of a passive class must not call an instance operation of an active class. Also, a guard, method, or activity of an active class must not call an instance operation for any active object other than the one referred to by `this`. Calls to class operations or operations of passive classes are not forbidden.

UML does not define a way to select the implementation to execute upon an operation call. The question is non-trivial when there are several methods with the same name or when inheritance is involved. In Jumbala, the selection procedure is based on the name of the method and the types of arguments, following the rules of Java.

Argument values passed to operations are either primitive values (integers or Booleans), references to objects, or the special value `null`. Objects cannot be passed as arguments. The use of Jumbala restricts operations to have at most one return value.

4.3.5 Other Expressions

The Jumbala expressions not discussed above are arithmetic and bitwise operators, comparison and conditional operators, and type testing expressions. These are side-effect free expressions with no semantics related to UML. They can be used freely in guards, methods, and expressions.

4.3.6 Sending Signals

A `send` statement sends a signal instance to an active object. In other words, it adds an occurrence of a signal event to the event pool of that object. The object can react to the event in a subsequent run-to-completion step.

The name of the signal is determined statically, but the expression denoting the target object can be any expression evaluating to an active object, including `this`. Both active and passive objects can send signals. Several signal instances can be sent to the same object during a run-to-completion step.

Jumbala allows signals to have associated parameter values. However, the details of the semantics of signal parameters have not been fixed yet in the simulator.

4.3.7 Local Variables

A local variable declaration statement may appear in a method or an activity. A local variable declared in the effect of a transition or in the exit or entry activity of a state only exists for the duration of the activity. Thus, the values of local variables are never preserved across run-to-completion steps and they are not part of the global configuration.

A guard cannot declare local variables because Boolean expressions cannot contain statements. However, a guard can call operations that use local variables for computation, as long as the operations do not change the global configuration.

4.3.8 Other Statements

A `return` statement is used to return a value from a method. State machine activities must not contain `return` statements.

Other Jumbala statements such as `if`, `while`, and `assert` can be used in methods and activities. Blocks of statements are also allowed. The execution of a complex statement is confined to one run-to-completion step. No active object is able to react to new events until the step has finished.

For this reason, nonterminating behavior must not be modeled with an infinite `while` loop. Instead, the loop must be converted to a cycle in a state machine, effectively splitting the execution to several run-to-completion steps.

5 IMPLEMENTATION

A programming language has little value unless there exists a tool that interprets and executes the language. In Section 3.2 we introduced the Jumbala action language interpreter and its place in the tool set of the SMUML project. The interpreter is a self-contained module that executes Jumbala programs without knowledge of the UML model that the programs originate from. All connections to UML are made in the tools that interface with the interpreter.

Consequently, the interpreter resembles a programming language compiler augmented with a simple run-time environment. The interpreter has been implemented in the Python programming language [4], which is a natural choice since the other tools in the SMUML project are also written in Python. It is clear that an interpreter running on top of a high-level interpreted language like Python cannot compete in efficiency with industrial-strength compilers. However, since speed is not expected to be an issue in running UML simulations, we believe that the increased ease of development more than compensates the reduced run-time performance.

5.1 OVERVIEW

A modern compiler divides its task into phases [1], and we follow the tradition. The first phase is to parse the syntax. The program text is translated into an *abstract syntax tree*, which captures the hierarchical structure of the program together with the details that are relevant to the later phases. The abstract syntax tree is translated into an *internal code* language to obtain a simple and effective representation of the program code. A full-fledged compiler typically has several additional phases, such as optimizations, flow analysis, and machine code generation. Our implementation does not make further conversions but executes the internal code directly in a *virtual machine* that is part of the interpreter. Figure 5.1 illustrates the process.

A complete usage example is displayed in Figure 5.2. The Python script in the figure uses the interpreter to execute a Jumbala program that prints the string “Hello, world!”. It should be noted that there is no such thing as a Jumbala source file. The interpreter only supports a programmatic interface that requires calling several Python functions to achieve any effect. The solution allows great flexibility, e.g. incremental programming and direct access to the abstract syntax tree (Section 5.2.1).

As shown in the code listing, the program is passed to the interpreter as a Python string. The phases of syntax parsing (`parse`) and translation to internal code (`translate`) are followed by execution (`initialize` and `run`). The variable `env` is used to store the static environment, which functions as a symbol table that maps names to types and variables during translation. The variable `vm` holds a reference to the virtual machine. To achieve incrementality (Section 3.3), the user of the interpreter needs to define an incremental program as a string and repeat the parsing, translation, and execution phases reusing the `env` and `vm` variables.

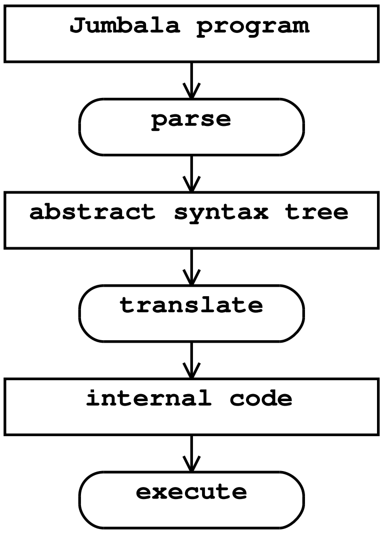

Figure 5.1: Data flow of the interpreter.

```
# Import the action language interpreter package.
import actlang
# Obtain a static environment and a virtual machine.
env = actlang.basetypes.getInitialEnvironment()
vm = actlang.basetypes.getInitialMachine()
# Make an input program.
prog = 'System.out.println("Hello, world!");'
# Parse the program into an abstract syntax tree.
tree = actlang.grammar.parse(prog)
# Translate the tree into internal code.
main = tree.translate(env)
# Load the code into the virtual machine.
vm.initialize(main)
# Execute the top-level statements.
vm.run()
```

Figure 5.2: A Python script that uses the interpreter to execute a program.

5.2 PARSING

A common way to implement the parsing phase is to use an automatic parser generator. We chose a parser generator named PLY (Python Lex-Yacc) [3], which is a lightweight parsing tool implemented in Python. The parser generator produces a lexical analyzer, which breaks the input into tokens using regular expressions, and an SLR [2] parse table, which is derived from a context-free grammar. The grammar rules are based on the Java grammar in [12], with simplifications due to reduced features. Some redundant grammar rules have been added to work around limitations of SLR parsing. PLY also claims to support LALR parse tables but the LALR generation crashes on our input.

Given an input program, the parser generates an abstract syntax tree whose structure agrees with the semantic structure of the program. Because the grammar describes a language that is a superset of Jumbala, an abstract syntax tree may be derived even from a program that contains errors, e.g. type violations. At a later phase, the abstract syntax tree is subjected to semantic checks.

As an example, the abstract syntax tree of the program

```
while (i > 0)
  i--;
```

is shown in Figure 5.3. Syntax nodes (nodes of the abstract syntax tree) are represented as framed rectangles, with child nodes drawn inside their parents. The class of a syntax node, such as WhileStatement or SimpleName, is shown in the upper part of the rectangle.

A node may have different kinds of children depending on its class. For example, the root node has a list of top-level statements as its children, the first and only one of which is *topStatements[0]*. A WhileStatement always has a *condition*, which is an expression, and a *body*, which is a statement, and so on. There is a Python class for each class of syntax nodes, with data attributes for representing the children.

The leaf node SimpleName has a data attribute *identifier* that contains a string, in this case `i`. Similarly, the IntLiteralExpr has an attribute *value*, which is an integer. The nodes GreaterThan and Minus contain no data because all the information is in the class itself. The symbols `>` and `--` are included in the figure only to illustrate the correspondence between the abstract syntax tree and the original concrete syntax.

The abstract syntax tree contains all the relevant information of the program source, with all the syntactic issues resolved, providing a clean starting point for the next phase of analysis.

5.2.1 The Abstract Syntax Tree Interface

As explained in Section 3.2, the interpreter is not the only tool that needs to understand the action language. We also need to support translation from Jumbala to a model checker. Because the parsing phase remains the same regardless of the target of translation, it seems reasonable to be able to reuse the parser.

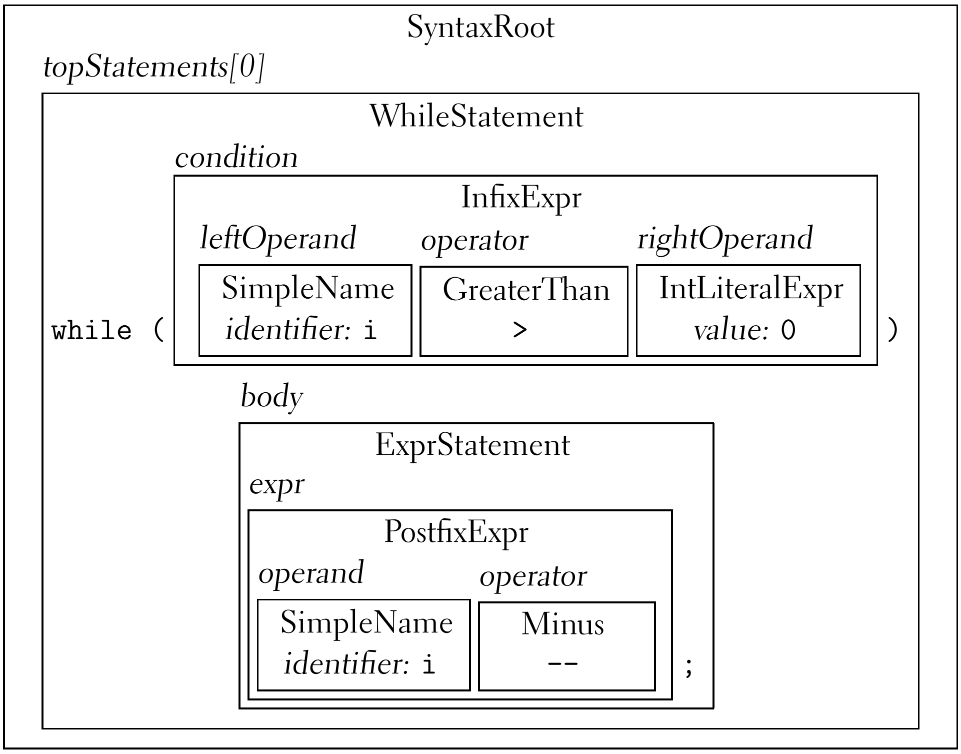

Figure 5.3: Abstract syntax tree for the program '`while (i > 0) i--;`'.

For this reason, the abstract syntax tree returned by the parser offers interface functions for examining the tree structure. It is thus possible to write an external tree traversal routine that checks the class of each syntax node and calls the appropriate interface functions to obtain the children and other data associated with the node. In practice, the interface consists mostly of one-line functions that return a data attribute of the node object. The interface has only been implemented for the classes of nodes that are expected to have relevance in formal verification.

5.3 TRANSLATION TO INTERNAL CODE

One way to execute a program is to traverse its abstract syntax tree. The program counter is a pointer to a syntax node, and a step of execution involves evaluating the statement or expression that the syntax node represents and moving the program counter to another node.

There are a number of reasons why we did not take this approach. The abstract syntax tree contains redundant structures due to the user-friendly features of Jumbala. For example, the expression `i++` is equivalent to `i += 1`, and the iteration constructs `while` and `for` are similar in nature. The Python function that executes a `while` statement would be very close to the one that executes a `for` statement.

Furthermore, if we want static typing or any compile-time checks at all, we would have to traverse the abstract syntax tree twice, first to make sure the program is valid and then to execute it. Both traversals need to analyze the types of expressions, causing redundant work unless the type information is attached to the syntax nodes during the first round. Also, if performance becomes an issue in the future, any automatic optimizations would require rewriting parts of the syntax tree.

Instead of executing the syntax tree directly, we introduce another level of indirection and translate the abstract syntax tree into an *internal code* language. The translation is done during the same phase as semantic checking, which is convenient because they both use the same information of types of expressions. The internal code language is much simpler than Jumbala, resembling the assembly language of a microprocessor. The code is represented as a data structure that can be executed without knowledge of the abstract syntax tree. Compilers use similar languages as an intermediate form before generating machine code [1].

Each syntax node class has a translation function that calls the translation functions of the child nodes, performs semantic checks, and generates the necessary data structures, such as a list of internal code instructions or a Python object representing a Jumbala reference type. The abstract syntax tree itself is left intact by the translation phase.

5.3.1 The Internal Code Language

The internal code language is an abstraction used privately in the interpreter. It is generally not written or read by humans, and it is not meant to be stored on disc or distributed among computers, so it does not have to meet the same standards as a programming language. Instead, the language should be small and fast to execute with a virtual machine, while still being powerful enough to allow a reasonably succinct translation from Jumbala.

The code language is unstructured in the sense that the only branching and iteration constructs are unconditional and conditional jumps to labels. The language is weakly typed. The only types are a 32-bit integer and a reference to an object. Boolean values are represented as the integers 0 and 1. The language is unsafe but, as long as it is only generated from valid Jumbala programs, this should never be a problem.

One difficulty in interpreting a high-level language such as Jumbala is storing temporary data values. Consider the expression `area = width() * height()`. To keep the internal code instructions simple, we want to divide the expression to at least three parts: one that invokes `width`, one the invokes `height`, and one that performs multiplication. The problem is that the return value of the method `width` has to be stored somewhere for the duration of the invocation of `height`.

A possible resolution is to use temporary variables to hold intermediate results. This leads to a three-address code [1] instruction language, where instructions have the form x := y *op* z. Another possibility is to use a last-in-first-out operand stack instead of explicitly named temporaries. During the implementation, the stack based approach was found to be a more elegant solution. Using a stack eliminates the need to keep track of the names and scopes of temporary variables. The suitability of an operand stack was not a surprise. For the most part, the semantics of Jumbala is equivalent to Java, and the Java Virtual Machine [24] itself uses operand stacks.

As a result, the internal code is similar to Java bytecode. A difference is that instructions of the internal code language are stored as Python objects, one graph of objects for each method or constructor, instead of bytes. Also, the internal code language is tuned for simplicity, not performance.

Below are examples of the nearly 50 different kinds of internal code instructions.

- **IntMultiply** removes two integers from top of the operand stack, multiplies them, and pushes the result on top of the operand stack.
- **CallDynamic** invokes an instance method. The method is chosen based on the class of an object whose reference is taken from the operand stack.
- **StoreAuto** reads the value of a local variable and pushes it on top of the operand stack.
- **IfGoto** removes an integer from top of the operand stack, and if it is nonzero, jumps to a given label.

Currently the translation produces rather naive and inefficient code. If performance becomes a bottleneck later, optimization patterns may be applied to the generated internal code.

5.4 RUN-TIME ENVIRONMENT

The run-time environment is a part of the interpreter, consisting of a simple virtual machine and the objects representing the user-defined Jumbala reference types.

The virtual machine (VM) is an object that executes internal code instructions in a loop until the program finishes. The VM also supports execution in steps, one instruction at a time, although there is no support for debugging other than printing the state of the VM. Before execution, the VM is initialized with a reference to the instructions that represent the top-level statements of the program.

The state of the virtual machine consists of the operand stack, the values of all class variables, and a call stack. The call stack has one stack frame for the top-level statements plus one for each nested method or constructor invocation. A stack frame contains a program counter, which is a reference to an instruction, and values for local variables, method or constructor parameters, and the `this` reference. When a program has finished executing, the state of the VM consists of the Jumbala configuration discussed in Section 3.10, acting as an initial state for executing incremental programs.

Jumbala objects are represented as Python objects of the class Instance, with data attributes for holding the values of instance variables. If the value of a Jumbala variable is a reference to an object, it is represented in the implementation as a reference to an Instance. The operand stack may also contain references to Instances. As a result, there is no need for an explicit heap of Jumbala objects. This simple design is possible because Python has a reference semantics similar to Jumbala, and it has a great advantage: there is absolutely no implementation trouble regarding the destruction of objects.

When an Instance becomes unreachable from the VM, the garbage collector of the Python interpreter automatically disposes of the object.

5.4.1 Native Methods

The interpreter has an interface to support user-defined native methods (Section 3.9.1). A native method is declared in a Jumbala program but its implementation is given in Python. If a program declares native methods, the user has to provide a mapping from native method declarations to Python functions before the translation phase.

When a native method is invoked, the interpreter calls the associated Python function. The method may take parameters of any type. Arguments of the primitive types `int` and `boolean` are converted to their Python counterparts. Objects, including strings, are passed as references to Instance objects. Conversely, the implementation may return a value back to the Jumbala program.

5.4.2 Predefined Classes

The Jumbala language itself is dependent on a set of primordial classes such as `Object` and `String`, which must exist when a translation of a program begins. In addition, the current implementation defines a class containing `print` and `println` methods similar to those in Java. Therefore it is legal to write simple test programs such as '`System.out.println(12+7);`' that print text to the standard output. Internally, the printing methods are implemented using the native interface described above.

Since the action language interpreter is not a general-purpose programming tool but a part of a UML tool set, a comprehensive standard library is not included in the implementation. When needed, such a library may be defined externally by writing a Jumbala program that declares the necessary classes and methods, possibly using native implementations for low-level functionality. The "linking" of the library to a main program is done using incrementality. The library program is first translated and executed using the interpreter, and only then the main program that uses the library is translated and executed as an incremental program.

5.5 ERROR HANDLING

Large portion of the implementation of any programming language interpreter concerns detecting and reporting errors in the input program. Detecting errors is laborious but, in principle, straightforward, since the interpreter only needs to compare the input to the language specification. However, producing useful error messages for the user is more intricate. Ideally, when an interpreter finds a compile-time error, it should not only point out the location of error but also guess the original intention of the programmer and give an error message that guides the programmer to the right direction.

The Jumbala interpreter returns error messages by raising Python exceptions. Separate exception classes are defined for compile-time errors and run-time errors. The quality of error messages is adequate for small programs but falls behind industrial-strength compilers.

5.5.1 Compile-Time Errors

Raising an exception during parsing or translation phase immediately aborts the analysis and returns control to the point where the interpreter was invoked. This is perfect for reporting an error. Exceptions eliminate the need to pass special error values from functions.

However, since we want to be able to catch several compile-time errors at the same time, the scheme has to be extended. A class named ErrorProxy is used internally to accumulate error messages from several parts of the program. When further progress is obstructed because of errors, the ErrorProxy raises an exception with all the error messages attached.

Syntax Errors

Syntax errors occur when the program does not comply with the rules of the context-free grammar. They are detected by the parser generated by PLY. Although PLY supports recovery and resynchronization using special error rules, these features have not been employed. Instead, when an unexpected input element is encountered, a simple message is added to an ErrorProxy. Parsing continues with the default recovery rules defined in PLY. At the end of the parsing phase, the ErrorProxy reports the errors to the user and no abstract syntax tree is generated.

The simple method of just reporting unexpected input tokens is sufficient as long as the user is familiar with the language syntax. More sophisticated techniques may be adopted in the future.

Semantic Errors

The translation phase checks the semantic meaning of a program. Errors caught at this phase include type violations, usage of undeclared names, attempts to instantiate an abstract class, and so on.

Semantic errors are usually localized at a branch of the abstract syntax tree. If two unrelated branches contain an error, the ErrorProxy mechanism makes it possible to detect both of them. Consider the following program.

```
int a = b;
int c = "three";
```

Even though the translation of the first statement raises an exception because `b` is undeclared, the implementation catches the exception and tries to translate the second statement to find out if it contains errors too (it does, `"three"` is not an `int`). At the end, an exception is raised and the user gets a message for both errors.

The current implementation makes no effort to remove duplicate messages originating from the same error.

5.5.2 Run-Time Errors

Run-time errors are caught by the virtual machine. The internal code produced at the translation phase contains the necessary checks, and if the checks fail during execution, an exception is raised by executing a dedicated instruction of the internal code language. The module that invoked the interpreter may catch the exception but it is not possible to resume execution without resetting the virtual machine.

5.5.3 Traceability

The process of program interpretation contains many phases, and errors may be found at any stage. A challenge in reporting errors in a useful way is *traceability*, i.e. the ability to map the point where an error was found back to an exact location in the original program.

For this reason every node in the abstract syntax tree is associated with a range of line numbers in the original program. Thus, all compile-time error messages have an attached line number that point out at least approximately the source of the error. Also the internal code produced at the translation phase contains references to the program, or actually, to the abstract syntax tree. The run-time error messages do not yet employ this information, but in principle the virtual machine knows which line of the program it is executing.

In the UML framework the problem of traceability is not completely solved by tracking down the line numbers. If a UML model has an error that is detected by the action language interpreter, the author of the model does not want to see a line number referring to Jumbala program but a specific spot in the UML diagram that contains the error. Therefore a total solution to displaying useful error messages will not be found until the UML tool set has found its final form.

6 RELATED WORK

UML is a widely used language and a lot has been published on the subject. However, not all modeling applications require having an action language. If UML is used only to describe structural aspects of a system, there is no need for specifying actions. Even behavioral modeling is sensible without an action language as long as the model is not intended to be machine-executable. In many situations, e.g. when describing use cases, an informal approach such as pseudocode or natural language is quite enough.

The use of formal action language becomes unavoidable when the actions in a model have to be executed or analyzed by a computer. Thus, behind every UML action language there is a tool (or an idea of a tool) that performs, for instance, automatic code or test generation or formal verification.

The field can be roughly divided in two. Many industrial modeling tools support a large number of UML features with an expressive action language and capability for code generation. On the other hand, academic research typically concentrates on formal verification of UML models with more emphasis on handling concurrency than on interpreting actions.

On the commercial side, the model-driven software development tool Tau by Telelogic [34] has its own action language. The language has a textual Java-like syntax, and in addition, some of the actions have a graphical syntax. Program code can also be written directly in the language targeted for code generation.

The action language in the context of Executable UML [30] is called ASL. It is used in Kennedy Carter's iUML tool [19]. The idea of ASL is to enable writing implementation-free specifications of actions. ASL has built in high level expressions with semantics such as "find all instances of Account where balance < 0.0". Both ASL and the Tau action language support a rich set of features, including integer, real, and string data types, loops and branching, creation of new objects, definition and invocation of methods, sending signals with parameters, nondeterministic choices, and timer operations. The extent of these languages is comparable to Jumbala, although our approach lacks support for real numbers, expressions that return nondeterministic values, and real-time models. The difference is explained by the different sets of goals. Because our aim is to support formal verification, we have not included some more intricate language features, especially those not present in our baseline language Java. Other industrial-level action languages are listed in [30, p. 247].

The idea of applying formal verification on UML-type state machine models is not new. Latella et al. [22] present a translation from UML state machines to PROMELA, the input language of the SPIN model checker [17]. They only allow a model to contain a single state machine, which may however have several orthogonal regions. Another translation to PROMELA is by Mikk et al. [27]. Their input language is not UML state machines but Statecharts, which is a similar formalism with slightly different semantics. Del Bianco et al. [5] have written a program that translates real-time UML models to timed automata that are verified by the tool Kronos. All these works have the limitation that no data attributes can be associated with objects or state machines. The only way to model data is to encode it as the states of a state machine. Consequently, there is no action language, and the only effect that a transition may have is to send a signal with no parameters.

The vUML tool by Lilius and Porres [23] also verifies UML state machine models with SPIN. The tool supports object attributes of integer type, which can be accessed from guards and effects of transitions. A dedicated action language is not introduced. Guards and actions are written directly in PROMELA, and dynamic creation of new objects is not allowed.

Haustein and Pleumann [16] suggest an action language named ASOQ that has OCL as a subset. OCL (Object Constraint Language) is a side-effect free query language used in the specification of UML. ASOQ incorporates side effects by including assignments and non-query method invocations. However, the considered problem domain is not the verification of reactive systems but a run-time environment for web applications. Consequently, there is no support for asynchronous signal transmission in ASOQ.

An action language for reactive systems has been developed in the Omega project [13], whose aim is to define a development framework for embedded systems based on UML and formal techniques. The Omega Action Language OMAL [21] has a minimal syntax and is statically typed. The features include loops and branches, nondeterministic choices, parallel interleaving, object creation, and sending of signals with parameters. Integer, real, and string data types are supported, and a subset of OCL is used for accessing collection types. Complexity of statements is limited to avoid the need for temporary variables in implementation. For example, nested operation calls are not allowed, and the only way to use the result of an operation call is to assign the return value to an attribute.

The tool Hugo/RT by Knapp et al. [20] verifies UML state machine models against scenarios specified by UML collaborations. Hugo/RT can translate models to several verification back-ends, including SPIN. The tool uses a proprietary action language that is similar to OMAL. However, the language does not support string or real data types, loop constructs, return values of operators, or creation of objects.

Compared to other action languages designed in a verification framework, Jumbala has a rich set of expressions, with support for all Java operators, type hierarchies and polymorphism, and arbitrary nesting of method invocations. Jumbala also allows declaration of variables local to a block, and dynamic creation of objects and arrays, which are not possible in most UML verification tools. Other action languages have features that are not present in Jumbala, such as parallel composition of statements and nondeterministic choice of values. We assume that parallelism and nondeterminism are modeled at the level of state machines, not at the level of actions. This reflects a conscious choice to favor modeling nondeterministic flow of control instead of nondeterministic data. Although Jumbala has features that make it possible to model complex computation, it is possible that the full potential will not be utilized in the verification process because of technical restrictions.

7 DISCUSSION AND CONCLUSIONS

We have specified a UML action language named Jumbala and implemented an interpreter for the language in Python. Jumbala is suitable for specifying activities in UML diagrams. At the same time, it is a complete object-oriented programming language based on Java.

The Jumbala interpreter is utilized by two other tools that have been developed in the SMUML project. A UML simulator executes UML models using the interpreter as a back-end, as described in Section 3.2. The currently supported UML subset is slightly different from that presented in Chapter 2. The second tool translates executable UML models to the input language of the SPIN model checker [17]. Action statements are translated from Jumbala using the abstract syntax tree interface of the interpreter (Section 5.2.1).

7.1 IMPLICATIONS OF FOLLOWING JAVA

The biggest design decision has been to choose Java as the basis of the action language. We have benefited from the choice during the design phase of the language. Having an established foundation has saved us from making (possibly bad) choices regarding the details of syntax and semantics. Java also has many features that are meaningful in the context of UML and that we could employ directly, such as classes, interfaces, enumerated types, fields (attributes in UML), and methods. The coherence between UML and Java is probably not a coincidence, as both languages evolved during the rise of object-oriented methods in the 1990's.

Despite the similarities, UML is not confined to support a single programming language. The class structure in UML is more permissive than in Java, allowing e.g. multiple inheritance of classes. The choice of a Java-based action language restricts the kinds of models that can be handled. For example, UML specifies an action named ReclassifyObjectAction, which changes the classifier of an existing object. Such an action severely contradicts the idea of a statically typed action language. Therefore we do not support models that rely on reclassifying objects.

At this point we have to remember that the ultimate goal is not to be able to interpret every conceivable UML construct but to formally verify behavioral aspects of nontrivial models. In this setting the real challenge is not in defining an action language that is expressive enough but in developing a verification framework that scales up to industrially relevant problems. The Jumbala language already has powerful features such as user-definable data types and dynamically allocated arrays, which must be used with care to avoid ruining verifiability. Language restrictions such as static typing are a way to increase the chances of successful verification.

Much of the effort in specifying and implementing Jumbala has gone into features that are technically redundant but that make life easier for the programmer. Examples include the increment and decrement operators (`++` and `--`), the ternary conditional operator (`? :`), the `switch` statement, and method overloading (allowing different methods with the same name) [1]. All these constructs can be circumvented in a program with modest effort. Another feature that complicates the specification but makes the language appear simpler is that a period between names can denote both field access and scope resolution. In practice it means that `X` in the expression `X.y` can be either a type or a variable, depending on the context. Disambiguation of names would have been simpler if we had introduced a dedicated notation for scope resolution.

The practical value of these features can be debated, especially since the number of Java programmers in the world is several orders of magnitude larger than the number of potential Jumbala programmers. We omitted several features of Java to make the language simpler but we could have gone even further without making the language too unwieldy to use. As a downside, this would have increased the number of differences between Jumbala and Java. One of the reasons for replicating much of the syntax of Java is that existing snippets of Java code can be used without having to convert them from one syntax to another.

The proper balance between simplicity and user friendliness depends on how the language is used. We might be inclined to incorporate new shortcut notations from Java after hearing feedback from programmers.

7.2 FUTURE WORK

The direction of future development depends on the kinds of models that we are required to support. The set of features that UML allows is huge, but we are only interested in those that are actually used in industrial models. The expansion of the supported UML subset mainly affects the simulator, but it may also affect the requirements for the action language.

A fundamental feature that is not yet supported in the simulator is attributes with multiplicities (Section 2.1.1). Arrays in Jumbala can hold multiple elements but their length cannot be changed after creation. One way to represent an attribute with variable multiplicity is to wrap it in a suitable class such as `List` that is implemented in Jumbala. This requires no extensions to the action language but the resulting syntax for accessing the attribute would become rather verbose. Another solution is to add more built-in container types to the action language. This would strengthen the connection to UML but at the same time make the language more complex and take it further away from Java.

Other possible extensions to the language include support for narrower than 32-bit integers, unbounded integers, and rational numbers.

The current implementation of the Jumbala interpreter is not fast. This is expected because the execution of a single instruction of internal code (Section 5.3) involves calling several Python functions in the virtual machine. This is only an issue when simulating models, because the actual formal verification phase will not be built on top of the interpreter. If simulation of large models turns out to be painfully slow, performance of the virtual machine

[1] It is interesting to note that Python – which is considered to be a programming language that maximizes productivity – does not have a counterpart for any of these shortcut notations.

has to be boosted. Perhaps the most straightforward way is to try to speed up the Python interpreter using a just-in-time compiler such as Psyco [31]. We could also optimize the implementation of the virtual machine by hand so that it executes code instructions faster. Furthermore, the number of instructions to execute could be reduced by applying optimization patterns to the generated internal code.

The existing implementation for simulating models does not fully enforce consistency between a UML model and its run-time instances. When executing a well-formed model, the evaluation of a guard expression never has any side effects, and the two ends of a bidirectionally navigable link always point at each other. In these respects, the simulator relies on the user to make only well-formed models. To make the system more robust, the simulator-interpreter interface could be improved by implementing run-time checks that give an error message if an execution violates the model, or by limiting write access to links so that illegal configurations could not be reached.

A related future task is to continue the work on defining the semantics of Jumbala with respect to UML. At present, we have informal execution semantics for Jumbala programs that are treated as self-contained entities with no connection to UML model elements. The implementation of the simulator makes the connection by replicating the UML class structure in a Jumbala program, although not all features of UML are yet supported. This could be the starting point for defining the full semantics of Jumbala expressions as UML actions.

ACKNOWLEDGEMENTS

This report is a reprint of my Master's thesis. I want to express my gratitude to Prof. Ilkka Niemelä, the supervisor of the thesis, for his insight and guidance and for helping me see what is essential. I also thank my instructor, Dr. Timo Latvala, who was never too busy to sit down and study my sketches, and who did not spare efforts in giving valuable comments on the work.

I would like to give special thanks to Dr. Tommi Junttila for the numerous discussions regarding the subtleties of UML and for offering me the opportunity to work with this subject in the first place.

This work has been funded by Tekes (Finnish Funding Agency for Technology and Innovation), Nokia Oyj, Conformiq Oy, and Mipro Oy. Their support is gratefully acknowledged.

BIBLIOGRAPHY

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[3] David Beazley. PLY (Python Lex-Yacc). Software. `http://www.dabeaz.com/ply/`.

[4] David Beazley. *Python essential reference*. New Riders, second edition, 2001.

[5] Vieri Del Bianco, Luigi Lavazza, Marco Mauri, and Giuseppe Occorso. Towards UML-based formal specifications of component-based real-time software. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering, 6th International Conference (FASE 2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 118–134. Springer, 2003.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference (TACAS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[7] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[8] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, second edition, 2005.

[9] Edmund M. Clarke. Counterexample-guided abstraction refinement. In *10th International Symposium on Temporal Representation and Reasoning / 4th International Conference on Temporal Logic (TIME-ICTL 2003)*, page 7. IEEE Computer Society, 2003.

[10] Edmund M. Clarke and Robert P. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[11] Bruce Powel Douglass. *Real-Time UML*. Addison-Wesley, third edition, 2004.

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[13] Susanne Graf and Jozef Hooman. Correct development of embedded systems. In Flávio Oquendo, Brian Warboys, and Ronald Morrison, editors, *Software Architecture, First European Workshop, (EWSA 2004)*, volume 3047 of *Lecture Notes in Computer Science*, pages 241–249. Springer, 2004.

[14] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[15] David Harel and Amir Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO, ASI Series*, pages 447–498. Springer, 1985.

[16] Stefan Haustein and Jörg Pleumann. OCL as expression language in an action semantics surface language. In Octavian Patrascoiu, editor, *OCL and Model Driven Engineering, UML 2004 Conference Workshop*, pages 99–113. University of Kent, 2004.

[17] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[18] Jukka Honkola, Sari Leppänen, and Teemu Tynjälä. Modeling the SpaceWire architecture with Lyra. In *Fifth International Conference on Application of Concurrency to System Design (ACSD 2005)*, pages 15–24. IEEE Computer Society, 2005.

[19] Kennedy Carter Ltd. iUML. Software. `http://www.kc.com/products/iuml.php`.

[20] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model checking timed UML state machines and collaborations. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium (FTRTFT 2002)*, volume 2469 of *Lecture Notes in Computer Science*, pages 395–416. Springer, 2002.

[21] Marcel Kyas, Joost Jacob, Ileana Ober, Iulian Ober, and Angelika Votintseva. *OMEGA syntax for users*, January 2005. Omega Deliverable D2.2.3 Annex 1.

[22] Diego Latella, István Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[23] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. Technical Report TUCS-TR-272, Turku Centre for Computer Science, Finland, May 18, 1999.

[24] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[25] Kenneth L. McMillan. Interpolation and SAT-based model checking. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification, 15th International Conference (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[26] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In Hubert Garavel and John Hatcliff, editors, *Tools*

and Algorithms for the Construction and Analysis of Systems, 9th International Conference (TACAS 2003), volume 2619 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[27] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing statecharts in PROMELA/SPIN. In *2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98)*, pages 90–101. IEEE Computer Society, 1998.

[28] Object Management Group. *MDA Guide Version 1.0.1*, 2003. `http://www.omg.org/cgi-bin/doc?omg/03-06-01`.

[29] Object Management Group. *UML 2.0 Superstructure Specification*, 2005. `http://www.omg.org/cgi-bin/doc?formal/05-07-04`.

[30] Chris Raistrick, Paul Francis, John Wright, Colin Carter, and Ian Wilkie. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004.

[31] Armin Rigo. *The Ultimate Psyco Guide*, 2005. `http://psyco.sourceforge.net/psycoguide.ps.gz`.

[32] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, second edition, 2004.

[33] Jørgen Staunstrup, Henrik Reif Andersen, Henrik Hulgaard, Jørn Lind-Nielsen, Kim Guldstrand Larsen, Gerd Behrmann, Kåre J. Kristoffersen, Arne Skou, Henrik Leerberg, and Niels Bo Theilgaard. Practical verification of embedded software. *IEEE Computer*, 33(5):68–75, 2000.

[34] Telelogic. Telelogic Tau G2. Software. `http://www.telelogic.com/`.

[35] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1998.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE RESEARCH REPORTS

HUT-TCS-A88	Harri Haanpää	Minimum Sum and Difference Covers of Abelian Groups. February 2004.
HUT-TCS-A89	Harri Haanpää	Constructing Certain Combinatorial Structures by Computational Methods. February 2004.
HUT-TCS-A90	Matti Järvisalo	Proof Complexity of Cut-Based Tableaux for Boolean Circuit Satisfiability Checking. March 2004.
HUT-TCS-A91	Mikko Särelä	Measuring the Effects of Mobility on Reactive Ad Hoc Routing Protocols. May 2004.
HUT-TCS-A92	Timo Latvala, Armin Biere, Keijo Heljanko, Tommi Junttila	Simple Bounded LTL Model Checking. July 2004.
HUT-TCS-A93	Tuomo Pyhälä	Specification-Based Test Selection in Formal Conformance Testing. August 2004.
HUT-TCS-A94	Petteri Kaski	Algorithms for Classification of Combinatorial Objects. June 2005.
HUT-TCS-A95	Timo Latvala	Automata-Theoretic and Bounded Model Checking for Linear Temporal Logic. August 2005.
HUT-TCS-A96	Heikki Tauriainen	A Note on the Worst-Case Memory Requirements of Generalized Nested Depth-First Search. September 2005.
HUT-TCS-A97	Toni Jussila	On Bounded Model Checking of Asynchronous Systems. October 2005.
HUT-TCS-A98	Antti Autere	Extensions and Applications of the A^* Algorithm. November 2005.
HUT-TCS-A99	Misa Keinänen	Solving Boolean Equation Systems. November 2005.
HUT-TCS-A100	Antti E. J. Hyvärinen	SATU: A System for Distributed Propositional Satisfiability Checking in Computational Grids. February 2006.
HUT-TCS-A101	Jori Dubrovin	Jumbala — An Action Language for UML State Machines. March 2006.

ISBN 951-22-8102-3
ISSN 1457-7615