UNIVERSITY OF OSLO
Department of Informatics

Supporting Distributed Active Objects: A Virtual Machine for Creol on the Java Platform

Master thesis

Ivar Alm

1st May 2006

Abstract

Distributed systems are becoming increasingly important. In order to facilitate the development of distributed systems, new high-level abstractions and programming languages may be convenient. Creol is an experimental high-level object-oriented language for distributed objects. This thesis investigates how to create a low-level run-time environment for Creol by proposing a computational model for the language. A prototype of the model is implemented on the Java platform; this prototype serves as a virtual machine on which Creol programs can be executed and tested. The thesis looks into subject areas such as distribution, concurrency, multiple inheritance, and interleaved execution of statement lists.

Preface

This thesis is part of the Creol research project[1] at the Department of Informatics at the University of Oslo. The project investigates programming constructs and reasoning control in the context of open distributed systems.

I would like to thank my supervisor Einar Broch Johnsen for all his help throughout my thesis work. He has been a great support and has always been available with an open door policy, and given me excellent guidance and constructive criticism. My fellow student Øystein Torget deserves thanks for continuous feedback on my work and for inspiring discussions. I also appreciate Arild Torjusen's careful read-through of the thesis.

[1] `http://www.ifi.uio.no/~creol/`

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Thesis Outline 3

2 Background **5**

2.1 The Creol Language 6

2.1.1 Interfaces 7

2.1.2 Classes 8

2.1.3 Methods 10

2.1.4 Imperative and Functional Code 11

2.1.5 Example: The Santa Claus Problem 15

2.2 Java Concurrency 19

2.2.1 Java Threads 20

2.2.2 Data Synchronization 22

3 A Computational Model for Creol **27**

3.1 Model Objectives 27

3.2 The Model 28

3.3 Structure and States 28

3.3.1 The CVM 29

3.3.2 The Central 30

3.3.3 Class Definitions 31

3.3.4 The Object 34

3.3.5 The Process 36

3.4 Computations 38

3.4.1 Initialization 39
3.4.2 The Central 40
3.4.3 The Object 42
3.4.4 Message Processing 47
3.4.5 Process Scheduling 49
3.4.6 Process Execution 52
3.4.7 Message Transportation 62
3.5 Summary 63

4 Implementation of the Creol Virtual Machine 65
4.1 Preliminaries 65
4.1.1 Java Properties 65
4.1.2 JVM Assumptions 66
4.2 Implementation Overview 67
4.2.1 Main CVM Parts 67
4.2.2 Activity: Flow of Control 68
4.2.3 Classes and Interfaces 69
4.3 Implementation Details 71
4.3.1 Creol Program Representation 71
4.3.2 Initialization of the CVM 74
4.3.3 The Central 75
4.3.4 The Creol Object 78
4.3.5 Messages and Message Transportation 87
4.4 Example Run: The Santa Claus Problem 90
4.5 Summary 91

5 Multiple Inheritance 93
5.1 Creol and Multiple Inheritance 95
5.1.1 Example: Combining Authorization Levels 97
5.2 Extending the Model 100
5.2.1 Changes in the Structure 100
5.2.2 Changes in the Computation 102
5.3 Extending the Implementation 106
5.3.1 Changes to the Creol Program Representation 107

5.3.2 Changes to the Central's Services 107

5.4 Example Run: Authorization Policies 110

5.5 Summary 112

6 CVM Intercommunication and Remote Objects 113

6.1 New Creol Language Constructs 113

6.1.1 Virtual Machines 113

6.1.2 Remote Objects 114

6.1.3 Example: File Downloads 116

6.2 Extending the Model 118

6.3 Extending the Implementation 119

6.3.1 Background: Java RMI 120

6.3.2 Overview of the Changes 123

6.3.3 Detailed Changes 124

6.4 Example Run: Distributed Santa Claus Problem 132

6.5 Summary 134

7 Conclusion 135

7.1 Contributions 135

7.2 Further Work and Research 136

Bibliography 139

A Creol Examples 145

A.1 The Bounded Buffer 145

A.2 The Santa Claus Problem 146

A.3 Authorization Policies 149

B Java Representation 151

B.1 The Bounded Buffer 151

C Prototype Notes 155

C.1 Details 155

C.2 Download and Use 159

Chapter 1

Introduction

In recent years, distributed programming has become increasingly important with the widespread use of Internet, faster networks, and less expensive multiprocessor systems. The properties of distributed systems are different from those of non-distributed systems; e.g., delay or loss of communication can occur in distributed systems [16]. Today's leading programming languages, such as Java, C++, and C#, are primarily developed for sequential systems. These languages conventionally support only synchronous remote method calls. Synchronous calls result in unnecessary waiting in the distributed setting [18]. Furthermore, error handling in case of network errors are low-level and difficult. Languages designed for sequential systems are far from perfect for developing efficient and reliable distributed applications.

Creol is a new object-oriented programming language specifically designed for distributed systems. In Creol, objects are active and run concurrently, each with its own processor. Communication between objects is asynchronous, and objects may perform other tasks while waiting for the return of a method call. The execution of methods is seen as processes, and control is transferred between processes by explicit processor release points.

A new programming language needs to be tested to reveal its flaws and weaknesses and to demonstrate its strengths. A virtual machine for Creol has been developed [2] in the language Maude [5]. Using this virtual machine to test Creol programs has revealed some flaws in the language and hence contributed to the development of Creol. However, Maude has some undesirable properties in this context such as its inability to perform random executions. For a given program at most two different executions are available. This severely limits the possibilities for testing nondeterministic constructs and parallelism in the language. Therefore, a runtime-environment which offers pseudo-random execution has been

developed [19] by using Maude's reflective capabilities. Still, the run-time environment does not support interaction with a user or a file system, parallelism and distribution are only conceptual, and the execution is inefficient.

The language OUN (Oslo University Notation) [27] is a precursor for Creol. It addresses system specification and design. OUN has many similarities to Creol such as active objects, objects typed by interfaces and implemented by classes, multiple inheritance, and asynchronous method invocations. However, OUN has a simpler execution model than Creol, as OUN objects do not have inner processes and method executions are not interleaved. An OUN to Java compiler has been developed [28]; this compiler translates OUN programs into Java programs. Hence, OUN programs can be executed efficiently. The thread models of Java and OUN differ, and Java does not support multiple inheritance. The properties of Java has led to some restrictions on how much of the OUN language can be compiled into Java code; e.g., multiple inheritance is not supported.

Our goal is to create a virtual machine which does not impose any restrictions on the language but is still reasonable efficient. Translating from Creol to Java (or a similar language) is not possible without imposing restrictions. Instead, we develop a virtual machine for Creol on the Java platform. Hopefully, our work will contribute to the Creol project by suggesting how a low-level run-time environment can be developed, and give a run-time environment in which Creol programs can be executed and tested. The latter will in turn test Creol as a programming language. In the next section we define the problem statement of this thesis more precisely.

1.1 Problem Statement

Creol is an object-oriented language which targets distributed systems by proposing an asynchronous communication model for concurrent objects. The language has an abstract operational semantics, formally defined in Rewriting Logic [24]. At this level of abstraction, method activations are achieved through code duplication and distribution is merely conceptual. The main goal of this thesis is to bring Creol closer to an efficient low-level run-time environment by answering the following questions:

- Can a run-time model for Creol be defined which supports code sharing?

- Can Creol and its operational model be extended to support real distribution?

By *code sharing* we mean that multiple method activations share the same code segment. By *real distribution* we mean that objects are distributed among different physical machines.

In order to address the above questions in a specific way, we develop

- a new imperative operational model
- a prototype of this model, implemented in Java

The prototype is a *proof of concept*, that is, it will demonstrate the model's feasibility of computing Creol programs.

1.2 Thesis Outline

The rest of this thesis is structured as follows. Chapter 2 presents the Creol language and Java concurrency. Chapter 3 presents a computational model for Creol intended to serve as a basis for a low-level implementation in an object-oriented language. Chapter 4 presents the implementation of the Creol virtual machine in Java. Chapter 5 presents multiple inheritance in the Creol language and extends the model (from Chapter 3) and the implementation (from Chapter 4) to support multiple inheritance. Chapter 6 extends the Creol language, the model and the implementation so that we get real distribution of objects. Chapter 7 closes the thesis by summarizing the contributions of the thesis and giving suggestions for further work.

Chapter 2

Background

In this chapter we take a look at the Creol language (Section 2.1) and concurrent programming in Java (Section 2.2). The example "Bounded buffer" illustrates concepts in both languages and will be presented next. It is assumed that the reader is familiar with basic notions of object oriented languages in general and Java in particular.

The Bounded Buffer

A producer computes a stream of information. This is passed to a consumer which analyzes it. The rate at which the producer computes information can vary over time, and so can the rate at which the consumer is able to analyze it. The different rates can affect the efficiency of the program; i.e., the producer must wait for the consumer or vice versa. Therefore, a *buffer* is used to even out the different rates. A producer adds information to the buffer and the consumer fetches it. As we do not want the buffer to use to much memory, the buffer is *bounded*; i.e., the size of the buffer is limited.

Hoare [14] defines the buffer by two abstract operations *append* and *remove* on a sequence of portions:

- *append*(x: portion):
  sequence := sequence ∩ <x>;
- *remove*(**result** x: portion):
  x:= *first*(sequence); sequence:= *rest*(sequence);

where ∩ denotes concatenation of two sequences, <x> is the sequence whose only element is x, *first* selects the first item of a sequence and

rest denotes the sequence with its first item removed. Here, x has type *portion*, a supertype of all types.

The buffer is *bounded*, that is, the length of the sequence is limited. This can be defined by a predicate

- *len*(sequence) <= limit

which must always be true. An append operation can only be done if the length of the resulting sequence does not exceed the limit. Similarly, the remove operation can only be done if the sequence is non-empty.

2.1 The Creol Language

Creol is a programming language under development at the University of Oslo.[1] It addresses distributed systems, and is based on concurrent objects. Each object has its own *thread of control*, that is, the threads of control are not separated from the objects as in, e.g., Java [12] (in Java, the thread of control follows the method calls). Conceptually, a Creol object has a *processor*, and methods are executed by *processes*. Only one process can be executed at a time. The thread model of Creol has two important impacts: an object acts like a monitor, and objects can execute concurrently.

Communication between objects can be done by either synchronous or asynchronous method calls. Creol has a notion of processor release points which enables the objects to do other tasks while waiting for the reply to an asynchronous method invocation. Processor release points also enables the objects to dynamically change between active and reactive behavior (client and server).

Objects are typed by interfaces and implemented by classes. Creol supports multiple inheritance both at the interface and class level. Interface inheritance gives a flexible way to type objects, whereas class inheritance makes it possible to reuse code.

In the next sections, we take a look at interfaces, classes, methods and method bodies (sections 2.1.1–2.1.4). We will define the syntax of Creol by a BNF syntax where

- Square brackets are used as meta parenthesis: [. . .] is used for optional parts (zero or one), [. . .]* is used for optional, repetitive parts (zero, one or more), and [. . .]+ is used for repetitive parts (one or more).

[1]This presentation of the Creol language is based on [2, 18, 21].

- *Italic words* are used as meta symbols.

To exemplify Creol constructs, we use the “Bounded Buffer” example (defined in the introduction of this chapter). In Section 2.1.5 we present a larger example: “The Santa Claus Problem”.

2.1.1 Interfaces

Creol uses interfaces to define both how objects communicate and to give semantic requirements for the objects. The semantic requirements are over the objects’ communication histories, that is, a predicate over the method invocations and method returns [7, 8]. Here we ignore the semantic requirements, as it has no influence on the thesis.

Syntax: The syntax of an interface is

```
interface I [inherits InhList]
begin
  [with I'
  MSIGs]
end
```

The string *I* is the name of the interface. *InhList* is a comma-separated list of inherited interfaces. If the interface defines one or more methods, *I'* is the name of the cointerface of the following methods. *MSIGs* is a list of method signatures `op m[([in Param] [out Param])]`. *Param* is a comma separated list of `v:T`, where `v` is the variable name and `T` is the type of the variable.

The specified cointerface imposes a restriction on the kind of objects which are allowed to call the methods declared in this interface; the caller must support the cointerface or a subinterface of it. The cointerface can be `Any`; if so, objects of all types can call the method.

An interface may inherit more than one interface. Multiple inheritance imposes no problem with interfaces; if there are two method declarations with the same method name, parameters and cointerface, only one of the declarations is saved; i.e., the methods of the interface is the set union of all its inherited interfaces.

Example: Recall the “Bounded buffer” example presented in the introduction of this chapter. We have a producer and a consumer which both need access to the bounded buffer. For the environment (the producer and the consumer), it is not possible to see that the buffer is bounded,

only the operations are visible. As a first solution we let all types of objects access the methods, that is, the cointerface is Any:

```
interface Buffer
begin with Any
  op append(in d:Data)
  op remove(out d:Data)
end
```

To get a more fine-grained solution, we define two interfaces BufferA and BufferP. Only objects of type Producer is allowed to access the *append* method; the *cointerface* must be of interface Producer. Similarly, only consumers may remove items from the buffer:

```
interface BufferP
begin with Producer
  op append(in d:Data)
end
```

```
interface BufferC
begin with Consumer
  op remove(out d:Data)
end
```

Of course a buffer needs both methods; this is solved by defining an interface Buffer which inherits both the append and the remove methods from BufferP and BufferC:

```
interface Buffer inherits BufferP, BufferC
begin
end
```

The producer and the consumer are typed by two interfaces without methods:

```
interface Producer
begin
end
```

```
interface Consumer
begin
end
```

Part of the implementation of these interfaces will be discussed in the next sections. For a complete listing of all the code, see Appendix A.1.

2.1.2 Classes

Classes define the objects' persistent state variables and methods. Objects are typed by interfaces; the interface or interfaces an object supports are given by the class definition.

Syntax: The syntax of a class is

```
class C[(Param)] [inherits InhList] [implements ImpList]
begin
  [var VarList;]
  [[with I]
   MDECL]*
end
```

The string *C* uniquely identifies the class and *Param* is as defined for interfaces. *InhList* is a list of *C'*[(*EL*)], defining the inherited classes of *C*. The list of expressions *EL* is optional; it is evaluated when an object is created, and the result gives the actual parameters of the inherited class. Multiple inheritance at the class level is quite complex, therefore, the discussion of multiple inheritance is postponed to Chapter 5. The interfaces that a class C implements are given in *ImpList*.[2] *VarList* is a comma separated list of variable declarations v:T[=e] where v is the variable name, T the type of the variable and e an optional initial expression. The discussion of the cointerface given by *I* and the method declaration given in *MDECL* is postponed to Section 2.1.3, where we take a closer look at methods.

The objects' persistent variables consist of both the variables declared by **var** *VarList* and the parameters of the class.[3] In addition, all objects have a pseudo variable this which refers to the the object itself.

Example: The interface Buffer is implemented by a class BoundedBuffer:

```
class BoundedBuffer(max:int) implements Buffer
begin
  var buffer:List[Data]=empty, n:int=0;
  with Producer
  op append(in d:Data) ==  await n < max;
                           buffer := add(buffer,d);
                           n := n + 1
  with Consumer
  op remove(out d:Data) == await n > 0;
                           d := first(buffer);
                           buffer := rest(buffer);
                           n := n - 1
end
```

[2]Note that an object of class C is typed by the 'implements' clause of class C only. The class inherits only *code* from its superclasses; interfaces of superclasses are not inherited.

[3]Class parameters give rise to persistent object variables; this is similar to Simula [6]. It is different from many other object oriented languages; e.g., in Java, parameters in constructors are local variables [12].

The length of the buffer is given by a parameter max. The list buffer stores data of type Data; initially it is empty. The current number of elements in the buffer is n. The methods are discussed in the next section.

2.1.3 Methods

Methods define what objects do. A special method run defines the *active* behavior of the objects, whereas the methods which also are declared in the interfaces defines the *reactive* behavior of the object. Creol classes can define both internal and external methods. An *internal* method can only be called from inside the object. If a method has no preceding with clause, the method is internal. An *external* method can be called from other objects. External methods are defined after a with clause; if a method *m* is declared after with I, the calling object must be of a class which implements the cointerface I.[4] The with clause spans over the succeeding methods, until the next with clause or the end of the class.

Syntax: The syntax of a method is:

```
op m[([in Param] [out Param])] == [var VarList;]
                                  SList
```

The method name *m* must be unique within the class. Both in- and out-parameters are optional. *Param* and *VarList* are as defined for classes in Section 2.1.2. The statement list *SList* consists of statements separated by semicolons.

The local variables of methods consist of the in and out-parameters and variable declarations. The in-parameters of methods are read-only. In addition, methods have a pseudo variable *caller*, giving the object identifier of the caller of methods. The caller variable is typed by the cointerface.

Example: The *append* method of the class BoundedBuffer has Producer as cointerface, a single in-parameter but no out-parameters nor variable declarations:

```
with Producer
op append(in d:Data) == await n < max;
                        buffer := add(buffer,d);
                        n := n + 1
```

[4]An object may call its own external method if it implements the corresponding cointerface.

The in-parameter `d` is of type `Data`, the supertype of all types. The statement `await n < max` suspends the execution of the method until the expression `n < max` is true. The assignment statement `buffer := add(buffer,d)` adds `d` to the buffer. These and other statements are explained more carefully in the following sections.

2.1.4 Imperative and Functional Code

Creol is an imperative programming language, that is, statements are executed one after the other, and these statements change the state of the execution. In addition, Creol has a functional part; for example, we have

- for integer expressions: arithmetics (+, -, *, /) and comparison (=, <, >)
- for object identifiers: object identifiers' equality (=)
- for boolean expressions: logical constructs ($\wedge$, $\vee$, $\neg$)
- for lists: functions *first*(list), *rest*(list), *add*(list, item)

These functions do not alter the state. The semantics of the three first are as expected. For lists, *first*(list) returns the first element of a sequence list, *rest*(list) returns list without the first element and *add*(list, item) returns list with item added last.

Creol supports well-known statement constructs such as assignment and if-expressions. For a variable list V, an expression list E, a boolean expression *b*, statement lists S_1 and S_2, and a variable v, we have the following statements:

- Multiple assignment: V := E
- Conditional: if *b* then S_1 else S_2 fi
- Object creation: v := new *classname*(E)

When V := E is executed, the expressions in the expression list E are evaluated first, and then each variable in V is assigned the corresponding (evaluated) expression in E. The semantics of the conditional statement is as expected. The statement v := new *classname*(E) creates a new object of the class *classname*, and the object identifier of this object is assigned to v.

Note: We do not define any loop construct as in [2, 18, 21]. However, this is no problem as Creol supports recursive calls; all loop constructs can be expressed by recursive calls.[5]

Synchronous and Asynchronous Method Calls

Creol supports both synchronous and asynchronous method calls. For an object identifier o, a method name m, an expression list E, a variable list V and label t, these are possible external method calls:

- Synchronous call: o.m(E;V)
- Asynchronous call: t!o.m(E); ... ; t?(V)
- Method invocation: !o.m(E)

The first statement, o.m(E; V), is as a traditional synchronous method call: the method m of object o is called with the evaluated expression list E as in-parameters. Then the caller waits for the method return; when this arrives the return values are bound to the variables given in the list V.

The statement t!o.m(E) invokes the method m of object o. Instead of waiting for the answer, the execution may continue. The answer is fetched later on by the reply statement t?(V). The return values are assigned to the corresponding variables given in V. The label t identifies the call; the use of labels makes it possible to start more than one method call and still be able to get the answer for each.

The statement !o.m(E) is much like the second, except that no label is given and so no answer may be used. Hence, this statement invokes a method and never waits for the answer.

The object identifier o is omitted in corresponding *internal* method calls; i.e., m(x; y) for synchronous local calls, t!m(E); ... ; t?(V) for asynchronous calls and !m(E) for method invocations.

A method call is said to be *local* if it is an internal call or if it is an external call where the caller and callee are the same (a call to *self*). For asynchronous method calls t!o.m(E); ... ; t?(V) or t!m(E); ... ; t?(V), the return of the method invocation has not necessarily arrived when t?(V) is to be executed. If the method call is local, this method is executed. If the call is not local, the object blocks until the return arrives.

[5] The reason for omitting loop constructs from the language is that we want to be able to update the code at run-time [20]. By using recursive calls to simulate a loop, it is possible to update in the middle of the simulated loop.

Processor Release Points

In Creol, we define *processor release points* explicitly by await statements:

- Release statement: `await` g

The guard g is constructed inductively:

- A boolean expression b over local variables and attributes is a guard. This guard evaluates to true if the boolean expression evaluates to true.
- If t is a label, then $t?$ and $\neg t?$ are guards. The guard $t?$ evaluates to true if the reply identified by t has arrived. Similarly, $\neg t?$ evaluates to true if the reply has *not* arrived.
- An explicit release point is defined by the special guard `wait` which is false until the process has been suspended in front of this await statement.
- If g_1 and g_2 are guards, then $g_1 \wedge g_2$ is a guard. This guard evaluates to true if both g_1 and g_2 evaluate to true.

If the guard evaluates to false, the processor can be released so that other methods can be executed. To define this more precisely, we define a predicate *enabled* over statements and statement lists:

- The release statement `await` g is enabled if g evaluates to true.
- The release statement `await` g is not enabled if g evaluates to false.
- Apart from the release statement, all *atomic* statements are always enabled.
- A statement list is enabled if its first statement is enabled.

We call a statement *atomic* if it is not composed of other statements or statement lists (the *if* statement is also called atomic as the actual selection of the two statement lists S_1 and S_2 does not involve S_1 nor S_2). So far, all statement constructs presented is atomic, and thus always enabled.

Non-deterministic Choice and Merge

For statement lists S_1 and S_2, we have statements

- Non-deterministic choice: $(S_1 \Box S_2)$
- Merge: $(S_1 ||| S_2)$

The non-deterministic choice statement $(S_1 \Box S_2)$ selects one of S_1 and S_2, but in such a way that the first statement of the selected statement list is ready to execute. If neither is ready, the whole statement is not ready and therefore not executed.

The merge statement $(S_1 ||| S_2)$ is more complex. S_1 and S_2 are executed in an interleaved manner; the control can move between S_1 and S_2 at processor release points; i.e., control can move to the other statement list in front of an await statement where the guard evaluates to false.

The $\Box$ and $|||$ statements complicates the semantics of processor release points. Both are non-atomic statements as they are composed of statement lists. $(S_1 \Box S_2)$ is enabled if either S_1 or S_2 is enabled; the same applies to $(S_1 ||| S_2)$.

Note that both $\Box$ and $|||$ statements can be nested. This does not impose any problems, as both are associative and commutative, which implies that we have sets of statement lists. For instance, $((S_1 \Box (S_2 \Box S_3)) ||| (S_4 ||| S_5))$ can be thought of as $(S_1 \Box S_2 \Box S_3) ||| S_4 ||| S_5$. The enabledness of nested $\Box$ and/or $|||$ is no problem, as the enable predicate can be used recursively. Intuitively, this nesting creates a tree of statement lists, and the method execution can only be suspended if *none* of the branches is enabled.

Examples: The producer and the consumer are implemented by classes `Prod` and `Cons`, respectively. Both need a reference to the buffer; therefore, the buffer's object identifier is passed as argument to `Prod` and `Cons`. In this example, method `m` creates a buffer of length 10 and passes the object identifier to the producer and the consumer:

```
op m == var b:Buffer, p:Producer, c:Consumer;
        b := new BoundedBuffer(10);
        p := new Prod(b);
        c := new Cons(b)
```

The producer uses a loop method to produce the integers. A synchronous call is used to append the integers to the buffer; it is important to wait for the append method to return or else the producer may start an

arbitrary number of method calls before the consumer consumes anything. Internally, the producer calls its methods by method invocations without any return:

```
op run == !loop(0)
op loop(in i:int) == b.append(i); !loop(i+1)
```

The consumer is similar. The full buffer example is given in the appendix.

2.1.5 Example: The Santa Claus Problem

This is an example due to Trono [32], modified to illustrate Creol constructions and possibilities.

Santa Claus works at the north pole with his nine reindeer and his elves (at least three). Each Christmas, Santa and the reindeer deliver presents to children all over the world. This is hard work, and the rest of the year the reindeer go on holiday, while Santa sleeps most of the time. The elves produce the toys, and consult Santa if they have a problem. As Santa needs a lot of sleep, he helps groups of three elves to be more efficient. Hence, an elf must wait until at least two other elves need help. Summarized, Santa sleeps until

- all of his 9 reindeer are back from holidays, or
- at least three elves need to consult him with a problem.

It is more important to deliver the toys than to help the elves. Thus, if both all reindeer are back from holiday and at least three elves need help, Santa and the reindeer deliver toys. Before Santa and the reindeer can deliver toys, Santa must harness the reindeer. When they are back, Santa most unharness the reindeer. Similarly, when (at least) three elves want to talk to Santa, he opens the office door, let three elves in and close the door. After the consultation, he opens the door, let the three elves out and closes the door.

Communication: Interfaces

Santa, the reindeer and the elves are modeled by Creol objects. First we take a look at how these objects communicate and the Creol interfaces this communication yields.

A reindeer notifies Santa when it is back from holiday, and an elf notifies Santa when he needs help. These events will be modeled by method

calls backFromHoliday and haveProblem, respectively. As the reindeer-to-Santa communication and the elf-to-Santa communication differ, we use two interfaces SantaClausR and SantaClausE:

```
interface SantaClausR          interface SantaClausE
begin with ReinDeer            begin with Elf
  op backFromHoliday             op haveProblem
end                            end
```

Santa Claus' communication with the reindeer consists of harnessing and unharnessing them. The communication with the elves consists of telling them to enter and leave the office. Therefore we have two interfaces Reindeer and Elf:

```
interface Reindeer             interface Elf
begin with SantaClausR         begin with SantaClausE
  op harness                     op enterOffice
  op unharness                   op leaveOffice
end                            end
```

As Santa communicates with both reindeer and elves, we have an interface SantaClaus which inherits both SantaClausR and SantaClausE:

```
interface SantaClaus inherits SantaClausR, SantaClausE
begin
end
```

An object of type SantaClaus can thus communicate with both reindeer and elves.

Implementation: Classes

We implement Santa Claus by a class SantaClausC; see Figure 2.1. Santa Claus sleeps most of the time. He wakes up (in a mysterious way) when all nine reindeer are back from holiday, or if at least three elves want to talk to him. To control this, the SantaClausC class has two integers ct_rd and ct_elves, counting the number of waiting reindeer and elves, respectively. References to the waiting reindeer are stored in a list wait_rd, and references to harnessed reindeer are stored in a list harnessed_rd. Similarly, the class has lists wait_elves and inoffice_elves for the elves.

The implementation of the method backFromHoliday is straight forward: the counter is incremented and the reference to the reindeer, given by the pseudo variable caller, is added to the wait list. The method haveProblem is similar. The internal methods will be discussed later; first we take a look at the implementation of the reindeer and elves.

```
class SantaClausC implements SantaClaus
begin
  var ct_rd:nat=0, wait_rd:List[Reindeer]=empty,
      harnessed_rd:List[Reindeer]=empty,
      ct_elves:nat=0, wait_elves:List[Elf]=empty,
      inoffice_elves:List[Elf]=empty
  op run == !loop
  op loop ==
      (await ct_rd = 9; deliverToys() []
       await ct_elves >= 3 /\ ct_rd != 9; talkToElves());
      !loop
  op deliverToys == ...
  op talkToElves == ...

  with Reindeer
  op backFromHoliday ==
      ct_rd := ct_rd + 1;
      wait_rd := add(wait_rd,caller)
  with Elf
  op haveProblem ==
      ct_elves := ct_elves + 1;
      wait_elves := add(wait_elves,caller)
end
```

Figure 2.1: The Santa Claus class

The reindeer start to go on holiday. When they are back, they notify Santa by sending a message backFromHoliday, and then they wait to get harnessed. When they are harnessed, they deliver toys together with Santa. When they are finished, Santa unharnesses them and they go on holiday again. The class ReindeerC is given in Figure 2.2. The elves have a similar behavior; the ElfC class is given in Figure 2.3.

The *active behavior* of the Santa Claus class consists of delivering toys and to talk to elves. This is reflected by a method loop where Santa waits for an activating condition to be true and then do the appropriate. See Figure 2.1.

Before Santa can deliver toys, he must harness the reindeer. This is done by invoking the method harness for all the reindeer objects (in the wait_rd list). For efficiency reasons, this method is invoked for *all* the reindeer before waiting for the answer; this way the method invocations may be executed in parallel. After reindeer are harnessed, Santa delivers the toys (together with the reindeer). When he is finished, he unharnesses the reindeer by invoking the unharness method for all reindeer. The method deliverToys is given in Figure 2.4.

Remark: In real life Santa and the reindeer would be finished delivering toys at the same time. Here, this is not the case. Santa unharnesses the

```
1  class ReindeerC(sc:SantaClausR) implements Reindeer
   begin
     op run == !holiday
     op holiday == <<Go on holiday>>; !sc.backFromHoliday
5    op deliverToys == <<Deliver Toys>>

     with SantaClausR
     op harness == !deliverToys
     op unharness == !holiday
10 end
```

Figure 2.2: The Reindeer class.

```
1  class ElfC(sc:SantaClausE) implements Elf
   begin
     op run == !work
     op work == <<Do work>>; !sc.haveProblem
5    op talkToSanta == <<Talk to Santa>>

     with SantaClausE
     op showIn == <<Go into Santa's office>>; !talkToSanta
     op showOut == <<Leave Santa's office>>; !work
10 end
```

Figure 2.3: The Elf class.

```
1    op deliverToys ==
       var t1:Label,..,t9:Label;
       ct_rd := 0;

5      t1!first(wait_rd).harness;
       harnessed_rd := add(harnessed_rd, first(wait_rd));
       wait_rd := rest(wait_rd);
       ...
       t9!first(wait_rd).harness;
10     harnessed_rd := add(harnessed_rd, first(wait_rd));
       wait_rd := rest(wait_rd);

       await t1? /\ t2? /\ ... /\ t9?;

15     <<Pick up and deliver toys>>;

       !first(harnessed_rd).unharness;
       harnessed_rd := rest(harnessed_rd);
       ...
20     !first(harnessed_rd).unharness;
       harnessed_rd := rest(harnessed_rd)
```

Figure 2.4: The method deliverToys in class SantaClausC.

```
1  class LeaderElfC(sc:SantaClausC, elves:List[Elf])
     inherits ElfC(sc) implements Elf
   begin
     op run == !run@ElfC
5    op work ==
        (<<Lead the elves>> ||| <<Make toys>>);
        !sc.haveProblem
   end
```

Figure 2.5: The Leader Elf class.

reindeer as soon as they have finished to deliver toys, and does not wait for the unharness method to complete (he is eager to go back to sleep).

The talkToElves method is similar, except that Santa must wait for the elves to leave the office before closing the door, in the same manner as he waits for the reindeer to get harnessed before he deliver toys. The full example with the talkToElves method and a starter class is given in Appendix A.2.

Defining a Leader Elf using Inheritance

The use of interfaces makes it possible to have different kinds of elves. For example, it is possible to define a leader elf which has other tasks than the other elves. It is natural that a leader has control over the workers; therefore, the leader has a list of the worker elves. For Santa, all elves are equal. Thus, the communication with Santa is equal to the other elves' communication with Santa. The leader elf interleaves between leading the working elves and to make toys as the other elves.

The leader elf is implemented by a class LeaderElfC. It inherits the ElfC class, and redefines the work method. As the run method of inherited classes (superclasses) is not started automatically, LeaderElfC has a method run which simply calls the run method in ElfC.[6] The LeaderElfC class is given in Figure 2.5. Note that the reference to Santa Claus is passed forward to the inherited ElfC class.

2.2 Java Concurrency

The concurrency model of the Java programming language is based on multiple execution threads, in contrast to Creol's concurrency model

[6] A method m of an inherited class C can be invoked explicit by m@C; this will be discussed further in Chapter 5.

which is based on active objects. In this section we give a brief summary of the Java thread model and some Java threading tools which we use in the implementation part of this thesis. The reader is assumed to be familiar with concurrent programming, that is, have a basic knowledge about threads and synchronization primitives such as, e.g., semaphores, locks, and condition variables. Andrews [1] gives an excellent overview of this area. The reader is also assumed to know the basics of Java: classes, objects, interfaces, etc., and how sequential programs are written in Java; Eckel [9] gives a good introduction to the basics of Java.

Java is used on a wide range of platforms; e.g., desktop computers, servers, mobile phones, and smart cards. Therefore, there is a number of editions of Java. Here, we only consider the standard edition. In the standard edition, it is possible to write concurrent programs by using multiple threads. A thread is a light-weight process with its own execution thread; a program can consist of multiple threads which share the same memory. In Section 2.2.1 we present how to create and start Java threads, and discuss some properties about the thread scheduling.

A thread executes code independent of activity in other threads, but the code in the different threads operate on values and objects residing in a shared main memory. The shared memory is not automatically protected, in principle several threads may access the same address in the shared memory at the same time. It is the programmers responsibility to prevent race conditions. Because of this, some consider Java's parallelism to be insecure [3]. However, safe programs can be written by careful synchronization. Early versions of Java came with few tools for thread synchronization, and advanced features were missing. However, Java 2 Standard Edition Version 5.0 (J2SE 5.0) provides a utility package which offers the tools we need for our implementation; in Section 2.2.2 we present some of the tools included in this version.

2.2.1 Java Threads

Java is an imperative object-oriented language which supports thread concurrency. The execution threads and the objects are separated; a thread executes code that operates on objects in the shared memory, and multiple threads' code can operate on the same object. However, there is a connection between objects and threads, as a new thread is created by making an instance (an object) of the `Thread` class. This object stores thread-specific data and has thread-specific methods.

```
class Producer implements Runnable {
    Buffer buffer;
    public Producer(Buffer b) {
        buffer = b;
    }

    public void run() {
        int i = 0;
        while(true) {
            buffer.append(new Integer(i));
            i++;
        }
    }
}
```

Figure 2.6: A Producer class in Java.

Thread creation

Threads in Java may be created in one of the two following ways:

- Create a subclass of Thread, say MyThread. Override the run() method of Thread, create an instance of MyThread and call the start() method of this instance. Example: new MyThread().start()
- Create a class which implements the Runnable interface, say MyRunnable. MyRunnable must implement a method run(). Create an instance of MyRunnable and give this instance as a parameter to the Thread constructor: new Thread(new MyRunnable()).start()

The start() method in the Thread class first initializes the thread and then it calls the run() method. In the first approach, the run() method is overridden by the programmer and the program-specific code is executed directly. In the second approach, the Thread.run() method calls the MyRunnable.run() method.

Example: A producer

Assume that we have a class or an interface Buffer which offers the *append* and *remove* operations defined at the beginning of this chapter by methods append() and remove(), respectively. We create a producer which produces the natural numbers and inserts each of them in the buffer; see Figure 2.6. The class Producer implements the Runnable interface and has a method run(). We create an instance of this class and an execution thread for it, and start the execution thread:

```
Producer p = new Producer(buffer);
Thread t = new Thread(p);
t.start();
```

The start() method in the Thread class calls the run() method in the Thread class, which again calls the run() method in the Producer class.

Thread scheduling

The Java Virtual Machine Specification [22] does not specify in detail how implementations of a Java virtual machine (JVM) should schedule threads; the scheduling is implementation-specific. In particular, the specification does not require that the virtual machine supports time-slicing. On JVMs without time-slicing, some threads may never get to run if there are more threads than processors. In this case we have starvation. The programmer must program carefully so that all threads are given execution time. On JVMs with time-slicing on the other hand, all threads are given execution time, and the programmer does not have to worry about starvation. Sun's latest versions of its JVM implementation for desktop computers support time-slicing.

2.2.2 Data Synchronization

Java threads communicate via shared memory. The data in Java are objects. By default, all objects have a single lock, called the object's *monitor*; this lock is used by *synchronized methods* and *synchronized blocks*. A method is declared to be synchronized by prefixing the method declaration with the key word synchronized; for example

```
synchronized void inc() {
  counter = counter + 1;
}
```

A synchronized method acquires the object's lock at the beginning of the method execution and releases it at the end; if all methods of a class are synchronized and the class attributes are only accessed by these methods, race conditions are avoided for objects of this class.

Each object has a single condition variable which is bound to the object's lock. Inside a synchronized method (or block) a thread can wait for a signal by the call wait(); this call releases the lock and the thread is suspended. A thread can signal one or all waiting threads by the calls

notify() or notifyAll(), respectively. To illustrate this, we define a decrease operation which decreases a counter when it is positive, and an increase operation which first increases the counter and then signals that the counter is different from 0:

```
synchronized void dec() {
  while(counter == 0) wait();
  counter = counter - 1;
}
synchronized void inc() {
  counter = counter + 1;
  notify();
}
```

Synchronized blocks are similar to synchronized methods and are not discussed here.

There are some limitations to the synchronization primitives offered by the Object class and the synchronized construct. Each thread has only one condition variable; for some problems this does not suffice as different threads might wait for different conditions to become true. Furthermore, the Java specification does not give any guarantees about fairness; if multiple threads want to access synchronized methods in the same object, there is a possibility of starvation, that is, some of the threads may never get to execute.

The concurrent package offers additional synchronization tools such as locks, semaphores and condition variables. Furthermore, this package has data structures which are thread-safe, that is, are protected against race-conditions; we will not discuss these data structures.

A simple lock is offered by the interface Lock and implemented by a class ReentrantLock. The lock is acquired by calling its method lock() and released by calling unlock(). The lock class has a method newCondition() which creates a new condition variable which is bound to the lock. The condition variables provide several methods, including the methods await() and signal(). When a thread calls await() on a condition variable of a lock, the lock is released and the thread is suspended until awoken by a signal on the same condition variable. When a thread calls signal() the thread that has waited longest is awoken, that is, the await() method of that thread will return.

Example: The Bounded Buffer

To illustrate, we implement the previously mentioned interface Buffer as a class BoundedBuffer, see Figure 2.7. Object references are stored in an

```
interface Buffer {
    public void append(Object x);
    public Object remove();
}

class BoundedBuffer implements Buffer {
    private Lock lock;
    private Condition notFull;
    private Condition notEmpty;
    private Object[] items;
    private int putptr, takeptr, count;

    public BoundedBuffer(int n) {
        lock = new ReentrantLock(true);
        notFull = lock.newCondition();
        notEmpty = lock.newCondition();
        items = new Object[n];
        putptr = 0;
        takeptr = 0;
        count = 0;
    }

    public void append(Object x) {
        lock.lock();
        while (count == items.length) {
            try { notFull.await(); }
            catch (InterruptedException ie) { }
        }
        items[putptr] = x;
        putptr = putptr + 1;
        if (putptr == items.length) putptr = 0;
        count = count + 1;
        notEmpty.signal();
        lock.unlock();
    }

    public Object remove() {
        lock.lock();
        while (count == 0) {
            try { notEmpty.await(); }
            catch (InterruptedException ie) { }
        }
        Object x = items[takeptr];
        takeptr = takeptr + 1;
        if (takeptr == items.length) takeptr = 0;
        count = count - 1;
        notFull.signal();
        lock.unlock();
        return x;
    }
}
```

Figure 2.7: The Bounded Buffer using fair locks.

array named items. The integers putptr and takeptr are used as indexes pointing to where in the items array object references are inserted and removed, respectively. The integer count stores the number of elements in the items array. To protect the buffer from race conditions, we use a fair lock lock and two condition variables notFull and notEmpty. The attributes are initialized in the constructor. Note that true is given as argument to the Lock constructor to specify that we want a fair lock.

The append() and remove() methods are very similar to each other; hence, we only consider the append() method. First, the lock is acquired. Then, if the buffer is full, the thread waits for the buffer to be not full by the call notFull.await(). Suspended threads may be interrupted, that is, an InterruptedException may be thrown; thus, the await() call is inside a try-catch block and we catch interrupt exceptions. There is no guarantee that the condition actually holds when await() returns; therefore, the condition must be rechecked (hence the while-loop). When the while-loop ends, the buffer is not full and the object can be inserted in the buffer (lines 29-33). Finally the lock is released.

Read/write locks

The concurrent package has a read-write lock, which is useful when we want multiple readers to access some shared data at the same time. It can be used to solve the traditional readers/writers problem. The package has an interface ReadWriteLock; this interface is implemented by a class ReentrantReadWriteLock. We create a fair read-write lock:

```
ReadWriteLock rwl = new ReentrantReadWriteLock(true);
```

The lock consists of a read-lock and a write-lock. When we want to acquire the read-lock, the read-lock is first fetched by the method readLock() and then acquired by the method lock(). It is released by the method unlock(). The write-lock is used in a similar way. We use the read-write lock as follows:

```
rwl.readLock().lock();
/* shared access, only read */
rwl.readLock().unlock();

rwl.writeLock().lock();
/* exclusive access, ok to write */
rwl.writeLock().unlock();
```

The write-lock has a condition variable, which works as for other locks. A thread which has taken the write-lock, can grab the read-lock, and hence it can downgrade from a write-lock to a read-lock by acquiring the read-lock and then releasing the write-lock. The other way around is not possible, because other threads may also have acquired the read-lock.

Chapter 3

A Computational Model for Creol

This chapter discusses a computational model for Creol, called Creol Computational Model (CCM). The reader is assumed to be somewhat familiar with Creol, to the same extent as given in Section 2.1.

We start by motivating why we need a computational model and what purposes it will serve in Section 3.1. Section 3.2 defines the model formally. Section 3.3 presents the various components of the CCM. Section 3.4 discusses the execution of Creol programs in the model.

3.1 Model Objectives

The design and especially the implementation of a virtual machine is complex, and details and implementation issues may overshadow the main structure and behavior of the virtual machine. This is why we create a computational model, which will abstract away inessential details and thus be much easier to work with.

We will define the model formally, because this will make the model unambiguous and because a formal notation helps us define the computations of the model in a concise manner. Working with a formal model gives some important benefits:

- It will be comparatively short and concise. Therefore, it should be easy to read and understand.
- It can be used as the high level model or an specification when designing and implementing a virtual machine.

- The Creol language can be defined formally in the model.
- It is possible to reason about computations in the model.

The semantics of the Creol language is given in rewriting logic and forms the basis of an implementation of a Creol interpreter in Maude [2]. Our implementation of a virtual machine must observe the semantics of Creol. This might be easier to accomplish if our model has the same structure as the RL semantics. However, the structure used in the RL semantics is not appropriate as the basis for an implementation in an imperative and object oriented language. Our model will be quite different. The differences may be interesting for the reader, but are omitted to keep the description of the model compact.

3.2 The Model

A Creol computational model

(CVM names, Initial objects, Class definition set)

intended to represent a Creol program, is given by the following components:

CVM names is a set of Strings. Each String serves the purpose of identifying a Creol Virtual Machine (CVM), defined in Section 3.3.

Initial objects is a set of tuples **(CVM name, Class name, parameters)**. Each tuple specifies an object to be created at initialization. More details in Section 3.4.

Class definition set represents the Creol program code. It is defined in Section 3.3.3.

The *execution* of programs is modeled by *computations* of the model; this is discussed in Section 3.4. We define computations as sequences of *states*. How these states are represented is defined in Section 3.3.

3.3 Structure and States

This section describes the structure of a Creol Virtual Machine and the formal representation of states. The structure described will serve as a basis for an implementation of a CVM (Chapter 4), and is the main focus of this section. The formal representation of states is for the convenience of describing computations in a concise and precise manner;

computations are discussed in Section 3.4. Therefore, for each part of the model we give an informal description to motivate the concepts and then give the formal representation of the state.

Creol is a programming language designed to create distributed programs. Creol objects may be distributed on different nodes[1]. On each of these nodes there is a *Creol Virtual Machine* (CVM). The CVMs communicate with each other. Formally, a *state* in a CCM computation is a set of such machines, represented as

$$state = \{ \text{CVM}_1, \text{CVM}_2, \ldots, \text{CVM}_n \}$$

Further, we will cut the state into pieces; we will describe, i.e., *the state of a CVM* and *the state of an object* (in a CVM).

3.3.1 The CVM

Creol is an object oriented language; objects are fundamental in a Creol program. Conceptually, an active object has its own thread of control. In our model, we have *CVM objects*[2], each with its own thread of control. They live inside the CVM. A CVM object reflects a Creol object. Communication between objects is modeled by message passing. We will not focus on how this is done, as we consider this to be an implementation issue. Objects put messages in their *out-queue* and receive messages in their *in-queue*; messages are transported between objects by some underlying (low-level) mechanism.

We focuses on modeling execution of Creol programs on one node (virtual machine). Yet, Creol objects may be distributed over different nodes, so communication between nodes must be taken into consideration. Therefore, the CVM has queues for communication with other CVMs. Some features are common to all objects in the CVM; i.e., class definitions and the creation of new objects. Therefore, each CVM has a *central* which stores these common structures and which offers services to the objects. The communication between the objects and the Central is modeled by message passing; thus the Central has an *in-queue* and an *out-queue* like the objects. See Figure 3.1.

The *state of a CVM* is represented as a tuple

$$\text{CVM} = \langle \text{Id}, \text{In-queue}, \text{Out-queue}, \text{Central}, \text{Objects} \rangle$$

[1]The nodes can be different computers in a network, but there is nothing wrong in having more than one node on the same computer.

[2]We use *CVM* in front of object to distinguish these objects from objects in the Creol language (when this is necessary).

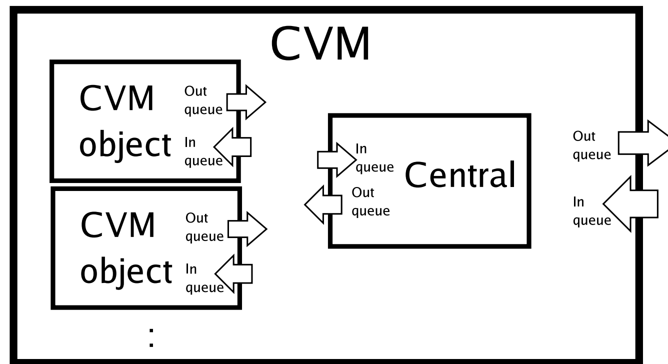

Figure 3.1: The CVM has objects, a central, an in-queue and an out-queue.

where ID is a String identifying this particular CVM and the *queues* are ordered[3] multisets of messages; i.e., $\{ \texttt{msg}_1, \texttt{msg}_2, \ldots \}$. CENTRAL is specified in Section 3.3.2. OBJECTS is a set of CVM objects; the structure and formal representation of an object are specified in Section 3.3.4.

3.3.2 The Central

The Central is mostly motivated by the main purpose of the model; it will be used as a high-level model for the implementation given in Chapter 4. The idea is that the Central should serve the objects in tasks like creating new objects and method lookup; these are *services* offered by the Central.[4] The services "*Create object*" and "*Get method definition*" are explained in Section 3.4.2.

Communication between the Central ad the CVM objects is modeled by message passing, in the same manner as between objects; therefore, the Central has an *in-queue* and an *out-queue*. When an object wants some service offered by the Central, it sends a message to the Central which then performs the requested service and sends an answer message back to the object.

Formally, the *state of a central* is represented as a tuple

$$\textsc{Central} = \langle\ \textsc{In-queue}, \textsc{Out-queue}, \textsc{Class definition set}\ \rangle$$

The queues are ordered sets of messages. The IN-QUEUE may look like

[3]The type of ordering in the queues is unspecified, but a first-in, first-out policy is probably a good choice for implementation.

[4]The services offered may be extended; i.e., *Update class*.

{ `newObj(...)`, `newObj(...)`, `getMethodDef(...)`, ... }

The message syntax is explained in detail in Section 3.4.2. The component CLASS DEFINITION SET is explained in the next section.

3.3.3 Class Definitions

Class definitions are needed by the Central in order to create objects and give objects method definitions. See Figure 3.1. Formally, the CLASS DEFINITION SET is a set of tuples, where each tuple specifies a Creol class:

CLASS = ⟨ CLASS NAME, PARAMLIST,
INHLIST, ATTRIBUTES, METHODS ⟩

where CLASS NAME is a String identifying the class, PARAMLIST a list of Strings specifying the formal parameters of this class, INHLIST is a list of pairs <*classname*,*exprList*>, where *classname* identifies an inherited class and *exprList* is a list of expressions over variables from PARAMLIST. ATTRIBUTES is a set of pairs <*var*,*val*> where *var* is a String identifying a variable with initial value *val*. METHODS is a set of tuples

METHOD = ⟨ METHOD NAME, SIGNATURE, COINTERFACE,
PARAMETER LIST, RETURN VARIABLES, CODE ⟩

The String METHOD NAME uniquely identifies the method (within the class). SIGNATURE is a list (in_1, in_2, ..., in_n; out_1, out_2, ..., out_m), where in_i and out_i are the types of the in and out parameters, respectively. COINTERFACE is a String, and PARAMETER LIST and RETURN VARIABLES are lists of Strings giving the names of the parameters and return parameters, respectively. CODE is a statement list giving the code of this method.

Translation from Creol class definitions to CCM representation

We assume strong static typing [21]. The type of an expression is determined at compile time; therefore, we do not need to store the type of variables. Nevertheless, as Creol supports virtual binding of method calls, the signature of method calls must be preserved. (Method look-up with multiple inheritance is explored in Chapter 5.) The CCM representation of the class name, parameter list and attributes are almost the same as for Creol. We illustrate by an example. Consider the following Creol class:

```
class A(v_1:T_1, v_2:T_2, ..., v_n:T_n) inherits inhList
begin
    var u_1:T_1 = expr_1, u_2:T_2 = expr_2, ..., u_m:T_m = expr_m;
    <methods>
end
```

The CCM representation of this class is

$$\langle \text{ A}, (v_1, v_2, \ldots, v_n), \textsc{InhList}, (\text{<}u_1,expr_1\text{>}, \text{<}u_2,expr_2\text{>}, \ldots, \text{<}u_m,expr_m\text{>}), \textsc{Methods} \rangle$$

The inheritance list is not interesting for now, as we do not consider inheritance; however, we choose to store it in the class definitions to make it easier to extend the model to support multiple inheritance (Chapter 5). METHODS is a set $\{ \text{M}_1, \text{M}_2, \ldots, \text{M}_n \}$ of methods M_i. Consider a Creol method

```
with Co
op m( in: v_1:T_1^in, v_2:T_2^in, ..., v_n:T_n^in
      out: u_1:T_1^out, u_2:T_2^out, ..., u_m:T_m^out) == Code
```

The CCM representation of this method is

$$\langle \text{ m}, (T_1^{in}, T_2^{in}, \ldots, T_n^{in}; T_1^{out}, T_2^{out}, \ldots, T_m^{out}), \text{Co}, (v_1, v_2, \ldots, v_n), (u_1, u_2, \ldots, u_n), \textsc{Code} \rangle$$

$(T_1^{in}, T_2^{in}, \ldots, T_n^{in}; T_1^{out}, T_2^{out}, \ldots, T_m^{out})$ is the signature, $(v_1, v_2, \ldots, v_n)$ is the in parameters and $(u_1, u_2, \ldots, u_n)$ is the out parameters. *Code* is a list var *vdecl*; s_1; s_2; s_3; ..., where var *vdecl*; consists of local variable declarations and s_i are statements. (var *vdecl*; may be omitted.) The translation from *Code* to CODE is not as straight forward as the rest of the components. We will make some changes to the code which makes it more convenient for computation and implementation. First, we make some assumptions:

- The String wait is not used as a variable name in *Code*.
- All variable names in *Code* are different from those used in the class.
- All variable declarations have initial values (possibly `null`).

It is a trivial matter to change the code to accomplish this (without changing the semantics of the program), if these assumptions are not already met.

The Creol statement await wait (or more generally await wait $\wedge$ g for a guard g) is an explicit processor release point. In a process, this statement evaluates to *false* until the process has been suspended. To accomplish this, we have a Boolean system variable wait in the process' local variables. When a process is suspended, wait is set to true, such that await wait evaluates to *true* next time. When a statement (different from $\Box$ and $|||$) in the process is executed, wait is set to false; hence, next time await wait evaluates to *false* and the process is suspended. The $\Box$ and $|||$ statements do not update the wait variable, as this would cause processes to be suspended unintentionally (see Section 3.4.6).

We make the following changes to the statements:

New object: The creation of a new object is modeled by sending a `newObj` message to the Central. A reference to this object is sent back to the object by a `newObjId` message. The statement v := new *classname*(E); is changed to the statements new *classname*(E); waitObjId(v); so that executing a statement is still one atomic step.

Merge operator: To execute a statement list s_1, s_2, s_3, ..., we execute the statements s_i one by one, and after each statement s_i is executed, control is set to s_{i+1}. In our model, we just remove s_i from the list. In the implementation, we use pointers to the code to avoid code duplication.

In the Creol statement list $(S_1 \; ||| \; S_2)$; S_3, S_1 and S_2 are executed in an interleaved manner. In the Maude interpreter, this is solved by manipulating the code; that is, changing $(S_1 \; ||| \; S_2)$ to $(S_2 \; ||| \; S_1)$ at processor release points and removing the first statement of S_1 or S_2 when executed. This is not very easy to do by pointer manipulations; therefore, our computational model uses a different approach than [18].

We introduce a statement joinMerge(v) and changes $(S_1 \; ||| \; S_2)$; S_3 to $(S_1 \; ||| \; S_2)$; joinMerge(v); S_3. The idea is that when the first statement of $(S_1 \; ||| \; S_2)$; joinMerge(v); S_3 is executed, we split into two statement lists S_1; joinMerge(v); S_3 and S_2; joinMerge(v); S_3. Both will be executed in an interleaved manner, but the first to reach joinMerge(v) terminates whereas the last one continues with S_3. The variable v is not used elsewhere in this method or in the class. A variable declaration v:Boolean = false; is added in *vdecl*. The computation of the merge statement and joinMerge(v) is defined formally in Section 3.4.6.

Remark: In case of nested merge statements, a *joinMerge*-statement is added for each $|||$. For example, the statement list $((S_1 \; ||| \; S_2) \; ||| \; S_3)$ is changed to $((S_1 \; ||| \; S_2)$; joinMerge(v_1) $||| \; S_3)$; joinMerge(v_2).

Return statement: After a process has terminated, a completion message must be send to the caller. The process has terminated when there are no more statements to execute; this is possible to figure out, but

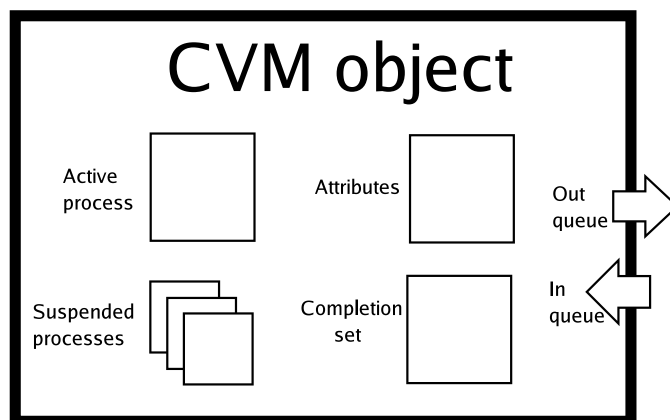

Figure 3.2: The CVM object has an active process, a set of suspended processes, attributes, a completion set, an in-queue, and an out-queue.

instead we introduce a special statement return *variable list*. *variable list* is a list of Strings giving the names of the variables which will be returned. Our previous example gives return (u_1, u_2, ..., u_n). This is inserted after the last statement of *Code*. The execution of the return statement is explained in Section 3.4.6.

In addition, we make some changes to have fewer statement types to consider:

- await t?(V) is translated to its equivalent await t?; t?(V)
- p(E;V) is translated to t!p(E); t?(V) for a fresh variable t, and var t:Label is added in front of *Code*.
- await p(E;V) is translated to t!p(E); await t?; t?(V) for a fresh variable t, and t:Label; is added to *vdecl*.
- !m(E) and t!m(E) are transformed into !this.m(E) and t!this.m(E), respectively.

3.3.4 The Object

Creol objects are modeled by *CVM objects*, see Figure 3.2. These objects live inside a CVM — the CVM is its environment. An object does not know much about its environment, except that it can communicate with it by sending and receiving messages through its queues. It also knows which services the central offers, and how to invoke these services. To

make a method call, an object sends a *method invocation message*. If the call is external (another object), the message is put on the object's out-queue. If it is internal, the message is put on the object's own in-queue. In the same manner, the return of method calls result in *method completion messages*.

Inside an object, we find components visible only to this particular object. The most important ones are the attributes and the processes, as these components more or less define the state and behavior of the object. The *attributes* are instantiated local copies of the class parameters and attributes. They are pairs <id, val>, where id is a String identifying the variable, and val is the value of the variable (or null if the variable is uninitialized). As Creol is a statically typed language, no type information is necessary.[5] Creol objects are active; they execute processes. The *active process* is the process currently executing in the object. As Creol objects create a new process for each method call, and these processes may have processor release points, the CVM objects also have *suspended processes*. This is a queue of processes; the ordering is unspecified (implementation issue). Each object has a *completion set*, which serves as a buffer where processes look for the return of their method calls. The answer to a method invocation, external or local, is given by a completion message. This message is transformed into a pair <label, val> and put into the completion set. In this pair, label identifies the method call and val is the list of return values.[6] As objects communicate, each object needs a unique *id*. The *class* the object is an instance of, is needed to get method definitions.

The *state of the object* is given by a tuple

OBJECT = ⟨ ID, CLASS, IN-QUEUE, OUT-QUEUE, ATTRIBUTES,
COMPLETION SET, ACTIVE PROCESS,
SUSPENDED PROCESSES, STATUS ⟩

ID is a unique identifier for the object. CLASS is a String identifying which class this object is an instance of. The queues are ordered sets of messages (first in first out), ATTRIBUTES is a set of pairs <id, val> such

[5] The only type information needed is the signature and the cointerface of method calls. This is static type information determined during type checking (compilation) and part of the method call at run time. Note however that we have not included the signature and cointerface yet, it will be included when inheritance is taken into account (Chapter 5).

[6] The completion messages could just stay in the in-queue of the object, but there are a number of reasons for using a completion set: 1. The queue is shared and should not be accessed more than necessary. 2. The queue is ordered in some way; this is not necessary for completion messages. 3. Keeping the messages in the queue implies that they need to be processed many times.

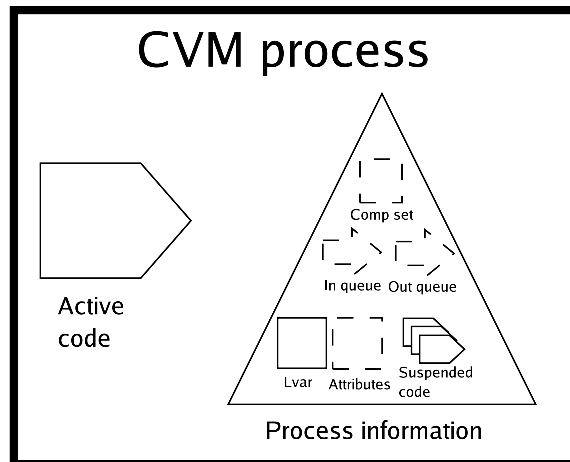

Figure 3.3: The process has active code, suspended code and local variables. The attributes, completion set and queues of the object are modified by the active code and therefore shown in the figure as stippled boxes/arrows. Components modified by the active code are grouped as "Process information".

that there is no two pairs with the same id, COMPLETION SET is a set of pairs <label, returnlist>, ACTIVE PROCESS is a process and SUSPENDED PROCESSES is an ordered set of processes (what sort of ordering is unspecified). The processes are described in the next section. The STATUS is a flag describing the current activity in the object. It is used to get a more deterministic behavior in the model, making the model more suited for implementation. This is explained Section 3.4.3.

3.3.5 The Process

In Creol, method calls are (conceptually) executed by *processes*; when an object receives a method invocation message, it creates a new process which executes the method and sends the return of the method invocation.

A process can be represented by its local variables and its code. The local variables are initialized copies of the parameters of the method and of the variable declarations in the method's code. Due to Creol's merge operator |||, which interleaves the execution of statement lists, the process may execute more than one statement list. In the Creol code (S_1 ||| S_2); S_3 the execution can change between statement lists S_1 and S_2 at

CCM	(CVM names, Initial objects, Class definition set)
CCM state	{ CVM_1, CVM_2, ..., CVM_n }
CVM	⟨ ID, IN-QUEUE, OUT-QUEUE, CENTRAL, OBJECTS ⟩
Central	⟨ IN-QUEUE, OUT-QUEUE, CLASS DEFINITION SET ⟩
Class	⟨ NAME, PARAMLIST, INHLIST, ATT, METHODS ⟩
Method	⟨ NAME, SIG, CO, PARAM, RETVAR, CODE ⟩
Object	⟨ ID, CLASS, IN-QUEUE, OUT-QUEUE, ATT, COMPSET, ACTIVEPROCESS, SUSPENDEDPROCESSES, STATUS ⟩
Process	⟨ LOCALVARIABLES, ACTIVECODE, SUSPENDEDCODE, ID, DYNAMICLINK ⟩

Table 3.1: The state representation of various structures in CCM.

processor release points[7]. We split these statement lists into the *active code* and the *suspended code*, where the active code is the statement list currently executing, and the suspended code are suspended statement lists. As the merge operator is commutative and associative, suspended code is represented as a *set* of code.

The *state of a process* is given by a tuple

PROCESS = ⟨ LOCAL VARIABLES, ACTIVE CODE,
SUSPENDED CODE, ID, DYNAMICLINK ⟩

The LOCAL VARIABLES is a set of pairs <id, val>, reflecting the parameters and the variable declarations in the method this process is an instance of. The String id identifies the name of a variable and val the current value of the variable; this is equivalent to the attributes in the object. The ACTIVE CODE is a list of Creol statements. The SUSPENDED CODE is an unordered multiset of statement lists.

A process may block waiting for the return of a local method call. To avoid deadlock, such local calls are allowed to be executed, and the current process is suspended. When the call has completed, the suspended process is reactivated. In order to 'return' to the caller, each process is identified by ID. When a local call is activated, DYNAMICLINK (of the process representing the call) is set to the identifier ID of the caller, and this dynamic link is used to reactivate the caller. See Section 3.4.5 for details.

Table 3.1 gives an overview of the representation of the structures which are discussed so far in this chapter.

[7] See Section 2.1 for the definition of processor release points.

3.4 Computations

The previous section discussed how Creol programs may be represented in a Creol computational model and how the states of an execution of a program are represented. In this section we discuss the *computations* in the model, representing the execution of Creol programs.

Formally, a computation of a CCM is a sequence $s_0, s_1, s_2, \ldots$ of states, where s_0 is the initial state (soon to be defined), and each s_i $(i > 0)$ follows from $s_{(i-1)}$ by using the rules we will define in this section.

First we will describe the *initialization* of the model; for a CCM, we define the initial state s_0. Then we will describe each rule. For each rule, we describe the *precondition* and the *postcondition* of the rule, which give the conditions for when the rule may be applied and the effect of applying it, respectively. The rules are thus specified in an implicit manner [25]. This is very similar to the implicit specification of operations in the Vienna Development Method (VDM) [29]. The use of a precondition and a postcondition is best explained by an example. Assume the state s_i of the CCM is

$s_i = \{ \ldots , \text{CVM}_i, \ldots \}$
$\text{CVM}_i = \langle$ Id, In-queue, Out-queue, Central, Objects $\rangle$
Objects $= \{ \ldots , \text{Object}_i, \ldots \}$
$\text{Object}_i = \langle$ id, class, In-queue, Out-queue, Attributes,
Completion set, Active process,
Suspended processes, Status $\rangle$

Say, for the sake of example, that an object may go from the state "passive" to "active" whenever the in-queue is not empty. The precondition would be:

Status = "passive" $\wedge$ In-queue != { }

The other components of the object and other objects are not taken into consideration, as there are no constraints of their values. The postcondition would be

Status = "active"

Components not affected by this rule are not mentioned. Formally, this is just a short notation for the precondition

state = { CVM$_i$, *RESTOFCVMS* } ∧
CVM$_i$ = ⟨ *ID*, *IN-QUEUE*, *OUT-QUEUE*, *CENTRAL*,
{ OBJECT$_i$, *RESTOFOBJECTS* } ⟩ ∧
...
STATUS = “passive” ∧
IN-QUEUE != { }

and the postcondition

state = { CVM$_i$’, *RESTOFCVMS* } ∧
CVM$_i$’ = ⟨ *ID*, *IN-QUEUE*, *OUT-QUEUE*, *CENTRAL*,
{ OBJECT$_i$’, *RESTOFOBJECTS* } ⟩ ∧
...
STATUS’ = “active”

Here, *ITALIC CAPS* are used to match an arbitrary value in the precondition and then used to preserve this in the postcondition. SMALL CAPS are used to refer to components in the precondition and likewise primed versions of SMALL CAPS are used in the postcondition.

The point is that all components which are not mentioned in the postcondition, are not modified by the rule.

To be able to refer to rules, we will give each of them a number; i.e., (3.3). This is printed at the right side of the page, just before the precondition. The rule is referred to as “Rule 3.3”.

3.4.1 Initialization

We now define the initialization of the set of CVMs. Recall that a Creol program is represented as a tuple **(CVM names, Initial objects, Class definition set)**. At initialization, we make an instance of a CVM for each CVM name (at least one CVM). Each CVM will have a central and an empty set of objects. To create the initial objects, we make `newObj`-messages and put them into the in-queue of the Central of the specified CVM.

Formally, the initialization gives rise to a state s_0, called the initial state. Assume that **CVM names** = $\{id_1, id_2, \ldots, id_n\}$, **Initial objects** = {(id_i, *class name*, *parameters*), ...} and **Class definition set** = *classdef*. Then s_0 will be a set of CVMs with empty queues, a central, and an empty set of objects. The Central will have `newObj`-messages in its in-queue, corresponding to the initial objects. Example:

CVM$_i$ = ⟨ id_i, { }, { }, ⟨ {`newObj(null`, *class name*, *parameters*), ...},
{ }, *classdef* ⟩, { } ⟩

Note that just after initialization, the class definition set is equal for all CVMs; this set can change if we introduce dynamic updates.

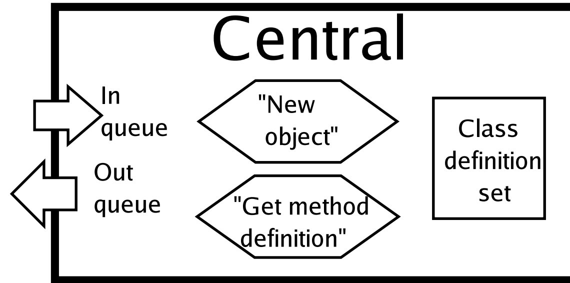

Figure 3.4: The Central stores the class definitions and offers two services: “New object” and “Get method definition”.

3.4.2 The Central

The Central offers services to the objects; see Figure 3.4. A service is invoked by sending a message to the Central. The Central processes each message it receives in its in-queue, performs the requested service, and send an answer message (via its out-queue) back to the object. The services are presented next.

Service: New object

A new CVM object is created when the Central receives a message

$$\texttt{newObj}(\mathit{obj}_{from},\ \mathit{classname},\ \mathit{actual\ parameters})$$

Recall that the Central is modeled by a tuple containing queues and class definitions. Assume the Central has a `newObj` message first in its in-queue:

$$\langle\ \{\texttt{newObj}(\mathit{obj}_{from}, \mathsf{A}, (ap_1, ap_2, \ldots, ap_n)), \ldots\},\ \textsc{Out-queue}, \textsc{Class definition set}\ \rangle \tag{3.1}$$

A identifies the class and ap_is identify the actual parameters. Further, assume

$$\langle \mathsf{A}, (v_1, v_2, \ldots, v_n), \mathit{inhList}, (<u_1,e_1>, <u_2,e_2>, \ldots, <u_m,e_m>), \textsc{Methods}\ \rangle \in \textsc{Class definition set}$$

The v_i's identify formal parameters and the u_i's identify attributes. The inheritance list *inhList* is not interesting for now. A new object

$\langle$ ID, CLASS, IN-QUEUE, OUT-QUEUE, ATT, COMP SET,
ACTIVE PROCESS, SUSPENDED PROCESSES, STATUS $\rangle$

is created, where

ID = `id`
CLASS = `A`
IN-QUEUE = { }
OUT-QUEUE = { }
ATT = { <this,id>, <v_1,ap_1>, <v_2,ap_2>, ..., <v_n,ap_n>,
<u_1,val_1>, <u_2,val_2>, ..., <u_m,val_m> }
COMP SET = { }
SUSPENDED PROCESSES = { }
STATUS = "Process scheduling"

The object identifier `id` is a fresh value not used as any other object identifier; v_i and u_i is assumed to be distinct variable names different from this, caller, label and wait.[8] Each attribute u_i is assigned the evaluated value of e_i (over parameters), that is, val_i = *evaluate*(e_i). If the run[9] method exists, that is

$\langle$ run, ( ), ϵ, ( ), ( ), *Code* $\rangle \in$ METHODS

a new process p is created. Recall that a process is represented as a tuple $\langle$LOCAL VARIABLES, ACTIVE CODE, SUSPENDED CODE, ID, DYNAMICLINK$\rangle$. The process p's components are:

LOCAL VARIABLES = {<caller,null>,<label,null>,<wait,true>}
ACTIVE CODE = *Code*
SUSPENDED CODE = { }
ID = `null`
DYNAMICLINK = `null`

The ACTIVE PROCESS is set to p; if there is no run method, ACTIVE PROCESS is set to null. If obj_{from} is not `null`, an object is waiting to get the new object's identifier. Therefore, the Central sends a `newObjId` message to the object:

`newObjId`(obj_{from}, `id`)

This is the case when objects execute statements v := new *classname*(E).

[8]The Strings this, caller, label and wait are special variable names.

[9]Recall that the run method has no parameters nor return variables, an empty signature, and no cointerface. Hence the empty lists ( ) and empty String ϵ.

Service: Get method definition

The second service offered by the Central is *Get method definition*. This service is invoked by sending a message getMethodDef to the Central. The message has the following syntax:

getMethodDef(*obj*, *class*, *method*)

The object identifier *obj* identifies the object requesting the method definition, *class* is the object's class name, *method* is the method name. Assume that the Central receives a message

getMethodDef(objA, A, m) (3.2)

We do not consider inheritance at this point; therefore, as we assume strong static typing, the class definition set in the Central is guaranteed to have the class definition of A with a method m:

⟨ A, *Parameter list*, *Inheritance list*, *Attributes*,
{ < m, sig, co, param, retVar, Code >, ...} ⟩

The Central answers this inquiry by sending a methodDef message back to object, in this case objA. The syntax of this message is

methodDef(*obj*, *par list*, *ret var*, *code*)

The object identifier *obj* identifies the object to which the message is supposed to be sent, *par list* is the parameter list of the object and *ret var* is a list of Strings giving the variable values which are to be returned. The method body is given as a list of statements in *code*. To construct the actual message, the parameter list par, return variables list ret and the method body code is fetched in the class definition set. This is straight forward and therefore omitted. Note however that the signature sig and cointerface co is irrelevant for now, as we do not consider inheritance.

3.4.3 The Object

In Section 3.3.4 we looked at the *structure* of the CVM object. Now we take a look at the *behavior* of the object, which is given by the three main tasks of the object: message processing, process scheduling, and process execution. See Figure 3.5.

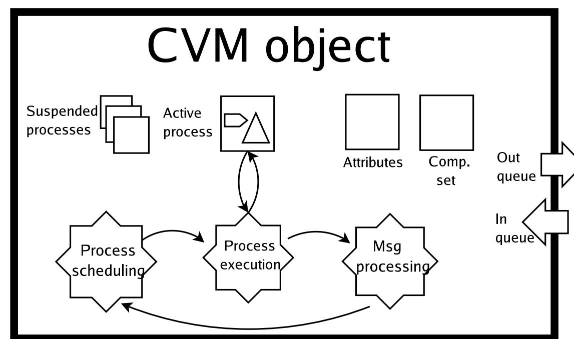

Figure 3.5: The object tasks and the flow of control.

Message processing: The object receives messages in its in-queue. These messages are of two types (invocation or completion of a method call) and must be processed accordingly.[10] See Section 3.4.4.

Process scheduling is the task of deciding which process is to be executed next. It is the active process or one of the suspended processes. Details are explained in Section 3.4.5.

Process execution: The object must check if the active process is able to execute its next statement, and if so execute this statement. Details are discussed in Section 3.4.6.

The object interchanges between these tasks. Depending on which of the tasks it is currently doing, the object is in the *state* of message processing, process scheduling or process execution, and has a corresponding *status*. In addition, we need a status *wait for message* because it is possible that the object is not able to do anything before it gets a message. This is the case if there is no process to execute or if the active process blocks, waiting for the return of an external method call.[11] See Figure 3.6.

Enabledness and readiness of processes and program statements

To define the transitions between the object's tasks, we need the definition of when a process is ready and when it is enabled.

The use of a predicate *enabled* is to precisely define *processor release points*, that is, when the processor may be released to other processes.

[10] Messages of type `methodDef` and `newObjId` (defined in Section 3.4.2) are also put in the object's in-queue; however, these message types are used differently (see Rule 3.9 and Rule 3.24).

[11] I.e., t!o.m(E); t?(V); where o is not the object in which the statements are executed.

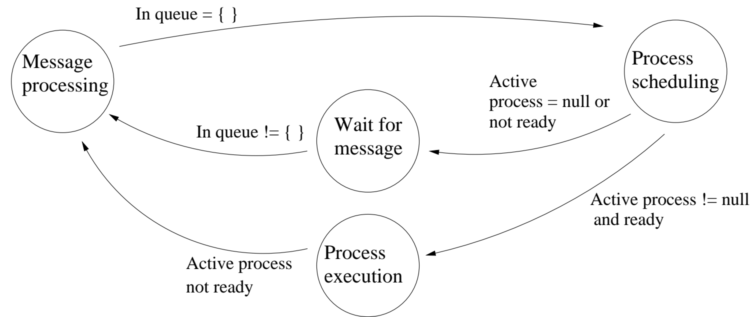

Figure 3.6: Object status changes.

The processor may be released when the active process' next statement is not enabled. It is the await statement which defines processor release points in Creol; thus, most of the definition is concerned with the await statement:

- A process P is enabled if P.ACTIVE CODE is enabled or there exists a statement list *S* in P.SUSPENDED CODE such that *S* is enabled.
- A statement list *S* is enabled if its first statement is enabled.
- $(S_1 \Box S_2)$ is enabled if S_1 or S_2 is enabled.
- $(S_1 ||| S_2)$ is enabled if S_1 or S_2 is enabled.
- await $g_1 \wedge g_2$ for guards g_1 and g_1 is enabled if await g_1 and await g_2 are enabled.
- await *b* for a Boolean expression *b* is enabled if *b* evaluates to true.
- await t? is enabled if the object's COMPLETION SET contains a pair <t, val>, where val can be any list of data values.
- await ¬ t? is enabled if the object's COMPLETION SET *does not* contain a pair <t, val>.
- await wait[12] is enabled if the wait variable for this process is true, that is: P.LOCAL VARIABLES = ⟨ `<wait,true>, ...` ⟩
- All other statements are always enabled (particularly t?(V) is always enabled).

Intuitively, a process is *ready* if it is able to execute at least one statement. It is defined as follows:

[12] The use of wait as a control variable is explained in Section 3.3.3.

- A process P is ready if P.ACTIVE CODE is ready, or if P.ACTIVE CODE is null or *not enabled* and there exists a statement list S in P.SUSPENDED CODE such that S is ready.
- A statement list S is ready if its first statement is ready.
- $(S_1 \square S_2)$ is ready if S_1 or S_2 is ready.
- $(S_1 ||| S_2)$ is ready if S_1 or S_2 is ready.
- The statement t?(V) is ready if the object's COMPLETION SET contains a pair <t, val>, where val can be any list of data values.
- For all other statements s, s is ready if it is enabled.

The only way a process can be enabled but not ready, is if it is waiting for a reply of a method invocation by a reply statement t?(V) and this reply has not yet arrived.

State changes

The status flag of the object specify what kind of operation the object is currently doing. For an object

⟨ ID, CLASS, IN-QUEUE, OUT-QUEUE,
ATTRIBUTES, COMPLETION SET, ACTIVE PROCESS,
SUSPENDED PROCESSES, STATUS ⟩,

STATUS takes the values "Process execution", "Message processing", "Process scheduling" and "Wait for message". There are five transitions which changes the status flag, as given in Figure 3.6 and by the following rules:

Process execution -> Message processing

The object continues to execute the active process until it is no longer ready.

PRECONDITION: (3.3)

ACTIVE PROCESS is *not ready* $\wedge$
STATUS = "Process execution"

POSTCONDITION:

STATUS = "Message processing"

Message processing -> Process scheduling

The object processes *all* messages before scheduling processes.

PRECONDITION: (3.4)

IN-QUEUE = { } ∧
STATUS = "Message processing"

POSTCONDITION:

STATUS = "Process scheduling"

Process scheduling -> Process execution

If the object is able to select a ready process, it goes to the task of process execution.

PRECONDITION: (3.5)

ACTIVE PROCESS is *ready* ∧
STATUS = "Process scheduling"

POSTCONDITION:

STATUS = "Process execution"

Process scheduling -> Wait for message

If there is no ready process, the object must wait for a message to arrive.

PRECONDITION: (3.6)

ACTIVE PROCESS = AP ∧
SUSPENDED PROCESSES = SP ∧
STATUS = "Process scheduling" ∧
((AP = `null` ∧ ¬ ∃ p ∈ SP s.t. p is *ready*) ∨
(AP is *enabled* but *not ready* ∧
¬ ∃ p : (p ∈ SP ∧ p is *ready* ∧ AP *is waiting for* p)))

POSTCONDITION:

STATUS = "Wait for message"

The infix function *is waiting for* is defined in Section 3.4.5. It is used to activate local method calls.

Wait for message -> Message processing

The object waits for an message to arrive.

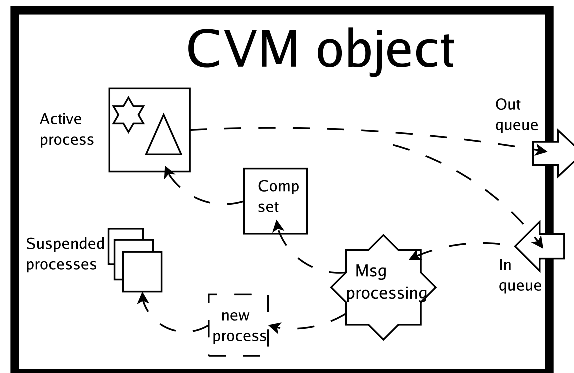

Figure 3.7: Message processing in the object.

PRECONDITION: (3.7)

IN-QUEUE != { } ∧
STATUS = “Wait for message”

POSTCONDITION:

STATUS = “Message processing”

In all other cases, the object must continue execution of the current task, without changing the status flag. The next three sections define the rules of the tasks message processing, process scheduling and process execution. Waiting for a message is not really an object task, as there is nothing to do except from waiting for a message; therefore, there are no rules with “Wait for message” as status.

3.4.4 Message Processing

This section describes the object task of processing messages; it gives rules for what the object should do in the state “Message processing”. There are two different cases to explore; the first is that the object receives a method invocation message, and the second is that it receives a method completion message. We will now give a rather formal description of the effect these two cases have on the object’s state.

See Figure 3.7 for an overview of the message flow inside an object. It might give a better intuitive understanding than the formal description.

Method invocation message

A method invocation message has the following syntax:

`invoc`(obj_{to}, obj_{from}, *label*, *method*, *par*)

The object identifiers obj_{to} and obj_{from} give the callee and the caller of the method, respectively. The *label* value identifies the call inside obj_{from}. The String *method* gives the method name. The actual parameters *par* is a list of data values.

The state of an object processing such a message, is given by a tuple:

$$\langle\ id,\ class,\ \{\texttt{invoc}(id,\ caller,\ label,\ method,\ par), \ldots\}, \text{OUT-QUEUE, ATTRIBUTES, COMPLETION SET, ACTIVE PROCESS, SUSPENDED PROCESSES, “}\texttt{Message processing}\text{”}\ \rangle \tag{3.8}$$

To create the new process, the object needs the method definition. Therefore, it sends a message

`getMethodDef`(*id*, *class*, *method*)

to the Central. Then it waits for the answer; a `methodDef` message. This is received in the object's IN-QUEUE. Other messages in the queue are bypassed by this message.[13] The service *Get method definition* is described in Section 3.4.2. So, when

$$\texttt{methodDef}(id,\ par\ list,\ ret\ var,\ code) \in \text{IN-QUEUE}_{id} \tag{3.9}$$

then the object creates a new process `p`, given by a tuple

⟨ LOCAL VARIABLES, ACTIVE CODE,
SUSPENDED CODE, ID, DYNAMICLINK ⟩

Assume *par list* is (v_1, v_2, ..., v_n), *ret var* is (u_1, u_2, ..., u_m) and *par* is (val_1, val_2, ..., val_n). Then the components of the process is:

LOCAL VARIABLES = { `<caller,obj>, <label,l>, <wait,true>,`
`<v`$_1$`,val`$_1$`>, <v`$_2$`,val`$_2$`>, ..., <v`$_n$`,val`$_n$`>,`
`<u`$_1$`,null>, <u`$_2$`,null>, ...,<u`$_m$`,null>` }
ACTIVE CODE = *code*
SUSPENDED CODE = { }
ID = *a new unique id*
DYNAMICLINK = `null`

The message is removed from the in-queue, and the new process is inserted in the set of suspended processes.

[13] The reason for treating the communication with the Central different from the other objects, is to make it easier to implement this as method calls; the object sends a message and *waits* for the answer, this is (more or less) equivalent to a traditional method call; i.e., Java method calls.

Method completion message

A method completion message has the following syntax:

comp(*destination*, *label*, *return values*)

The object identifier *destination* identifies the object which made the call (and is waiting for the return), *label* is the label identifying the call within the object, and *return values* is a list of data values.

Consider an object which receives a comp message:

(3.10)

⟨ id, {comp(id, *label*, *return values*), . . . },
OUT-QUEUE, ATTRIBUTES, COMPLETION SET,
ACTIVE PROCESS, SUSPENDED PROCESSES, “Message processing” ⟩

The message is removed from the in-queue, transformed into a pair <label, val> and inserted into the completion set:

⟨ id, { . . . }, OUT-QUEUE, ATTRIBUTES,
COMPLETION SET ∪ { <*label*, *return values*> },
ACTIVE PROCESS, SUSPENDED PROCESSES, “Message processing” ⟩

The process waiting for the completion of the method call identified by *label* may now fetch the result in the completion set and avoid inspecting the in-queue directly.

3.4.5 Process Scheduling

The task of *process scheduling* is to suspend or terminate the active process, and to select the new active process (which will be executed next).

Activating a suspended process

If there is no active process, any ready process may be selected:

PRECONDITION: (3.11)

ACTIVE PROCESS = null ∧
SUSPENDED PROCESSES = SP ∧ p ∈ SP ∧ p is *ready*

POSTCONDITION:

ACTIVE PROCESS = p ∧
SUSPENDED PROCESS = SP \ { p }

Activation of local calls

When a process is blocked, that is, it is enabled but not ready, we must activate local calls. The purpose is to allow local processes that the active process is waiting for, to proceed.

To describe this precisely, we use an infix predicate *is waiting for*:

> `p` *is waiting for* `p'` if
> ATTRIBUTES = {`<this,id>, ...`} ∧
> `p'`.LOCAL VARIABLES = {`<caller,id>,<label,`*lval*`>,...`} ∧
> *lval* ∈ *waitsFor*(`p`)

The function *waitsFor* takes as argument a process and returns a set of label values, giving the label values for which the process is waiting. For a process P, P's active code AC, P's suspended code SP, and P's local variables LV,

$$\mathit{waitsFor}(\mathrm{P}) = \begin{cases} \mathit{waitsFor}(\mathrm{AC}, \mathrm{LV}) & \text{if AC is enabled} \\ \mathit{waitsFor}(\{\mathrm{AC}\} \cup \mathrm{SC}, \mathrm{LV}) & \text{if AC is not enabled} \\ \mathit{waitsFor}(\mathrm{SC}, \mathrm{LV}) & \text{if AC is } \mathtt{null} \end{cases}$$

For SSL, a set of statement lists S_i,

$$\mathit{waitsFor}(\mathrm{SSL}, \mathrm{LV}) = \mathit{waitsFor}(S_1, \mathrm{LV}) \cup \ldots \cup \mathit{waitsFor}(S_n, \mathrm{LV})$$

For a statement s and a statement list S,

$$\mathit{waitsFor}(s; S, \mathrm{LV}) = \mathit{waitsFor}(s, \mathrm{LV})$$

For statement lists S_1 and S_2, a label t, a variable list V, guards g_1 and g_2, a boolean expression b, and a set of variables LV,

$$\mathit{waitsFor}((S_1 \,\Box\, S_2), \mathrm{LV}) = \mathit{waitsFor}(S_1, \mathrm{LV}) \cup \mathit{waitsFor}(S_2, \mathrm{LV})$$
$$\mathit{waitsFor}((S_1 \,|||\, S_2), \mathrm{LV}) = \mathit{waitsFor}(S_1, \mathrm{LV}) \cup \mathit{waitsFor}(S_2, \mathrm{LV})$$
$$\mathit{waitsFor}(\mathsf{t?(V)}, \mathrm{LV}) = \{\ \mathit{eval}(\mathsf{t}, \mathrm{LV})\ \}$$
$$\mathit{waitsFor}(\mathsf{await\ t?}, \mathrm{LV}) = \{\ \mathit{eval}(\mathsf{t}, \mathrm{LV})\ \}$$
$$\mathit{waitsFor}(\mathsf{await}\ \neg\ \mathsf{t?}, \mathrm{LV}) = \{\ \}$$
$$\mathit{waitsFor}(\mathsf{await\ wait}, \mathrm{LV}) = \{\ \}$$
$$\mathit{waitsFor}(\mathsf{await}\ b, \mathrm{LV}) = \{\ \}$$
$$\mathit{waitsFor}(\mathsf{await}\ g_1 \wedge g_2, \mathrm{LV}) = \mathit{waitsFor}(\mathsf{await}\ g_1, \mathrm{LV}) \cup \mathit{waitsFor}(\mathsf{await}\ g_2, \mathrm{LV})$$

The *eval* function returns the value of the given variable name, that is, *eval*(t, LV) returns the label value. The function is defined in Section 3.4.6. For all other statements s,

$$\mathit{waitsFor}(\mathsf{s}, \mathrm{LV}) = \{\ \}$$

Remark: The relation *is waiting for* and hence the function *waitsFor* is only used when the active process is not ready. Therefore, the values of

waitsFor(S_1 ||| S_2) and other statements which is always ready, is of no importance.

Now we are ready to define the rule of activating local calls. Notice that the activated process' dynamic link is set, as it is supposed to give control back to the caller when it terminates.

PRECONDITION: (3.12)

> ACTIVE PROCESS = AP ∧ AP is *enabled* ∧ AP is *not ready* ∧
> SUSPENDED PROCESSES = SP ∧ p ∈ SP ∧ p is *ready* ∧
> AP *is waiting for* p ∧
> AP.ID = id

POSTCONDITION:

> ACTIVE PROCESS = p ∧
> SUSPENDED PROCESS = (SP \ { p }) ∪ { AP } ∧
> p.DYNAMICLINK = id

Terminated process

A process has *terminated* if

> p.ACTIVE CODE = null ∧ p.SUSPENDED CODE = { }

If the dynamic link is not set, or if the process identified by the dynamic link has terminated, no process is selected as the active process. The active process is set to null:

PRECONDITION: (3.13)

> ACTIVE PROCESS = AP ∧
> AP.ACTIVE CODE = null ∧ AP.SUSPENDED CODE = { } ∧
> SUSPENDED PROCESSES = SP ∧
> (AP.DYNAMICLINK = null ∨
> ¬ ∃ p : (p ∈ SP ∧ p.ID = AP.DYNAMICLINK))

POSTCONDITION:

> ACTIVE PROCESS = null

If the dynamic link is set and the process p identified by the dynamic link has not terminated, p is set as the active process:

PRECONDITION: (3.14)

> ACTIVE PROCESS = AP ∧
> AP.ACTIVE CODE = null ∧ AP.SUSPENDED CODE = { } ∧
> AP.DYNAMICLINK != null ∧ SUSPENDED PROCESSES = SP ∧
> p ∈ SP ∧ p.ID = AP.DYNAMICLINK

POSTCONDITION:

ACTIVE PROCESS = p

Suspending the active process

If the active process has not terminated but is not enabled, it is inserted into the queue of suspended processes:

PRECONDITION: (3.15)

ACTIVE PROCESS = AP $\wedge$ AP is *not enabled* $\wedge$
(AP.ACTIVE CODE != null $\vee$ AP.SUSPENDED CODE != { }) $\wedge$
AP.LOCAL VARIABLES = LV $\wedge$
SUSPENDED PROCESSES = SP

POSTCONDITION:

ACTIVE PROCESS = null $\wedge$
AP.LOCAL VARIABLES = *update*(LV, wait, true) $\wedge$
SUSPENDED PROCESS = SP $\cup$ { AP }

Notice that wait is updated; thus, *enabled*(await wait) evaluates to true after suspension, which corresponds to the semantics of await wait.

3.4.6 Process Execution

The following describe the effect of executing statements. This is done when the object is in the state *Process execution*. Formally, the state of the object for which the following rules apply is:

⟨ ID, CLASS, IN-QUEUE, OUT-QUEUE, ATTRIBUTES,
COMPLETION SET, ACTIVE PROCESS,
SUSPENDED PROCESSES, "Process execution" ⟩

It is the active process' statements that are executed; these statements must be *ready*[14] to be executed. If the active process is *not* ready, the object will go to the state *Message processing* (Rule 3.3).

Recall the definition of the state of a process:

⟨ LOCAL VARIABLES, ACTIVE CODE, SUSPENDED CODE,
ID, DYNAMICLINK ⟩

We will soon define the rules concerning execution of each type of statement, but first we define some functions used in these rules.

[14] See Section 3.4.5 for the definition of *ready*.

Preliminary definitions

The function *eval* is used to evaluate expressions. It takes as argument an expression and a set of pairs `<id,val>`. It returns data values; e.g., a String, an Integer or a Boolean value.

eval(Expression, VariableSet)

For a variable name `id`,

eval(`id`, {`<id,val>`, ...}) = `val`

In Boolean expressions, $\wedge$, $\vee$ and $\neg$ are evaluated as expected. In Integer expressions, $+$, $-$, $*$, etc. are evaluated as expected. The function is extended to list of expressions; if its first argument is a list, it returns a list of data values.

The function *evalGuard* takes as arguments a guard, a variable set and a label set. Both the variable set and the label set are a set of pairs `<id,val>`.

evalGuard(Guard, VariableSet, LabelSet)

The set variable set corresponds to the local variables of a process and the persistent variables of an object, and the label set corresponds to the completion set of an object. The function returns a Boolean. For a Boolean expression b, a label t, guards g_1 and g_2, a variable set VS and a label set LS,

evalGuard(b, VS, LS) = *eval*(b, VS)
evalGuard(t?, VS, LS) = *eval*(t, VS) $\in$ *var*(LS)
evalGuard($\neg$ t?, VS, LS) = not *eval*(t, VS) $\in$ *var*(LS)
evalGuard($g_1 \wedge g_2$, VS, LS) = *evalGuard*(g_1, VS, LS) and
evalGuard(g_2, VS, LS)

The function *update* is used to update variable values in the local variables and the attributes. It takes as arguments a set of pairs `<id,val>`, a list of variable names (v_1, v_2, ..., v_n) and a list of values (val_1, val_2, ..., val_n):

update(VariableSet, VariableList, ValueList)

Note that for a given variable name v_i in the variable list, it does not necessarily exist a pair with corresponding identifier. The function updates only pairs which exist in the variable set. To be more precise, we define the function as follows: for all $v_i \in$ VariableList and corresponding val_i $\in$ ValueList, if there exists id and val such that $v_i = id$ and $<id,val> \in$ VariableSet, then $<id,val>$ is replaced by $<v_i,val_i>$. The resulting set is returned. Where appropriate, we will use a single variable name v and corresponding value val as an abbreviation for the lists (v) and (val), respectively.

In the following, the set union operator $\cup$ is used for both ordered and unordered sets; if the set is ordered, the ordering is preserved. We use \ for set intersection and + for list concatenation and application.

Suspending and activating code

If the process' ACTIVE CODE is not enabled, it is suspended:

PRECONDITION: (3.16)

ACTIVE CODE = *code* $\wedge$ SUSPENDED CODE = SC $\wedge$
code is *not enabled*

POSTCONDITION:

ACTIVE CODE = null $\wedge$ SUSPENDED CODE = SC $\cup$ { *code* }

If there is no active code to execute, one of the suspended statement lists which is ready is selected as the active code:

PRECONDITION: (3.17)

ACTIVE CODE = null $\wedge$
SUSPENDED CODE = SP $\cup$ { *code* } $\wedge$ *code* is *ready*

POSTCONDITION:

ACTIVE CODE = *code* $\wedge$ SUSPENDED CODE = SP

Statement execution

We now define rules for the execution of Creol statements in the model. The statements must be *ready*, as defined in Section 3.4.3. The system variable wait is, as explained in Section 3.3.3, updated to *false* when executing an 'atomic' statement; that is, for statements different from $(S_1 \Box S_2)$ and $(S_1 ||| S_2)$, wait := false is a side-effect of executing the statement. When executiong $(S_1 \Box S_2)$ or $(S_1 ||| S_2)$, wait is not updated. The reason for this is that updating wait may cause the process to be suspended twice. Consider the following example: The active process' next statement is (await wait; $S_1 \Box$ await wait; S_2). As neither the left nor the right branch is enabled, the process is suspended and the wait variable is updated to *true*. When this process is activated, both the left and the right branch are ready and either of them is selected, say the left: await wait; S_1. If the wait variable is updated to *false* now, await wait is evaluated to *false* and the process is suspended again, unintentionally. By only updating wait when executing 'atomic' statements, we avoid this problem.

Variable declaration var u_1:T_1=e_1,u_2:T_2=e_2,...,u_n:T_n=e_n

For each variable u_i, u_i is added to the local variables. The expression e_i is evaluated and the resulting value is assigned to u_i.

PRECONDITION: (3.18)

Active code = var u_1:T_1=e_1,u_2:T_2=e_2,...,u_n:T_n=e_n; S ∧
Local variables = LV ∧ Attributes = ATT

POSTCONDITION:

Active code = S ∧
Local variables = *update*(LV, wait, `false`) ∪
$\{<u_1, eval(e_1, LV \cup ATT)>, \ldots, <u_n, eval(e_n, LV \cup ATT)>\}$

Multiple assignment V := E

For variable list V and expression list E, each $e_i \in$ E is evaluated and the resulting value is assigned to $v_i \in$ V. Strong typing ensures that each v_i ∈ V exists. Further, we have assumed that all variable names are distinct. Thus, there is a pair $<v_i, val>$ in the local variables *or* in the attributes, but not in both.

PRECONDITION: (3.19)

Active code = V := E; S ∧
Local variables = LV ∧ Attributes = ATT

POSTCONDITION:

Active code = S ∧
Local variables = *update*(LV, V+wait, *eval*(E,LV∪ATT)+`false`) ∧
Attributes = *update*(ATT, V, *eval*(E, LV ∪ ATT))

Passing of processor release point await g

To execute an Await statement, this statement must be *ready*. As other basic statements, a side effect is that wait is set to false:

PRECONDITION: (3.20)

Active code = await g; S ∧
Local variables = LV ∧ Attributes = ATT ∧
Completion set = CS ∧
evalGuard(g, LV ∪ ATT, CS) = `true`

POSTCONDITION:

Active code = S ∧
Local variables = *update*(LV, wait, `false`)

Note that all statements including this is not executed if it is not ready, that is, if *evalGuard*(g, LV ∪ ATT, CS) = `false`. Then other rules apply; e.g., Rule 3.16.

If statement if b then S_1 else S_2 fi

For the if statement, there are two cases to explore.
1) b evaluates to true:

PRECONDITION: (3.21)

ACTIVE CODE = if b then S_1 else S_2 fi; S_3 $\wedge$
LOCAL VARIABLES = LV $\wedge$ ATTRIBUTES = ATT $\wedge$
eval(b, LV $\cup$ ATT) = `true`

POSTCONDITION:

ACTIVE CODE = S_1; S_3 $\wedge$
LOCAL VARIABLES = *update*(LV, wait, `false`)

2) b evaluates to false:

PRECONDITION: (3.22)

ACTIVE CODE = if b then S_1 else S_2 fi; S_3 $\wedge$
LOCAL VARIABLES = LV $\wedge$ ATTRIBUTES = ATT $\wedge$
eval(b, LV $\cup$ ATT) = `false`

POSTCONDITION:

ACTIVE CODE = S_2; S_3 $\wedge$
LOCAL VARIABLES = *update*(LV, wait, `false`)

New object new *classname*(E); waitObjId(v)

Recall that the Creol statement v := new *classname*(E) is changed to the statements new *classname*(E); waitObjId(v) in CCM. Further, the waitObjId(v) is ready even though the `newObjId` message has not yet arrived; thus, it blocks the object in the process execution state.

To execute these statements, the object must send a `newObj` message to the Central, wait for an answer message `newObjId` and assign the object value given by this message to v.

PRECONDITION: (3.23)

ACTIVE CODE = new *classname*(E); waitObjId(v); S $\wedge$
LOCAL VARIABLES = LV $\wedge$
ATTRIBUTES = ATT = { `<this,myid>`, `...`} $\wedge$
OUT-QUEUE = OQ

POSTCONDITION:

ACTIVE CODE = waitObjId(v); S $\wedge$
OUT-QUEUE = OQ $\cup$ {`newObj`(`myid`, *classname*, *eval*(E, LV $\cup$ ATT)}

Now the object must wait to get the object identifier of the new object; when it arrives the identifier is applied to v:

PRECONDITION: (3.24)

ACTIVE CODE = waitObjId(v); S ∧
LOCAL VARIABLES = LV ∧ ATTRIBUTES = ATT ∧
IN-QUEUE = { newObjId(id$_{this}$, id$_{newobject}$) } ∪ IQ

POSTCONDITION:

ACTIVE CODE = S ∧ IN-QUEUE = IQ ∧
LOCAL VARIABLES = *update*(LV, (v, wait), (id$_{newobject}$, false)) ∧
ATTRIBUTES = *update*(ATT, (v), (id$_{newobject}$))

Remark: The statement waitObjId(v) is always ready, even when it is not possible to execute it. This may seem strange. The reason for blocking the object in the execution state, is that we consider the statement v := new *classname*(E) to be implemented by method calls. Sending a message and waiting for the answer is in accordance with traditional method calls as in, e.g., Java.

Method invocation without label !$o.m$(E)

The first type of method invocation statement is !$o.m$(E), where o is an object expression, m is the method name and E is an expression list. See Section 3.3.3. Recall that a message invocation message has the following syntax (see Section 3.4.4):

invoc(*obj*$_{to}$, *obj*$_{from}$, *label*, *method*, *par*)

This message type is used by all method invocation variations; if the method call has no label, this is set to null. The call may be local or external; therefore, we have two rules. The first rule considers local calls:

PRECONDITION: (3.25)

ACTIVE CODE = !$o.m$(E); S ∧
LOCAL VARIABLES = LV ∧
ATTRIBUTES = ATT = {<this,*id*>, ...} ∧
IN-QUEUE = IQ ∧
eval(o, LV ∪ ATT) = id

POSTCONDITION:

ACTIVE CODE = S ∧
LOCAL VARIABLES = *update*(LV, wait, false) ∧
IN-QUEUE = IQ ∪ invoc(id, id, null, m, *eval*(E,LV∪ATT))

The second rule considers external calls:

PRECONDITION: (3.26)

ACTIVE CODE = $!o.m$(E); S ∧
LOCAL VARIABLES = LV ∧
ATTRIBUTES = ATT = {<this,id>, ...} ∧
OUT-QUEUE = OQ ∧
eval(o, LV ∪ ATT) != id

POSTCONDITION:

ACTIVE CODE = S ∧
LOCAL VARIABLES = *update*(LV, wait, false) ∧
OUT-QUEUE = OQ ∪ invoc(*eval*(o, LV ∪ ATT), id, null, m, *eval*(E, LV ∪ ATT))

Method invocation with label $t!o.m$(E)

The second type of method invocation statement is $t!o.m$(E). The only difference is the additional label t, which is guaranteed to be declared by strong static typing. To give t a unique value, we use a function *fresh*() which gives a new fresh identifier each time it is used. (This may be implemented by having an integer for each object which is incremented and returned each time *fresh*() is called.) As for calls without label, the call may be local or external, and we will have two rules. The first considers local calls:

PRECONDITION: (3.27)

ACTIVE CODE = $t!o.m$(E); S ∧
LOCAL VARIABLES = LV ∧
ATTRIBUTES = ATT = {<this,id>, ...} ∧
IN-QUEUE = IQ ∧
eval(o, LV ∪ ATT) = id

POSTCONDITION:

ACTIVE CODE = S ∧ $label$ = *fresh*() ∧
LOCAL VARIABLES = *update*(LV, (t, wait), ($label$, false)) ∧
IN-QUEUE = IQ ∪ invoc(id,id,$label$,m,*eval*(E,LV∪ATT))

The second rule considers external calls:

PRECONDITION: (3.28)

ACTIVE CODE = $t!o.m$(E); S ∧
LOCAL VARIABLES = LV ∧
ATTRIBUTES = ATT = {<this,id>, ...} ∧
OUT-QUEUE = OQ ∧
eval(o, LV ∪ ATT) != id

POSTCONDITION:

ACTIVE CODE = $S \wedge label = fresh() \wedge$
LOCAL VARIABLES = $update(\mathsf{LV}, (t, \mathsf{wait}), (label, \mathtt{false})) \wedge$
OUT-QUEUE = $\mathsf{OQ} \cup \mathtt{invoc}(eval(o, \mathsf{LV} \cup \mathsf{ATT}), id, label, m,$
$eval(\mathsf{E}, \mathsf{LV} \cup \mathsf{ATT}))$

Reply $t?(\mathsf{V})$

The reply statement assigns the values of the reply identified by the label value of t to the given variables in the variable list V. As only ready statements are executed, the reply identified by t must arrive before this rule can be applied. In the following, D is the data list returned.

PRECONDITION: (3.29)

ACTIVE CODE = $t?(\mathsf{V}); S \wedge$
LOCAL VARIABLES = $\mathsf{LV} = \{\langle t, label\rangle, \ldots\} \wedge$
ATTRIBUTES = $\mathsf{ATT} \wedge$ COMPLETION SET = $\{ \langle label, \mathsf{D}\rangle, \ldots\}$

POSTCONDITION:

ACTIVE CODE = $S \wedge$
LOCAL VARIABLES = $update(\mathsf{LV}, \mathsf{V} + \mathsf{wait}, \mathsf{D} + \mathtt{false}) \wedge$
ATTRIBUTES = $update(\mathsf{ATT}, \mathsf{V}, \mathsf{D})$

Nondeterministic choice $(S_1 \square S_2)$

The nondeterministic choice statement selects a statement list which is ready. At least one of the statement lists must be ready, or else it will not be executed. If both are ready, either one may be selected. This gives two symmetric rules; the first selects the left branch:

PRECONDITION: (3.30)

ACTIVE CODE = $(S_1 \square S_2); S_3 \wedge ready(S_1)$

POSTCONDITION:

ACTIVE CODE = $S_1; S_3$

The second rule selects the right branch:

PRECONDITION: (3.31)

ACTIVE CODE = $(S_1 \square S_2); S_3 \wedge ready(S_2)$

POSTCONDITION:

ACTIVE CODE = $S_2; S_3$

As discussed previously, the wait variable is not updated.

Merge $(S_1 \mid\mid\mid S_2)$

The merge operator interleaves the execution of two statement lists. To simulate this, we use the construct *suspended code*. As at least one of the two branches must be suspended, ||| should be defined as a processor release point. Thus we can suspend *both* branches.

PRECONDITION: (3.32)

ACTIVE CODE = $(S_1 \mid\mid\mid S_2)$; S_3 ∧ SUSPENDED CODE = SP

POSTCONDITION:

ACTIVE CODE = `null` ∧
SUSPENDED CODE = SP ∪ { S_1; S_3 } ∪ { S_2; S_3 }

Recall that the first statement of S_3 is a joinMerge statement. The joinMerge statement is executed twice; the rest of S_3 is only executed once.

Join Merge joinMerge(v)

Recall (Section 3.3.3) that v is initially false (Section 3.3.3). When the first statement list is finished, v is set to true:

PRECONDITION: (3.33)

ACTIVE CODE = joinMerge(v); S ∧
LOCAL VARIABLES = LV = { `<v,false>`, ...}

POSTCONDITION:

ACTIVE CODE = `null` ∧
LOCAL VARIABLES = *update*(LV, v, `true`)

When the second statement list finishes, the statement is passed, and S is executed:

PRECONDITION: (3.34)

ACTIVE CODE = joinMerge(v); S ∧
LOCAL VARIABLES = { `<v,true>`, ...}

POSTCONDITION:

ACTIVE CODE = S

Remark: As the variable v is unique for this particular |||, the variable will not be used again (as Creol has no loop construct). Therefore, v may be left unchanged. Introducing a while statement to Creol implies that the rule must be changed; i.e., adding LOCAL VARIABLES = *update*(LV, `wait`, `false`) to the postcondition of rule 3.34.

Sending completion message return *VarList*

Recall that the statement list defining the method body is extended with a statement return *VarList*, where *VarList* is a list giving the return parameters. The *return*-statement is the last statement of the method body; hence the active code is set to null after this statement is executed.

There are three different possibilities, reflected by three rules. The first considers the possibility that this was a method invocation without label (rules 3.25 and 3.26); this is the case if the process' label variable is null. Then no completion message is sent:

PRECONDITION: (3.35)

ACTIVE CODE = return (u_1, u_2, ..., u_n) $\wedge$
LOCAL VARIABLES = {<label,null>, ... }

POSTCONDITION:

ACTIVE CODE = null

If the process' label variable is different from null, a completion message must be sent. Recall the syntax of a completion message:

comp(*destination*, *label*, *return values*)

The second rule considers local calls. If the process' caller variable is equal to the attribute this, the message is put in the in-queue:

PRECONDITION: (3.36)

ACTIVE CODE = return (u_1, u_2, ..., u_n) $\wedge$
LOCAL VARIABLES = { <label,l>,<caller,c>,
<u_1,v_1>,<u_2,v_2>, ...,<u_n,v_n>, ... } $\wedge$
ATTRIBUTES = {<this,id>, ...} $\wedge$
IN-QUEUE = IQ $\wedge$
l != null $\wedge$ $c = id$

POSTCONDITION:

ACTIVE CODE = null $\wedge$
IN-QUEUE = IQ $\cup$ comp(id, l, (v_1, v_2, . . , v_n))

The third rule considers external calls. If the process' caller variable is not equal to the attribute this, the message is put in the out-queue:

PRECONDITION: (3.37)

ACTIVE CODE = return (u_1, u_2, ..., u_n) $\wedge$
LOCAL VARIABLES = { <label,l>,<caller,c>,
<u_1,v_1>,<u_2,v_2>, ...,<u_n,v_n>, ... } $\wedge$
ATTRIBUTES = {<this,id>, ...} $\wedge$
OUT-QUEUE = OQ $\wedge$
l != null $\wedge$ c != id

POSTCONDITION:

ACTIVE CODE = `null` $\wedge$
OUT-QUEUE = OQ $\cup$ `comp`(c, l, (v_1, v_2, .., v_n))

Remark: If the method has no return variables, the empty list `( )` is returned, similar to an acknowledgment message. Note also that the suspended code set is empty when the return statement is executed; hence, after executing this statement, the process' active code is `null` and the suspended code is empty; thus, the process has terminated and is removed (by rules 3.13 and 3.14).

3.4.7 Message Transportation

All the rules we have defined so far describe local transitions in the objects and in the Central. We have not defined any rules for message transportation. This is omitted on purpose, as we do not want to dictate how this should be implemented.

Here we will sketch how the model may be extended to take care of message transportation. First, we need rules for communication between objects and the Central (within a CVM):

- Service messages `getMethodDef`(...) and `newObj`(...) in an object's out-queue must be moved to the Central's in-queue.
- Service messages `methodDef`(o, ...) and `newObjId`(o, ...) in the Central's in-queue must be moved to the in-queue of the object o.

Second, we need rules for communication between objects. As objects may be distributed among different CVMs, this yields four rules:

- If a message `comp`(o, ...) or `invoc`(o, ...) in an object's out-queue is addressed to an object in the same CVM, that is, there is an object with id o in the CVM, the message is just moved directly to o's in-queue.
- If there is no such object in the CVM, the message is moved to the CVM's out-queue.
- A message `comp`(o, ...) or `invoc`(o, ...) in a CVM's out-queue is moved to the in-queue of the CVM in which the object with id o exists.
- A message `comp`(o, ...) or `invoc`(o, ...) in a CVM's in-queue is moved to o's in-queue.

Communication between objects in different CVMs may be a challenge for implementation, as each CVM is supposed to be run on different physical machines.

3.5 Summary

In this chapter we have developed a computational model for Creol. First, we presented our motivation for defining a formal model: it gives an unambiguous and concise way of describing the computations of programs. Then the model was presented: a representation of a Creol program as a tuple (CVM names, Initial objects, Class definition set). The structure and representation of states were presented as tuples and sets. We saw that in order to make computations feasible, we had to make some changes to and assumptions about the method bodies (the imperative code). The initial state of the computation of CCM was defined, and the successor states were defined by pre- and post-conditions. Finally, we gave an informal description of message transportation.

Chapter 4

Implementation of the Creol Virtual Machine

This chapter discusses the implementation of a Creol Virtual Machine (CVM). The Creol Computational Model goes beyond this, as it discusses a *set* of CVM's. The difference is not substantial, except that communication between CVM's must be handled if we have more than one CVM. Communication between CVM's will be discussed in Chapter 6.

The CVM will be implemented in Java. The choice of language is, of course, not arbitrary. In Section 4.1 we start by pointing out some important properties of Java which were important for choosing Java as the implementation language. The section also discusses some early decisions and assumptions we made about the Java Virtual Machine. In Section 4.2 we present an overview of the implementation; in Section 4.3 we look into details. The reader is assumed to be familiar with Creol and Java, including Java threads. The introductions to Creol and Java in Chapter 2 should suffice.

4.1 Preliminaries

Before discussing the implementation of the virtual machine, we consider some aspects of Java which made it attractive as an implementation language, and we present some assumptions we have made about the Java Virtual Machine.

4.1.1 Java Properties

The project is to write a virtual machine for Creol. First, we had to choose an implementation language. Both C++ and C# were considered

to be adequate; however, we chose Java. Without discussing other languages, we list some properties of Java which was important for the decision to choose it as the implementation language:

- *Object oriented:* Java is an object oriented language. This fits well with how we model the execution of Creol programs.
- *Multi-threaded:* This is important as Creol objects are active and execute concurrently (at least conceptually).
- *Network:* Java has extensive and easy-to-use communication primitives, and both code and objects may be transferred across networks. These properties are important for communication between CVMs and for distributed updating of Creol class definitions at runtime.
- *Widespread use:* We wanted to use an implementation language which is familiar to many people, both because this makes the implementation easier to understand for a reader and because others may want to change or expand the implementation.

4.1.2 JVM Assumptions

The Java specification does not give strong guarantees about fairness of threads, nor does it say anything about the number of threads a JVM implementation needs to support. This is unfortunate, as we would like to use the multi-threaded capabilities of Java and therefore need some guarantees concerning thread scheduling and efficiency.

An implementation of a Java Virtual Machine may give better properties than what is given by the Java specification. This is the case for Sun's JVM (version 1.5) for both Windows and Linux. The following assumptions are fulfilled by Sun's JVM and probably by most other JVMs.

First, we assume that the JVM is preemptive, that is, after some period of time (i.e. 10 ms), the current executing thread is suspended and another thread is executed. This property implies that all threads are given processor time, even if other threads run forever without blocking.[1] Hence, we can benefit from the scheduling of threads in the JVM.

Second, we assume that the JVM supports a high number of threads. Normally, the number of threads supported is limited by memory size. Tests of Sun's JVM showed that it could handle about 3500 threads on

[1] Java threads may have different priorities and some JVMs do not execute a thread if there is a higher priority thread which is not blocked. We do not consider this to be an issue, as giving all threads the same priority solves this problem.

a Linux implementation and about 7000 threads on a Windows implementation. We consider this to be enough for our purpose.

4.2 Implementation Overview

In this section we give an overview of the structure of the virtual machine. The implementation follows the Creol Computational Model to a great extent; however, there are some differences due to efficiency, ease of implementation and properties of Java.

4.2.1 Main CVM Parts

We now give an overview of the objects which compose the virtual machine. In later sections we go into details and discuss the actual implementation of the different parts.

In the model in Chapter 3 we identified components as

- The CVM
- The Central
- CVM objects
- Message queues

In the implementation, we find a similar structure. The virtual machine is composed of

A central which offers services.

Creol objects which corresponds to what was called *CVM objects* in the model and *objects* in the Creol language.

Message queues which corresponds to the message queues in the model.

In addition we have an object responsible for initializing the virtual machine and creating initial objects; this object we simply call 'CVM'. As in the model, the Central must support the services "New object" and "Get method definition". Later we will add other services to the Central. Each Creol object has an in-queue and an out-queue.[2] A Creol object inserts messages in its out-queue and receives messages in its in-queue; message transportation is not part of the object's tasks. How messages are transported is discussed in the next section.

[2] We do not consider communication between different CVMs yet; therefore, there are no CVM queues.

4.2.2 Activity: Flow of Control

In Java, the flow of control is separated from the objects, that is, each object does *not* have its own execution thread as in Creol. Because the thread models differ, an important design choice is how the control flow should be implemented. The choice of solution will have great impact on both the efficiency of the program and on how parts of the program should be implemented.

The activity in a CVM, corresponding to computations in the model, is distributed among different components. Each Creol object has internal activity. The Central has activity in form of the services offered. In addition, we have the activity of message transportation.

We will use one execution thread for each Creol object. This simplifies the implementation considerably, as the operating system and the Java VM take care of the scheduling of Creol objects. We get concurrency for free and can concentrate on internal activity in the Creol object.

In the model, the Central is modeled as an active object which receives service messages, carries out the services and answers by completion messages. The object for which the Central is performing a service, waits until it receives the completion message. For efficiency reasons, the Central will *not* be implemented as an active object. Instead, the services will be offered by two methods: *getMethodDefinition*() and *newObject*(). The Creol objects call these methods to invoke the services; thus, we use the Creol object's execution thread when performing a service. This is more efficient because:

- The services can be invoked without involving another thread; no thread context switch is needed.
- It enables Creol objects to invoke services concurrently, as Java allows multiple threads to execute an object's methods at the same time. We must synchronize to get exclusive write access, but by using a lock which supports multiple readers we still get concurrent execution of part of the services.[3]

Message transportation is not defined in the model. There are a number of possibilities for how it can be implemented:

[3] We give preference to parallel execution over the amount of synchronization. The additional synchronization which is required to get this increased parallelism, might make our solution more inefficient than a more 'straight-forward' solution using e.g. the `synchronized` construct. However, we focus on efficiency *in theory* and not in practice.

- A *message transporter object* polls out-queues for messages and moves each message to the in-queue of the addressed Creol object.
- Each *Creol object* transports the messages.
- Each *out-queue* transports the messages.

The first solution is inefficient. The transporter would have to inspect all the out-queues to look for messages, then sleep for a while before checking all the out-queues again. It can be done more efficiently if each Creol object signals the transporter object each time a message is sent, but this would violate the principle of encapsulation as the transporter object and the out-queue (or the Creol object) would be tied together. The second solution violates the principle of *encapsulation*; we do not want the Creol object to worry about message transportation. The Creol object is only supposed to put messages in the out-queue and nothing more.

It will be the out-queue which transports the messages. However, we will use the Creol object's execution thread. That is, the out-queue offers a method *put*() to the Creol object. When called, this method forwards the message to the in-queue. This solution implies that the Creol object is blocked for a while, but we consider this as a fair trade off for less threads and less synchronization. To be able to transport a message, the out-queue needs a reference to the in-queue of the addressed Creol object. As it is the Central which creates the Creol objects and their queues, it is natural that it also stores references to the in-queues. We extend the Central to offer a service "Get queue". As other services in the Central, "Get queue" is implemented by a method. The out-queue calls *getQueue*() which returns the in-queue of the specified Creol object (specified by the object identifier). Then the out-queue inserts the message in the in-queue.

4.2.3 Classes and Interfaces

We now take a closer look at the actual implementation of the objects discussed in the previous sections.

Objects in Java are implemented by classes and typed by classes and interfaces. Java supports single inheritance for classes and multiple inheritance for interfaces. We use interfaces to enforce encapsulation, that is, objects hide their data and methods from the rest of the world. This way the internal data and methods of an object can be changed without changing the way the object is used. The dependencies between the virtual machine components are reduced, and the machine will be easier

to understand and modify. It will be easier to *understand* because we can reason about one part of the machine at a time; e.g., the Creol object. It will be easier to *modify* because we can change one component which does not affect other components; e.g., if we want to change the out-queue to have its own execution thread, this is possible without changing anything in the Creol object.

The initializing CVM object is implemented by a class CVM, whereas the Central is implemented by a class Central. The service offered by the Central to message queues is of no interest for the Creol object, and vice versa. Therefore, we use two interfaces CreolObjectServices and MsgQueueServices to restrict which methods can be accessed by the Creol objects and the message queues, respectively. These interfaces are implemented by the Central class.

The Creol object is implemented by a class CreolObject. As this object is supposed to be active, it must either inherit the Thread class or implement the Runnable interface; we have chosen the latter solution. The Creol objects use message queues to communicate. The Creol object is not concerned about how the queues are actually implemented, as long as they offer the methods needed. For the out-queue, a method *insert*() is the only method required; therefore, we have an interface MsgQueueOut which offers this method. Similarly, we have an interface MsgQueueIn which offers a method *next*() and some additional methods which will be discussed later. The Creol object also needs to treat the in-queue as an out-queue, as invocation and completion messages will be inserted directly into the in-queue if the call is local. Therefore, we have an interface MsgQueueInOut which inherits both MsgQueueIn and MsgQueueOut. As discussed in Section 4.2.2, the out-queue will forward the message to the in-queue of the intended Creol object. Therefore, we have a class ForwardingMsgQueue which implements MsgQueueOut. We also have a class MsgQueue which implements the interface MsgQueueInOut.

Summarized, we have the following classes and interfaces:

- class CVM
- class Central
- interface CreolObjectServices, implemented by Central
- interface MsgQueueServices, implemented by Central
- class CreolObject, which implements Runnable
- interfaces MsgQueueIn, MsgQueueOut and MsgQueueInOut
- class MsgQueue, which implements MsgQueueInOut

- class ForwardingMsgQueue, which implements MsgQueueOut

Figure 4.1 shows a graphical representation of the classes and interfaces, and their relationships.

4.3 Implementation Details

In this section we look more into details. We present how Creol programs are represented, and discuss the class CVM and initialization of the CVM. Further, we discuss the Creol object, the Central and the message queues.

We will not discuss the implementation of basic data structures as lists and sets, nor will we discuss methods which are straight forward implementations of functions defined in the model in Chapter 3. The selected code snippets and Java classes we present are without comments to keep things short; the explanations are given in the text. The full code is available on Internet, see Appendix C. To refer to a class A's method *m*, we use dot notation: A.*m*(), even when the method *m* is not static[4]. The same will be done for interfaces; i.e., I.*m*(), even though this does not make sense within the Java language. We leave out parameters when these are not interesting.

4.3.1 Creol Program Representation

The Creol virtual machine executes Creol programs. These programs should be represented in such a way that execution is efficient and simple. We could read, interpret and manipulate *textual* Creol code at run time. Then there would be no need to translate the code before executing it. But we believe this to be both error-prone and inefficient, and it would make our machine very complex. Instead of interpreting text, all Creol constructs are transformed into Java objects (instances of Java classes). This way we can define operations on an object in its class, and we enforce encapsulation. For instance, the semantics of executing an assignment statement is defined by a method *execute*() in a class Assignment.

In Chapter 3, we defined a Creol computational model

(CVM names, Initial objects, Class definition set)

[4]In Java, if A is a class and *m* is a method, A.*m*() is called a static reference. Static references to non-static methods are not allowed. These methods are called by object references; i.e., for an object o of class A: o.*m*().

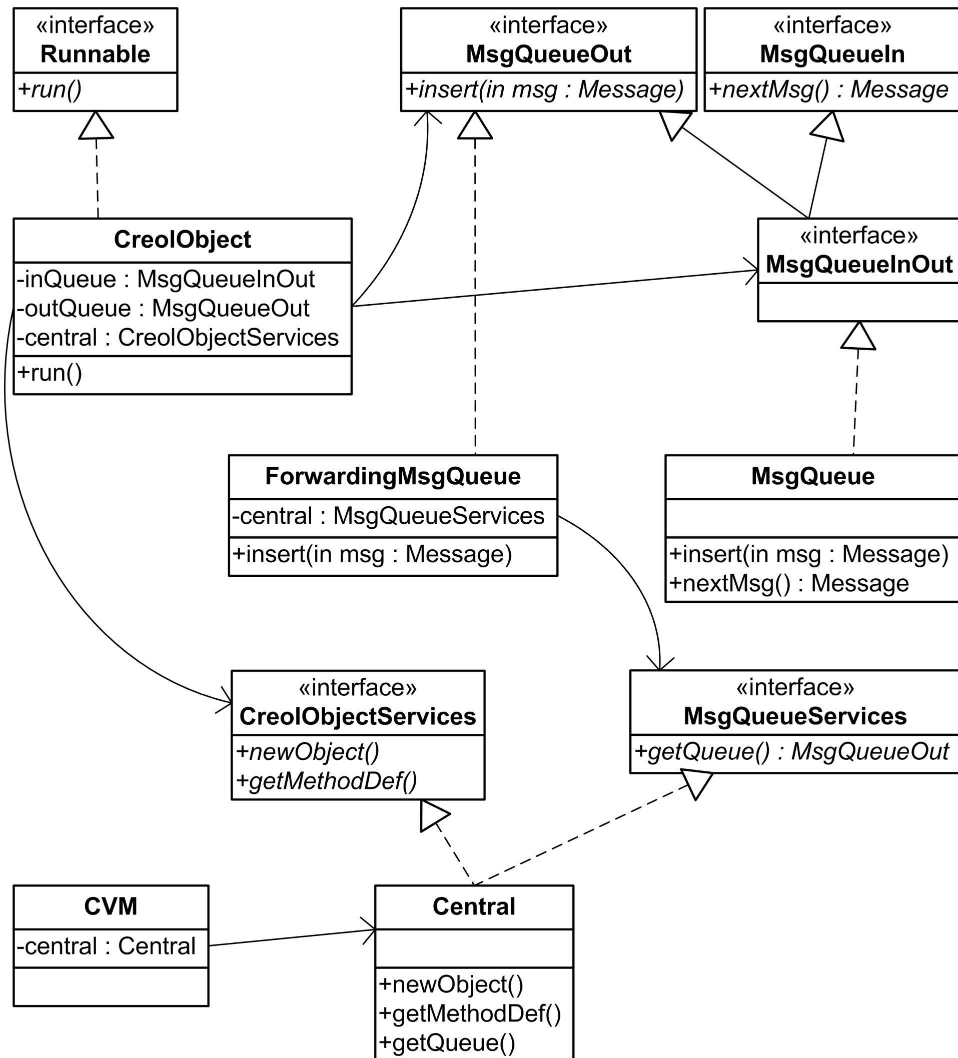

Figure 4.1: UML class diagram [23] giving an overview of some of the interfaces and classes of the Creol Virtual Machine.

```
public class BoundedBuffer implements CreolProgram {
  public ClassDefinitionSet getClassDefinitions() {
    ClassDefinitionSet classdef = new ClassDefinitionSet();
    ...
    return classdef;
  }

  public InitObjectList getInitObjects() {
    return new InitObjectList(new InitObject("Starter", null));
  }
}
```

Figure 4.2: The Bounded Buffer translated into the Java representation.

For now, we do not consider multiple CVMs; therefore, no CVM name is needed, and all initial objects must, necessarily, be created in the single CVM. We define classes InitObject and InitObjectList to be able to represent the set of initial objects as a Java object. Similarly, we define a class ClassDefininitionSet to store the class definitions. At last we define an interface CreolProgram with methods *getClassDefinitions*() and *getInitObjects*(). A Creol program is translated into an object of a class which implements CreolProgram. For example, we translate the bounded buffer example from Chapter 2 into a Java class BoundedBuffer; see Figure 4.2. An instance of this class is created:

CreolProgram creolprogram = **new** BoundedBuffer()

The object `creolprogram` refers to is now a Java object representation of the Creol program Bounded Buffer, and can be given as a program to the Creol Virtual Machine. We have omitted the code where class definitions are added to the `classdef` object, as it is low-level details; the interested reader is referred to Appendix B. The class definition set is an object with a mapping from class names to objects of class CreolClass. The CreolClass object has, among other things, a mapping from method names to CreolMethod objects. Statements are represented by instances of classes which implements the interface Statement. In these classes, the semantics of the statement is defined by methods; e.g., *execute*(), *enabled*() and *ready*(). The statements are concatenated to a statement list by a next pointer, that is, the objects will have a pointer to the next statement (or next statements in case of branching statements as the *if* statement). The functional part of Creol is also represented as Java objects. For integers, booleans and strings we have classes IntVal, BoolVal and StrVal. Objects of these classes work as wrappers for the corresponding Java types and new instances are created when needed; e.g., new IntVal(2), new BoolVal(true) and new StrVal("HelloWorld"). These

```
public class CVM {
  public CVM(CreolProgram cp) {

    /* Create central */
    Central central = new Central(cp.getClassDefinitions());

    /* Create Initial objects */
    InitObject io;
    InitObjectList iol = cp.getInitObjects();

    while(iol != null) {
      io = iol.first();
      central.newObject(io.getClassname(), io.getParameters());
      iol = iol.rest();
    }
  }
}
```

Figure 4.3: The class CVM.

classes will implement the interface Data. For compound expressions, as integer addition, boolean tests and object identifiers equality, we have corresponding classes. All expressions have a method *evaluate*().

Note: Later, we will extend the implementation to support multiple inheritance (Chapter 5) and communication between CVMs (Chapter 6). Then we will need the subtype graph and a CVM-name to Internet-address mapping. The CreolProgram interface and the classes which implements CreolProgram will then be extended with necessary methods.

4.3.2 Initialization of the CVM

As previously mentioned, the class CVM has the responsibility to initialize the virtual machine. As in the model (Chapter 3, Section 3.4.1), a central is created. As the service "New object" is invoked by method calls and not messages, the creation of initial objects is a bit different from the model; the CVM will call the *newObject*() method for each initial object. Figure 4.3 gives the class definition of CVM. The constructor takes as argument a CreolProgram reference. Assume that creolprogram is a reference to an object of a class which implements CreolProgram. Then, a new Creol virtual machine is created by the statement

new CVM(creolprogram)

and the program defined by the creolprogram object is executed.

```
1  public class Central implements CreolObjectServices,
                                  MsgQueueServices {

       private ClassDefinitionSet classdef;
5      private HashMap<ObjVal,MsgQueue> inqueues;
       private int objCnt;
       private ReadWriteLock classdefLock, inqueuesLock, objCntLock;

       public Central(ClassDefinitionSet classdef) { ... }
10
       public ObjVal newObject(String classname,
                               DataList actualParameters) { ... }

       public CreolMethod getMethodDef(String methodname) { ... }
15
       public MsgQueue getQueue(ObjVal ov) { ... }
   }
```

Figure 4.4: The class Central (private methods are omitted).

4.3.3 The Central

The Central supports three services: create a new object, get the method definition and get the in-queue of an object. Therefore, the Central needs to store the class definitions, the Creol objects' in-queues, and a counter used to give unique object identifiers to objects; therefore, we will have attributes classdef, in-queues and objCnt. In addition, we use locks to ensure exclusive write-access: one read-write lock for each of the three mentioned attributes.[5] Thus, we allow multiple readers. At initialization, the locks are declared to be fair; this is to prevent starvation. All attributes are declared private, such that they are not visible to other objects. The attributes are initialized in the constructor of Central. An outline of the Central class is given in Figure 4.4.

Services offered to Creol objects

The Central offers two services to the Creol objects, which both will be implemented by method calls: *newObject*() and *getMethodDefinition*(). As previously mentioned, these methods are declared in an interface CreolObjectServices to restrict the Creol objects from accessing other services.

[5] The read-write lock is a *reentrant* lock, that is, a thread which has the lock succeeds in locking it again. This has no influence here; the use of reentrant lock is simply because this is what the Java API offers.

```
1  public ObjVal newObject(String classname, DataList actualParam) {
       ObjVal id = freshId(classname);

       MsgQueue inQ = new MsgQueue();
5      MsgQueueOut outQ = new ForwardingMsgQueue(this);
       inqueuesLock.writeLock().lock();
       inqueues.put(id, inQ);
       inqueuesLock.writeLock().unlock();

10     VarSet attributes = new VarSet();
       attributes.put("this", id);
       attributes.put("sys_class", new StrVal(classname));

       classdefLock.readLock().lock();
15     CreolClass creolclass = classdef.getClass(classname);
       declareParameters(attributes, creolclass, actualParam);
       declareAttributes(attributes, creolclass);
       classdefLock.readLock().unlock();

20     CreolObject co = new CreolObject(attributes,this,inQ,outQ);
       Thread t = new Thread(co);
       t.start();
       return id;
   }
```

Figure 4.5: The method newObject() in class Central.

The CVM objects will have a reference to the Central, and call these methods to invoke the services; i.e., central.newObject(...). This way of implementing the Central's services differs from how it is done in the CCM. Recall that in the CCM, the invocation of these services is done by sending messages to the Central:

> `newObj`(*obj*, *classname*, *actual parameters*) and
> `getMethodDef`(*obj*, *class*, *method*, *sig*, *co*)

This corresponds to the method calls in the implementation. The return of the method calls corresponds to the answer messages from the central:

> `newObjId`(*obj*, *id*) and
> `methodDef`(*obj*, *par list*, *ret var*, *code*)

As Java methods only return one value, the *getMethodDef*() method will return a reference to a CreolMethod object, which contains the formal parameter list, return parameter list and the code.

The *newObject*() method is given in Figure 4.5. It implements Rule 3.1 from Chapter 3. First, a fresh identifier is created for the object (wrapped

```
1  public CreolMethod getMethodDef(String classname, String mtdname) {
       classdefLock.readLock().lock();
       CreolClass cc = classdef.getClass(classname);
       CreolMethod cm = cc.getMethod(mtdname);
5      classdefLock.readLock().unlock();
       return cm;
   }
```

Figure 4.6: The method getMethodDef() in class Central.

in a ObjVal object). Then, in- and out-queues are created for the Creol object, and the in-queue is stored in the inqueues-attribute of Central. Note that a reference to the Central is passed as a parameter to the out-queue; message queues are discussed in Section 4.3.5.

A variable set to store the Creol object's persistent state variables is created. The object identifier is stored as a read-only variable this, and the object's class is stored as a special system variable sys_class. The class definition is fetched and the parameters and attributes are declared by two private methods *declareParameters*() and *declareAttributes*(). For now, these methods are rather simple; when multiple inheritance is introduced (Chapter 5), things becomes more difficult. We postpone the discussion of these methods to Chapter 5.

At last the Creol object is created. In addition to the in-queue, out-queue and the attribute set, a reference to the Central is passed as arguments to the CreolObject class. An execution thread is created for the Creol object and the thread is started. The reference to the object is returned. The Creol object is discussed in Section 4.3.4.

The *getMethodDef*() method is straight forward as we do not consider inheritance; see Figure 4.6. It implements Rule 3.2 from Chapter 3.

Service offered to message queues

The Central has a method *getQueue*() which are used by out-queues to forward messages to the intended in-queues. The method is defined in the interface MsgQueueServices. It is implemented by a simple look-up in the in-queue register; see Figure 4.7. As the in-queue register is a shared object, the look-up is protected by a lock. This lock is declared as fair; therefore, this method will eventually succeed in getting the lock (and therefore it will eventually return). Creol objects can only send messages to objects after they are created. The in-queue of an object is created and inserted in the in-queue register before the object itself is created; therefore, we are guaranteed that the queue exists.

```
public MsgQueue getQueue(ObjVal ov) {
    inqueuesLock.readLock().lock();
    MsgQueue msgQ = inqueues.get(ov);
    inqueuesLock.readLock().unlock();
    return msgQ;
}
```

Figure 4.7: The method getQueue() in class Central.

4.3.4 The Creol Object

In the model (Section 3.3.4), the object is represented as a tuple:

⟨ ID, CLASS, IN-QUEUE, OUT-QUEUE, ATTRIBUTES,
COMPLETION SET, ACTIVE PROCESS,
SUSPENDED PROCESSES, STATUS ⟩

Most of the components translate into attributes of the class CreolObject. However, due to implementation issues, we will make some changes. The object identifier is stored as a special read-only attribute this, and the class is stored as a special system attribute sys_class. This is done to reduce the number of attributes.[6] The status flag was used in the model to enforce a more deterministic computation of the programs; in the Java implementation there is no need for such a flag, as the execution (within the object) is deterministic.

The rest of the Creol object components are attributes in the CreolObject class: references to the Central, the in-queue, the out-queue, the attributes, the completion set, the active process and the queue of suspended processes; see Figure 4.8. The queues and the attributes are created by the Central, whereas the completion set and the queue of suspended processes are created in the constructor of CreolObject. The reference to the active process is set to null by the constructor. If the (Creol) method run exists, a new process instance of this method is created in the (Java) method CreolObject.run(); the attribute activeProcess is set to refer to this process. The Creol object also needs two more attributes: processCnt and labelcount; these are used to give unique values to identify processes and message invocation messages, respectively.

[6] All our system variables which have nothing to do with the Creol program are given the prefix 'sys_' to distinguish them easier. Identifiers beginning with sys_ are assumed not to be used by Creol programs.

```
1  public class CreolObject implements Runnable {
       private CreolObjectServices central;
       private VarSet attributes;
       private MsgQueue inqueue, outqueue;
5      private CompSet compset;
       private ProcessQueue processQueue;
       private CreolProcess activeProcess;
       private int processCnt;
       private LabelCount labelcount;
10
       public CreolObject(VarSet attr, CreolObjectServices central, ↩
           ↪MsgQueueInOut inqueue, MsgQueueOut outqueue) { ... }

       public void run() { ... }
       private void controller() { ... }
15     private void processScheduling() { ... }
       private void processExecution() { ... }
       private void waitForMessage() { ... }
       private void messageProcessing() { ... }
       private CreolProcess newProcess(InvocMsg msg) { ... }
20     private String processId(String methodname) { ... }
   }
```

Figure 4.8: The class CreolObject.

Object tasks

Recall from Section 3.4.3 that the object performs three tasks: message processing, process scheduling, and process execution. In addition there is the 'task' of waiting for a message. Each of these tasks is implemented by private methods of the class CreolObject:

- *processScheduling*()
- *processExecution*()
- *messageProcessing*()
- *waitForMessage*()

The top level method that calls these tasks is called *controller*(). We now discuss *controller*() and the four methods it calls.

The method *controller*()

The *controller*() method in the class CreolObject controls the activity in the object, by calling the methods *processScheduling*(), *processExecution*(), *messageProcessing*() and *waitForMessage*() at the appropriate time

```
private void controller() {
    while(true) {
        processScheduling();

        if(activeProcess != null && activeProcess.ready())
            processExecuting();
        else
            waitForMessage();

        messageProcessing();
    }
}
```

Figure 4.9: The controller loop of the object.

and in the right order. As previously mentioned, these methods are closely related to the status flag in the model. The *controller*() method must call the methods such that it observes the rules 3.3 to 3.7 in Chapter 3. When the object has finished a task, the method corresponding to the next task is called; i.e., *processExecution*() is called after *processScheduling*(), corresponding to Rule 3.5. A new object starts with the task of process scheduling (Section 3.4.2). Hence the first method to call is *processScheduling*(). The next method to run depends of the result of the scheduling. If it was able to select a ready process, the next method to run is *processExecution*(); this corresponds to Rule 3.5. If it was *not* able to select a ready process, the next method is *waitForMessage*(), corresponding to Rule 3.6. After *processExecution*() or *waitForMessage*() returns, the next method is *messageProcessing*() (rules 3.3 and 3.7), before *processScheduling*() is called again (Rule 3.4). Summarized, the algorithm to implement is:

1. Start with method *processScheduling*().

2a. If a ready process was selected, that is, the active process is different from `null` and it is ready, call *processExecution*().

2b. If a ready process was *not* selected, call *waitForMessage*().

3. When *processExecution*() or *waitForMessage*() terminates, call *messageProcessing*().

4. When *messageProcessing*() returns, go to 1.

The transitions are implemented by using an endless while-loop and an if statement; see Figure 4.9.

```
1  private void messageProcessing() {
       Message msg;
       while(inqueue.hasMsg()) {
           msg = inqueue.nextMsg();
5          if (msg instanceof InvocMsg) {
               CreolProcess p = newProcess((InvocMsg) msg);
               processQueue.insert(p);

           }
10         else if (msg instanceof CompMsg) {
               CompMsg compmsg = (CompMsg) msg;
               compset.insert(compmsg.getLabel(),compmsg.getRetVal());
           }
           else { /* Not supposed to happen */ }
15     }
   }
```

Figure 4.10: The message processing in the object.

Message processing

Message processing is described in Section 3.4.4 in Chapter 3. The message queue has methods *hasMsg*() and *nextMsg*(). These are used to process all messages currently in the in-queue of the object.

There are two types of messages to process: invocation messages and completion messages. An invocation message is an object of class InvocMsg, a completion message is an object of class CompMsg. Java has an infix operator *instanceof* which checks if an object is of a specified class; this is used to figure out the message type. The *messageProcessing*() method is given in Figure 4.10. Invocation messages give rise to new processes; we define a method *newProcess*() which creates the new process in accordance with the invocation message. Processes are discussed below. The return values of method calls, given in completion messages, are inserted into the completion set.

Note that the *messageProcessing*() method terminates when the in-queue has no messages. This is in accordance with the precondition of Rule 3.4: when there is no more messages in the in-queue the object goes from message processing to process scheduling.

Wait for message

In the model, there are no rules describing what the object should do when it has the status “Wait for message”; this is quite natural as there is nothing to do. The object goes from status “Wait for message” to status “Message processing” when the in-queue is non-empty. The method

waitForMessage() will therefore wait until this condition is satisfied, and then it will return. This is implemented by introducing a method *waitForMsg*() in the class MsgQueue.[7] The class MsgQueue uses a condition variable msgArrived to wait for and signal the arrival of a message. The use of signals is imperative; without it the object would need to poll the in-queue for messages. This would lead to an inefficient solution with "busy waiting".

Process scheduling

The process scheduling, described in Section 3.4.5 in Chapter 3, is implemented by a method *processScheduling*().

Recall that the object has an attribute processQueue of class ProcessQueue. The process queue and its methods are important in the algorithm for choosing which process to execute. The methods are:

- *insert*(CreolProcess p): Inserts the process at the end of the queue.
- *nextReady*(): Returns the next *ready* process.
- *nextReady*(ObjVal caller, LabelSet labelSet): If there exists a process with the given caller and a label within the given label set, the first such process is returned.
- *getProcess*(String id): If the process identified by id exists, this process is returned. If not (that is, it has terminated), null is returned.

Figure 4.11 shows the whole scheduling algorithm in Java code. The first block of code (lines 4–9) suspends the active process if it is not ready. This corresponds to Rule 3.15. This rule says that the wait variable should be updated to true; therefore, sys_wait is set to true. The active process is inserted in the process queue.

The next block of code (lines 12–20) addresses the case of a terminated process. In the model this is taken care of by the rules 3.13 and 3.14. Rule 3.14 says that if the dynamic link is set, the process identified by this link is to be selected as the active process. Thus the call *getProcess*(dynamiclink) on line 14. Note that this call is made even if the process has terminated; in this case ProcessQueue.*getProcess*() returns null, and the active process is set to null corresponding to Rule 3.13. If

[7] We have chosen to have a method *waitForMsg*() in the queue. Another solution is to use the method *nextMsg*() which waits for a message to arrive and then returns the message. The use of *nextMsg*() implies that the message returned most be handled in the Creol object's *waitForMessage*() method, that is, we would not follow the model.

```
private void processScheduling() {

    /* Suspending the active process. */
    if(activeProcess != null &&
       !activeProcess.enabled() && !activeProcess.terminated()) {
        activeProcess.procInfo.lvar.put("sys_wait", new BoolVal(↩
            ↪true));
        processQueue.insert(activeProcess);
        activeProcess = null;
    }

    /* Terminated process */
    if(activeProcess != null && activeProcess.terminated()) {
        ObjVal dynamiclink = (ObjVal) activeProcess.procInfo.lvar.↩
            ↪get("sys_dynamiclink");
        if(dynamiclink != null) {
            activeProcess = processQueue.getProcess(dynamiclink);
        }
        else {
            activeProcess = null;
        }
    }

    /* Activation of local call */
    if(activeProcess != null && activeProcess.enabled() &&
       !activeProcess.ready() && activeProcess.waitsFor() != null){
        CreolProcess childProcess = processQueue.nextReady((ObjVal)↩
            ↪ attributes.get("this"), activeProcess.waitsFor());

        if(childProcess != null) {
            ObjVal ov = (ObjVal) activeProcess.procInfo.lvar.get("↩
                ↪sys_id");
            childProcess.procInfo.lvar.put("sys_dynamiclink", ov);
            processQueue.insert(activeProcess);
            activeProcess = childProcess;
        }
    }

    /* Activating a suspended process */
    if(activeProcess == null) {
        activeProcess = processQueue.nextReady();
    }
}
```

Figure 4.11: The scheduling of processes in the object.

```
1 private void processExecution() {

      while(activeProcess.ready())
          activeProcess.execute();
5 }
```

Figure 4.12: The execution of processes in the object.

the dynamic link is not set, the active process is also set to null; this corresponds to Rule 3.13.

The activation of local calls is a bit tricky. It is handled by the third block of code (lines 23–33). It corresponds to Rule 3.12. The method CreolProcess.*waitsFor*() is similar to the definition of the function *waitsFor* in the model (Section 3.4.5), except that only label values identifying *local* method invocations are returned. Thus it is not necessary to check if the call is local (as the function *isWaitingFor* does). If the *nextReady*() method succeeds in selecting a ready process, the dynamic link of this process is set to the active process. The active process is suspended and the new ready process is set as the active process.

The last block of code (lines 36–38) handles activation of a suspended process. This is done only if the active process is null; this corresponds to Rule 3.11.

Note that when the method CreolObject.*processScheduling*() terminates, one of the precondition of Rule 3.5 or Rule 3.6 is satisfied. Hence the task of process scheduling is finished.

Process execution

Section 3.4.6 describes process execution. The method *processExecution*() in the class CreolObject implements process execution, given by the rules 3.16–3.37. The method is given in Figure 4.12. To execute processes, or more precisely, execute process statements, *processExecution*() calls the method CreolProcess.*execute*(), which executes one statement. This method must only be called when the process is ready. Therefore, we use a while-loop and check if it is possible to execute a statement by the call `activeProcess.ready()`.

Processes

Recall the representation of a process in CCM:

⟨ LOCAL VARIABLES, ACTIVE CODE, SUSPENDED CODE, ID, DYNAMICLINK ⟩

The components ID and DYNAMICLINK are translated into local variables sys_id and sys_dynamiclink; this is done to reduce the number of attributes in the class CreolProcess.

The statement is executed by calling the method *execute*() for the given statement. A statement needs access to components of the object and the process. This has an impact on how we organize things: we collect all components necessary in an object procInfo of class ProcessInformation. Then, when a statement s is to be executed, s.*execute*(procInfo) is called. The class ProcessInformation has attributes corresponding to the local variables, the object attributes, the in-queue, the out-queue, the suspended code, the completion set, and the Central. In addition it has a pointer to an object which gives unique label values, and an attribute where all local label values are stored. The latter mentioned is used to determine if a call was local, used by the method CreolProcess.*waitsFor*().

In Chapter 3, we defined some functions for the process. In the implementation, we have corresponding methods (without side-effects):

- *enabled*()
- *ready*()
- *terminated*()
- *waitsFor*()

The first three methods are straight forward implementations of the definitions given in Section 3.4.3 and Section 3.4.5. The last method, *waitsFor*(), is slightly different from the function *waitsFor* defined in Section 3.4.5 — it returns only label values identifying local calls. The last method of the class CreolProcess is the most interesting one: *execute*(). It is described in the next section.

The method CreolProcess.*execute*()

The method CreolProcess.*execute*() implements the execution of a single statement, the activation of suspended code, and the suspending of not enabled code (described in Chapter 3, Section 3.4.6). For a given statement type, we have a Java class which represents the statement. This way we can define the appropriate effect the statement has by a method *execute*() in the corresponding class. All classes representing a statement implements the interface Statement; this is necessary to be able to treat all types of statements as a common type.

```
public void execute() {
    if(activeCode == null) {
        activeCode = procInfo.suspendedCode.getReady(procInfo);
    }

    activeCode = activeCode.execute(procInfo);

    if(activeCode != null && !activeCode.enabled(procInfo)) {
        procInfo.suspendedCode.insert(activeCode);
        activeCode = null;
    }
}
```

Figure 4.13: The execute() method in class CreolProcess.

```
public interface Statement {
    CreolCode execute(ProcessInformation pi);
    boolean enabled(ProcessInformation pi);
    boolean ready(ProcessInformation pi);
    LabelSet localCalls(ProcessInformation pi);
}
```

Figure 4.14: The interface Statement.

The Java code for Process.*execute*() is given in Figure 4.13. When active-Process.execute() is called, *execute*() is called for the current statement. The *execute*() method returns the next statement to be executed; hence we have

activeCode = activeCode.execute(procInfo)

The process' code consists of a *set* of statement lists, that is, the active code and the suspended code. Any of these statement lists may be ready. Therefore, the class SuspendedCode has a method *getReady*() which is called if the attribute `activeCode` is `null` (lines 2-4); this corresponds to Rule 3.16. Similarly, when the first statement of the active code is executed, if the next is not enabled, the active code is suspended (lines 8-11); this corresponds to Rule 3.17.

Next we look at some details for the statements.

The method Statement.*execute*()

In our implementation, the active code attribute is a reference to an object of a class implementing the interface Statement (see Figure 4.14).

```
1  public class AssignmentList implements Statement {
       VarList variables;
       ExprList el;
       Statement next;
5
       public CreolCode execute(ProcessInformation pi) {
           VarList vl = variables;
           DataList dl = el.evaluate(pi);
           while(vl != null) {
10             if(pi.lvar.has(vl.first()))
                    pi.lvar.put(vl.first(), dl.first());
               else
                   pi.att.put(vl.first(), dl.first());
               vl = vl.rest();
15             dl = dl.rest();
           }
           return next;
       }
       ...
20 }
```

Figure 4.15: The execution of the multiple assignment statement.

The methods *enabled*(), *ready*() and *localCalls*() are straight forward implementations of the corresponding functions defined in Section 3.4.3 and Section 3.4.5.

The multiple assignment statement `V := E` for a variable list `V` and an expression list `E` is implemented by a class AssignmentList. The *execute*() method of this class is given in Figure 4.15; it corresponds to Rule 3.19. Note that the expression list is evaluated *before* each of the variable is assigned the corresponding value, and that the `sys_wait` attribute is updated to `false`.

The conditional *if*-statement `if b then` S_1 `else` S_2 `fi` for a boolean expression `b` and statement lists S_1 and S_2 is implemented by a class IfThenElse. The *execute*() method of this class is given in Figure 4.16. Recall that statements are also statement lists, as statements have a next-pointer. As for the assignment statement, the `sys_wait` attribute is updated to `false`. Then the boolean expression `b` is evaluated, and the appropriate statement (list) is returned (Rule 3.21 and Rule 3.22).

4.3.5 Messages and Message Transportation

The Creol objects communicate by asynchronous method calls, modeled by sending invocation and completion messages. The messages are objects of class InvocMsg or CompMsg, both implementing the interface

```
1  public class IfThenElse implements Statement {
       BoolExpr be;
       Statement ifStatement, elseStatement;

5      public CreolCode execute(ProcessInformation pi) {
           BoolVal res = be.evaluate(pi);
           pi.lvar.put("sys_wait", new BoolVal(false));
           if(res.getValue())
               return ifStatement;
10         else
               return elseStatement;
       }
       ...
   }
```

Figure 4.16: The execution of the *if*-statement.

Message and a method *getDestination*(). This method returns the object identifier of the destination Creol object.

As previously discussed, the object must be able to treat its in-queue as an out-queue, that is, it must be able to insert messages in its in-queue. Therefore, the in-queue is implemented by a class MsgQueue which implements MsgQueueInOut; the code is given in Figure 4.17.

The messages are stored in a linked list of messages. We use a lock to protect the in-queue against interference and a condition variable to signal the arrival of a message. The lock is declared to be fair to prevent starvation. The *nextMsg*() and *waitForMsg*() methods wait for a message to arrive; the call msgArrived.await() call is done in a loop and within a try block because of special Java properties; see Section 2.2.

The transportation of messages between objects will be taken care of by the Creol object's out-queue, implemented by a class ForwardingMsg-Queue; see Figure 4.18. The class has one method: *insert*(). When called, this method uses the message queue services object to get a reference to the in-queue of the Creol object for which the message is addressed. Note that in the class ForwardingMsgQueue, no persistent object variables are modified; hence, there is no need for synchronization and therefore no locks. Also, the out-queue is only accessed by the Creol object, so there would be no risk of interference even if some variable was modified.

```
1  public class MsgQueue implements MsgQueueInOut {
       private LinkedList<Message> queue;
       protected Lock lock;
       protected Condition msgArrived;
5
       public MsgQueue() {
           queue = new LinkedList<Message>();
           lock = new ReentrantLock(true);
           msgArrived = lock.newCondition();
10     }

       public Message nextMsg() {
           lock.lock();
           try {
15             while(queue.size() == 0) msgArrived.await();
           } catch (InterruptedException ie) {}
           Message msg = queue.remove();
           lock.unlock();
           return msg;
20     }

       public void insert(Message msg) {
           lock.lock();
           queue.addLast(msg);
25         msgArrived.signal();
           lock.unlock();
       }

       public boolean hasMsg() {
30         lock.lock();
           boolean answer = queue.size() > 0;
           lock.unlock();
           return answer;
       }
35
       public void waitForMsg() {
           lock.lock();
           try {
               while(queue.size() == 0) msgArrived.await();
40         } catch (InterruptedException ie) {}
           lock.unlock();
           return;
       }
   }
```

Figure 4.17: The class MsgQueue.

```
public class ForwardingMsgQueue implements MsgQueueOut {
    private MsgQueueServices mqs;

    public ForwardingMsgQueue2(MsgQueueServices mqs) {
        this.mqs = mqs;
    }

    public void insert(Message msg) {
        MsgQueue msgQ = mqs.getQueue(msg.getDestination());
        msgQ.insert(msg);
    }
}
```

Figure 4.18: The class ForwardingMsgQueue.

4.4 Example Run: The Santa Claus Problem

We test our implementation by coding the Santa Claus problem in the Java representation. We do not have any notion of a console or a 'print' statement in Creol, and we do not get any output from the programs. Therefore, we define a new statement print *expr* where *expr* is an expression over strings and variables (here, + is used to concatenate expressions). We use Java's System.out.println() statement to write to the console.

We are only interested in what Santa Claus is doing; hence, we translate the pseudo code «Pick up and deliver toys»; to print "Santa Claus delivers toys."; and «Talk to elves»; to print "Santa Claus talks to elves: " + inoffice_elves; A run gives the following output on the console:

```
$ java SantaClauseProblem
Santa Claus delivers toys.
Santa Claus talks to elves: (cvm,11), (cvm,12), (cvm,14)
Santa Claus delivers toys.
Santa Claus delivers toys.
Santa Claus delivers toys.
Santa Claus talks to elves: (cvm,13), (cvm,15), (cvm,11)
Santa Claus delivers toys.
Santa Claus delivers toys.
Santa Claus delivers toys.
Santa Claus talks to elves: (cvm,12), (cvm,14), (cvm,13)
Santa Claus delivers toys.
  .
  .
```

The (cvm,11), (cvm,12), etc. are object identifiers for elves.

4.5 Summary

In this chapter we have presented an implementation of the Creol virtual machine. Java was chosen as the implementation language because it has some properties that appealed to us: it is object-oriented, multi-threaded, has network capabilities and is in widespread use. However, we made some assumptions about the Java virtual machines to take full advantages of multi-threading. We argued for using one execution thread for each Creol object, and also for using this execution thread for message transportation. We defined a number of classes and interfaces in a way that supported encapsulation and hiding of data. The design follows the model to a large extent, and most of the implementation was straight-forward; however, we had to put a lot of emphasize on synchronization to achieve safe and efficient communication between the Creol objects and the Central, and between message queues. Finally, we gave an example run of the Santa Claus problem.

Chapter 5

Multiple Inheritance

In this chapter we present multiple inheritance in the Creol language and discuss how to extend the model and the CVM implementation with multiple inheritance. We have inheritance both at the interface and class level; here, only inheritance at the class level is discussed. First, we take a look at multiple inheritance in general.

A class C inherits from another class B if B is an *ancestor* class of C; the ancestor class can be either direct or indirect. With *single* inheritance, a class is derived from at most one direct ancestor class (Figure 5.1a), whereas for *multiple* inheritance there may be several direct ancestors (Figure 5.1b). An ancestor B of a class C is called a *superclass* of C; C is called a *subclass* of B.

Class level inheritance facilitates code reuse: a class inherits the methods and attributes of its superclasses. Methods may share the same name and the signature may overlap, in which case which method definition to use must be determined. Likewise, there may be attribute naming conflicts. For single inheritance, the inheritance graph defines a total ordering[1] of the inherited classes. This ordering can be used to determine which method definition or attribute to use. With multiple inheritance, the inheritance graph defines only a *partial* ordering[2]; therefore, the conflicts are more difficult to resolve.

The class of an object is not always statically known. Therefore, many programming languages bind methods virtually; a method is *virtually bound* if the body corresponding to a method invocation is selected at run-time. In Creol, external calls to objects typed by interfaces are always virtually bound.

[1] An ordering R over a set S is *total* if all elements in S are comparable, that is, for all c_1 and c_2 in S, $R(c_1,c_2)$ can be determined.

[2] An ordering R over a set S is *partial* if there may be elements in S which is not comparable.

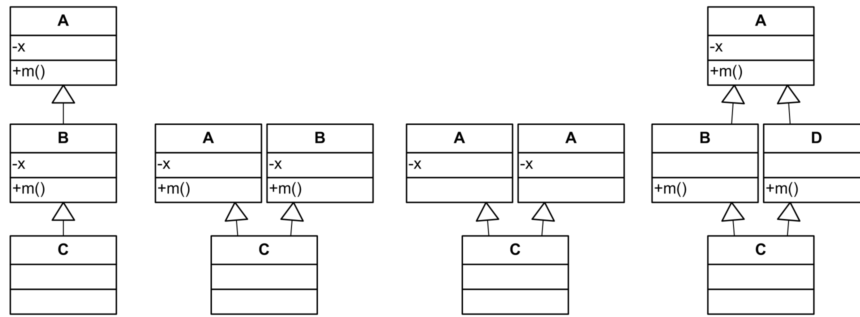

Figure 5.1: Examples of class inheritance: a) single inheritance, b) multiple inheritance, c) duplicate inheritance, and d) a common ancestor in the inheritance graph. `A`, `B`, `C` and `D` denotes class names, `m` a method and `x` an attribute.

We identify four problems (or challenges) with multiple inheritance which must be resolved:

Method ambiguities: If a method is inherited from two or more superclasses which are not in the same inheritance path (Figure 5.1b), how should we determine which method definition to use?

Duplicate inheritance: If a class is inherited more than once, either directly (Figure 5.1c) or indirectly (Figure 5.1d), should there be one or more instances of the attributes in the object?

Invocation of a superclass' methods: How can a method be called from a method in a subclass, possibly a redefinition of the method?

Attribute naming conflicts: How should attribute naming conflicts be resolved, such that superclasses' attributes can be accessed even if there are naming conflicts?

C++ allows multiple inheritance at the class level [30]. Ambiguities are removed by explicit resolution. The object gets multiple copies of the attributes if a class is inherited more than once. Java [12] and C# [11] do not support multiple inheritance at the class level and therefore avoid the first two problems. In C++, the class names are used to access superclass' methods and attributes. The inheritance graph can be followed to the intended class; e.g., `Class1::Class2::method()`. In Java, only the methods of the *direct* ancestor of a class can be called; this is done by using the keyword `super`; e.g., `super.m()`. Similarly, attributes can be accessed by, e.g., `super.a`.

In the next section, we will see how potential problems with multiple inheritance are solved in the Creol language.

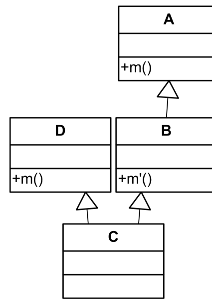

Figure 5.2: An unintended method definition can be selected due to same method name and signature in two unrelated classes.

5.1 Creol and Multiple Inheritance

Creol supports multiple inheritance at both the interface and class level. Interfaces define the types of objects; therefore, interface inheritance define subtypes. Interface inheritance is rather simple: a set union of all the methods of the inherited classes. Multiple inheritance at the class level imposes some challenges. We will now discuss how the challenges encountered in the previous section are solved in the Creol language. The presentation is based on [17, 21].

Methods may have the same name but different signature or cointerface. Consequently, in addition to the method name, the bounded method must also match the signature and cointerface. In the following discussion we will ignore the signatures and cointerfaces of method calls; this is to keep the presentation as simple as possible.

Method binding

In Creol, we do not require explicit resolution of ambiguities, nor do we define any ordering of the classes (besides the ordering defined by the inheritance graph). Methods are bound virtually, and which method body to bind is selected non-deterministically among the candidates. However, we need a restriction to avoid unintended methods to become candidates.

Consider this example: A class B has a method *m'* which invokes a method *m*; see Figure 5.2. The method *m* exists in class A, a superclass of B. Obviously, A.*m* or a redefinition of this method is supposed to be called; however, we have no guarantee that the method which is bound actually is a redefinition: Assume we have a class C which inherits A, B and a unrelated class D, and that class D also has a method *m*. As

methods are bound at run-time based on the actual class of an object, we risk that D.*m* is bound to the call, which was not intended. Creol avoids this problem by the *pruned binding strategy*. To explain it, we need two predicates over classes:

- A class A is *above* a class C if A is the same class as C or if A is a superclass of C.
- A class A is *below* a class C if A is the same class as C or if A is a subclass of C.

For a method call *m*, type checking guarantees that there is a matching method definition in some class A *above* the class in which the method call is made. The class A is thus identified at compile time. The actual binding is restricted to a method in a class *below* A; we denote this by *m*<A.

- The *pruned binding strategy* ensures that a method call *m* is only bound to an intended redefinition of *m*. At compile time, a method call *m* in a class C is replaced by *m*<A where A is the statically identified class above C. When executed, the binding of the method call is restricted to methods defined in classes below A.

In Creol, it is possible to call methods in superclasses by using qualified names. A method call to a method *m* in a superclass C is written *m*@C. The exact semantics is that this is a call to a method *m above* C, that is, the class C need not implement *m*, but C or a superclass of C must implement *m*. In our previous example, a method in C might make a call m@B. We use the class names directly; we do not follow the inheritance graph as in, e.g., C++. This works fine because we only get one copy of the class attributes even if a class is inherited more than once (see below).

Note: The invocation of superclass' methods is done explicitly by the programmer, whereas the pruning of method calls is done by the compiler. In our example, the method call m@B would be replaced by m@B<A by the compiler.

Attributes and parameters

If a class is inherited more than once, as for the inheritance graphs c) and d) in Figure 5.1, the object only gets one instance of the attributes of each class; e.g., only one variable with name x for each class. Attributes

```
1  class SAuth
   var gr:Agent=null
   begin
     op grant(in x:Agent) == await gr = null; gr := x
5    op revoke(in x:Agent) == if gr = x then gr := null fi
     op auth(in x:Agent) == await (gr = x)
   end

   class MAuth
10 var gr:Set[Agent]=empty
   begin
     op grant(in x:Agent) == gr := gr U { x }
     op revoke(in x:Agent) == gr := gr \ { x }
     op auth(in x:Agent) == await (x in gr)
15 end
```

Figure 5.3: Single and multiple authorization policies.

in superclasses are accessed by qualified names; e.g., x@A. Thus, there will be no naming conflicts between class attributes.

Note that also parameters (including the parameters of superclasses) give rise to persistent object variables. At object creation, parameter values may be passed down to inherited classes. The attributes are given initial values by expressions over the class' parameters and inherited parameters and attributes.

5.1.1 Example: Combining Authorization Levels

We now present an example to illustrate the use of multiple inheritance in Creol. The example is presented in [17, 21]; however, here we present a slightly different version.

Assume we have a database

```
class DB
begin
  op access(in key:int, high:bool out y:Data) == ...
  op clear(in x:Agent out ok:bool) == ...
end
```

The method access(in key:int, high:bool out y:Data) accesses the database and returns the data associated by integer key. If high is true, more (sensitive) information may be given. The method clear(in x:Agent out ok:bool) checks if an agent x has access to sensitive data.

The database does not implement any interfaces; thus, the methods and thereby the data can not be accessed from outside. The reason for this

is that we want to control the access to the database. We want high clearance agents to have unique access to (classified) data, whereas low clearance agents may share access to unclassified data. To implement this, we first define two classes SAuth and MAuth, used to give single and multiple access to a resource; see Figure 5.3. These classes implement methods *grant*, *revoke* and *auth*, used to grant access to the resource, revoke access and to wait for an agent to be granted access (authorized).

The access to the database is through the interfaces High and Low and implemented by classes HAuth and LAuth, see Figure 5.4. The class HAuth inherits both DB and SAuth, the latter used to enforce exclusive access to the database. As the methods of SAuth are internal and are not offered through any interface, low clearance agents can not call the grant method to bypass the openH method and get access to classified data. The Low interface is similar with methods openL and closeL (openL always succeeds and therefore it does not have an out-parameter as openH).

Combining the authorization policies

We will now use multiple inheritance to combine the authorization policies. We define a class HLAuth which implements both the Low and High interface; see Figure 5.5. An object of the HLAuth class supports both high and low access to data. The *access* method is redefined so that agents which have succeeded in getting high access rights (by the *openH* method) get access to sensitive information while agents which have low access rights (by the *openL* method) only get insensitive information.

Both SAuth and MAuth have an attribute gr. This is no problem, as the *access* method uses a qualified reference gr@SAuth to access gr in SAuth. Qualified names are also used to call the *acc* methods of HAuth and LAuth. The methods of HAuth and LAuth call methods in SAuth and MAuth, respectively. The pruned binding of method calls ensures that the correct methods are called; e.g., the call auth(x) in HAuth is changed to auth<SAuth(x) by the compiler, restricting the *auth* call to be bound to a method below SAuth. Thus, in an instance of class HLAuth, the *auth* in SAuth will be called as SAuth is the only class below SAuth with a method auth. As class attributes are only inherited once even if the class is inherited more than once, there will be only one instance of the data (attributes) in the database.

Note that we allow low level access at the same time as high access is performed. We can give exclusive access to the database to agents with high clearance by redefining the openH and openL methods. High access succeeds when no agents have high nor low access:

```
interface High
begin
 with Agent
  op openH(out ok:Bool)
  op access(in key:int out y:Data)
  op closeH
end

interface Low
begin
 with Agent
  op openL
  op access(in key:int out y:Data)
  op closeL
end

class HAuth implements High inherits SAuth, DB
begin
  op acc(in x:Agent, key:int out y:Data) ==
     auth(x);
     await access@DB(key, true; y)
 with Agent
  op openH(out ok:Bool) ==
     await clear(caller; ok);
     if ok then grant(caller) fi
  op access(in key:int out y:Data) ==
     acc(caller, key; y)
  op closeH == revoke(caller)
end

class LAuth implements Low inherits MAuth, DB
begin
  op acc(in x:Agent, key:int out y:Data) ==
     auth(x);
     await access@DB(key, false; y)
 with Agent
  op openL == grant(caller)
  op access(in key:int out y:Data) ==
     acc(caller, key; y)
  op closeL == revoke(caller)
end
```

Figure 5.4: High and low access to data.

```
1  interface HighLow inherits High, Low
   begin
   end

5  class HLAuth implements HighLow inherits LAuth, HAuth
   begin with Agent
     op access(in key:int out y:Data) ==
        if caller=gr@SAuth
        then acc@HAuth(caller, key; y)
10      else acc@LAuth(caller, key; y)
        fi
   end
```

Figure 5.5: Both high and low access to data.

```
op openH(out ok:bool) ==
   await clear(caller;ok);
   if ok then await gr@MAuth = empty ∧ gr@SAuth = null;
              grant@SAuth(caller)
   fi
```

Similarly, the opening of a low clearance access must wait until there is no high clearance agent accessing the database:

```
op openL == await gr@LAuth = null; grant@MAuth(caller)
```

5.2 Extending the Model

In order to allow multiple inheritance, we must extend the model with some additional information and change some of the computation part.

5.2.1 Changes in the Structure

The changes in the structure consist of adding more information to the method calls an to add a subtype graph to CCM and the Central.

Method calls

External calls must be extended with the signature and cointerface of the call; internal calls must in addition preserve above-constraints and the below-constraints must be added. Recall that in the model (Section 3.3.3), we changed some of the method call statements to minimize the number of different statements. We will do the same here,

and denote missing information by ϵ. The general method call is either $\texttt{t!o.m@C<C'(E)}_{sig,co}$ or $\texttt{!o.m@C<C'(E)}_{sig,co}$. The class names C and C' give the above-constraint and below-constraint, respectively. Method calls with and without labels are very similar; method calls with labels are changed in the following way:

- t!o.m(E) is changed to $\texttt{t!o.m@}\epsilon\texttt{<}\epsilon\texttt{(E)}_{sig,co}$
- t!m@C(E) is changed to $\texttt{t!this.m@C<C'(E)}_{sig,co}$
- t!m(E) is changed to $\texttt{t!this.m@}\epsilon\texttt{<C'(E)}_{sig,co}$

where sig and co are the signature and cointerface of the method call, respectively. The class C' is the class in which the type analysis finds the method m with matching signature and cointerface. The types of the actual in- and out-parameters are determined at compile time and added to the method invocation. Likewise the *cointerface* is determined at compile time and preserved. See [21] for details.

Example: Assume that the method's statement list contains the method call $t!o$.*methodName*(E); t?(V). The types of the expressions in the expression list E and the the variables in the variable list V are given by lists of types T^{in} and T^{out}, respectively. Further, assume that the object in which this call is made is of interface *I*. Then $t!o$.*methodName*(E) is changed to $t!o.\textit{methodName}(E)_{sig,I}$ where sig is T^{in}; T^{out}.

We do not go into more details about how the signature, cointerface and below constraint can be determined, as it is outside the scope of this thesis. However, an important fact is that it will always succeed in any well-typed program [21].

Subtype graph

To check if signatures and cointerfaces match, the Central needs to store the subtype graph. The subtype graph is defined by the *inherits* clause of interfaces and predefined subtypes of data types. In addition, we denote by Data the supertype of all types, and Any the supertype of all interface types. There is one exception: the type Internal is a special interface type which is used in internal calls, that is, the cointerface of internal methods is Internal and internal calls are made by giving Internal as the interface of the caller. Internal has no supertypes.

The model is extended with the subtype graph:

CCM = (CVM names, Initial objects, Class def. set, Subtype graph)

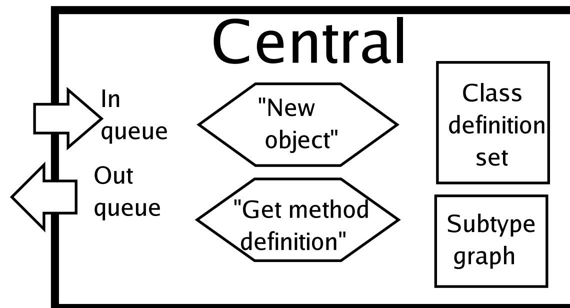

Figure 5.6: The Central is extended with the subtype graph. The services are changed.

Likewise, the Central is extended with the subtype graph:

CENTRAL = ⟨ IN-QUEUE, OUT-QUEUE,
CLASS DEFINITION SET, SUBTYPE GRAPH ⟩

The subtype graph is a set of pairs `<type,supertypes>`, where `supertypes` is a list of types $(t_1,..,t_n)$ where each type t_i is a supertype of `type`. Only the direct supertypes are given, that is, indirectly defined supertypes are determined by traversing the graph. Example:

SUBTYPE GRAPH = { <int,(Data)>, <bool,(Data)>, <nat,(int)>,
<Any,(Data)>, <High,(Any)>, <Low,(Any)>,
<HighLow,(High,Low)>, ...}

It may be more efficient to store all superclasses of a class, but we do not consider efficiency to be important. Furthermore, updates may be easier to handle with our solution. Figure 5.6 presents the extended central.

5.2.2 Changes in the Computation

Recall that we have extended both the model and the Central by the subtype graph. At initialization, the Central's SUBTYPE GRAPH component is set equal to the subtype graph given in the CCM tuple.

We must change the Central's services "New object" and "Get method definition" and how the latter service is invoked. To define these services, we will use functions over the messages received by the Central, the class definition set and the subtype graph. We will use matching of arguments; e.g., to get the first item of a list, we will write `(first, rest)` which will match any non-empty list; in case of a list of a single item, `rest` will be ϵ. We also denote the empty list by ϵ. The functions are close to the actual implementation presented in Section 5.3.

Service: New object

The creation of a new object with and without class inheritance is very similar; the only difference is that with inheritance, we must also instantiate parameters and attributes of superclasses. Recall that a new object is created when the Central receives a message

`newObj`(obj_{from}, classname, actParam)

We now define how to create and initialize the persistent variables of the object; we refer to these as the *attributes*; however, they also include the class parameters. We create a set of pairs `<id,val>` while traversing the inheritance graph, by a function *varSet*:

ATTRIBUTES = *varSet*(classname, actParam, CLASS DEF SET)

The function *varSet* first instantiates the parameters and then the attributes by two functions *instPar* and *instAtt*:

varSet(class, actPar, CD) =
instAtt(*instPar*(∅, <class,actPar>, CD), class, CD)

The two functions *instPar* and *instAtt* are now presented.

Instantiation of the parameters: The function *instPar* traverses a list of pairs `<class-name, expression-list>` and instantiates parameters for each class. The inherited classes (with parameter expressions) are added in front of the list; hence, parameters are instantiated depth-first.

instPar(VAR, ϵ, CD) = VAR
instPar(VAR, (<class,E>,rest), CD) =
instPar(*applyP*(VAR,*par*(class,CD),E), (*inhList*(class,CD), rest), CD)

The function *inhList* returns the inheritance list of the class, whereas *applyP* traverses a list of variables and assigns to each of the variables the corresponding evaluated expression:

applyP(VAR, ϵ, ϵ) = VAR
applyP(VAR, (v,V), (e,E)) = *applyP*(VAR ⊕ {<v,*eval*(VAR,e)>}, V, E)

The function *eval* evaluates an expression in the context of the given variable mapping. The operator ⊕ is similar to a set union operator. It takes two set of pairs `<id,val>` and keeps the left version if both have a pair with same `id`:

$$
\begin{array}{l}
S \oplus \emptyset = S \\
S \oplus \{\langle \text{id,val} \rangle, \text{rest}\} = \\
\quad \textbf{if}(\ \exists\ \text{val' s.t.} \langle \text{id,val'} \rangle \in S\ )\ \textbf{then}\ S \oplus \{\text{rest}\} \\
\quad \textbf{else}\ S \cup \{\langle \text{id,val} \rangle\} \oplus \{\text{rest}\}\ \textbf{fi}
\end{array}
$$

Instantiation of the attributes: The function *instAtt* is similar to *instPar*; however, as attributes are initialized by expressions over inherited class' attributes, superclass' attributes are instantiated first.

$$
\begin{array}{l}
\mathit{instAtt}(\text{VAR}, \epsilon, \text{CD}) = \text{VAR} \\
\mathit{instAtt}(\text{VAR}, (\text{class,rest}), \text{CD}) = \\
\quad \mathit{instAtt}(\mathit{applyA}(\mathit{instAtt}(\text{VAR}, \mathit{inhC}(\text{class,CD}), \text{CD}), \mathit{att}(\text{class,CD})), \\
\qquad\qquad \text{rest, CD})
\end{array}
$$

The class attributes are looked up by a function *att* which simply returns the ATTRIBUTES component of the class; *inhC* returns the inherited classes (without the initial expression). The function *applyA* is almost as *applyP*, except from how the variable names and expressions are given:

$$
\begin{array}{l}
\mathit{applyA}(\text{VAR}, \epsilon) = \text{VAR} \\
\mathit{applyA}(\text{VAR}, (\langle \text{v,e} \rangle, \text{rest})) = \mathit{applyA}(\text{VAR} \oplus \langle \text{v}, \mathit{eval}(\text{VAR,e}) \rangle)
\end{array}
$$

Service: Get method definition

The introduction of multiple inheritance implies that we must make some changes to the statements, the invocation messages, the "get method definition" messages and how the actual look-up of methods is performed.

Recall that methods are invoked by sending invocation messages. The format of this message type must be changed; we need to extend it with the above and below constraints, the signature and the cointerface:

$\texttt{invoc}(obj_{to}, obj_{from}$, label, method, above-class, below-class, sig, co, par)

The rules which concern method calls must be changed (rules 3.25-3.28). In Section 5.2.1 we changed the call statements so that we only have two different call statements:

1) $\mathsf{t!o.m@C{<}C'(E)}_{sig,co}$
2) $\mathsf{!o.m@C{<}C'(E)}_{sig,co}$

The changes to the rules is straight forward: C, C', *sig* and *co* is added to the invocation message; for instance:

invoc(obj_{to}, obj_{from}, label, m, C, C', sig, co, par)

When an object receives an invocation messages, method definitions are fetched by sending a *get method definition* message. As for invocation messages, we must change the format of this message:

getMethodDef(obj, method, above-class, below-class, sig, co)

Note that we have removed the class part of the message, as now the class is defined by the above and below constraints. Invocation messages where the above-class is ϵ, *above-class* is set to the class of the object *obj*. The other components in the *get method definition* message are as in the invocation message. For the example of an invocation message given above, the receiving object obj sends a message

getMethodDef(obj, m, C, C', sig, co)

to the Central (assuming the above-class parameter is not null). Now, when the Central receives this message, it looks up the method definition in the class definition set. To check for a matching signature and cointerface, the subtype graph is also used. The look-up is defined by a function *lookUp*; this function is soon to be defined. The Central answers by a methodDef message:

methodDef(obj, *parList*(M), *retVar*(M), *code*(M))

where

M = *lookUp*(m,sig,co,C',C,ClassDefSet,SubtypeGraph)

The method look-up mechanism (Rule 3.2) is changed to take inheritance into account. We start in the above-class C and search for a method which matches the method name, signature and cointerface, with the constraint that the method must be in a class below C'. As we start to search in C and only search upwards through the class inheritance graph, we are assured that we only visit classes above C. The search algorithm is defined by a recursive function

lookUp(method name, signature, cointerface, below class,
class list, class definition set, subtype graph)

The first three arguments are the method's name, signature and cointerface. The fourth argument gives the class below which the method must be bound. The fifth argument is a list of class names. The search for the method is done by adding inherited classes at the beginning of this list, giving a depth-first search. The two last arguments are the class definition set and the subtype graph, respectively. The function's definition is

lookUp(mname, sig, co, belowClass, (class, rest), CD, SG) =
  if(*isBelow*(class,belowClass,CD) **and**
    $\exists$ M $\in$ *methods*(class,CD) s.t. *match*(mname,sig,co,M,SG))
  then M
  else if(*isBelow*(class,belowClass,CD))
  then *lookUp*(mname,sig,co,belowClass,(*inhC*(class,CD),rest),CD,SG)
  else *lookUp*(mname,sig,co,belowClass,rest,CD,SG)
  fi fi

where the the function *methods* returns the methods of the class, the function *inhC* returns the a list of the inherited classes and the functions *isBelow* and *match* are defined as

isBelow(class1, class2, CD) =
  class2 == ϵ **or** class1 == class2 **or**
  ($\exists$ c $\in$ *inhC*(class1,CD) s.t. *isBelow*(c,class2,CD))

match(mname, sig, co, <MNAME,SIG,CO,..>, SG) =
  mname == MNAME **and** *matchSig*(sig,SIG,SG) **and** *isSubtype*(co,CO,SG)

The function == checks for equality and the functions *matchSig* and *isSubtype* are defined as

matchSig((sig^{in}; sig^{out}), (SIG^{in}; SIG^{out}), SG) =
  length(sig^{in}) == *length*(SIG^{in}) **and** *length*(sig^{out}) == *length*(SIG^{out})
  and $\forall$ i $\in$ *length*(sig^{in}) : *isSubtype*(sig_i^{in}, SIG_i^{in}, SG)
  and $\forall$ i $\in$ *length*(sig^{out}) : *isSubtype*(SIG_i^{out}, sig_i^{out}, SG)

isSubtype(type1, type2, SG) =
  type1 == type2 **or** $\exists$ <type1,(..,t,..)> $\in$ SG s.t. *isSubtype*(t, type2, SG)

The function *length* gives the length of the list.

5.3 Extending the Implementation

In this section, we will show how the Creol Virtual Machine implementation is changed to support multiple inheritance. Most of the changes are straight forward implementations of the functions defined in Section 5.2.

5.3.1 Changes to the Creol Program Representation

The subtype graph is implemented by using a Java Hash-Map with the type as the key and a list of supertypes as the value. The interface `CreolProgram` is extended with a method

```
public SubtypeGraph getSubtypeGraph()
```

Thus, the subtype graph is part of the Java class which represents a Creol program (and implements the `CreolProgram` interface). The class `SubtypeGraph` has a method

```
public boolean isSubtypeOf(String type1, String type2)
```

which implements the function *isSubtype* from the previous section. The subtype graph and its method is used by the `getMethodDef()` method.

We define new classes `Sig` and `Co` which are used to give the signature and cointerface in method calls and method definitions. The class `Sig` has a method

```
public boolean matches(Sig sig, SubtypeGraph sg)
```

which implements the *matchSig* function from the previous section.

The call statement is changed so that the above and below class can be specified (for internal calls) and the signature and cointerface is added. The invocation message is changed accordingly. These changes are straight forward and therefore we will not describe the changes any further.

5.3.2 Changes to the Central's Services

Service: New object

The method `Central.newObject()` is given in Figure 4.5 in Chapter 4. The introduction of inheritance implies that we must declare and initialize the parameters and attributes of superclasses. The methods

```
Central.declareParameters() and
Central.declareAttributes()
```

implements the functions *declPar* and *declAtt*, respectively; see Figure 5.7.

```
private void declareParameters(VarSet att, CreolClass creolclass, ↩
    ↪DataList actualParameters) {
  /* Parameters for this class */
  DataList parValues = actualParameters;
  VarList vl = creolclass.getParameters();
  while(vl != null) {
    att.put(vl.first(), parValues.first());
    vl = vl.rest();
    parValues = parValues.rest();
  }

  /* Parameters for inherited classes */
  InheritsList il = creolclass.getInheritsList();
  Inherits inh;
  while(il != null) {
    inh = il.first();
    if(inh.getParameters() != null)
      declareParameters(att, classdef.getClass(inh.getClassname()), ↩
          ↪inh.getParameters().evaluate(att));
    else
      declareParameters(att, classdef.getClass(inh.getClassname()), ↩
          ↪null);
    il = il.rest();
  }
}

private void declareAttributes(VarSet att, CreolClass creolclass) {
  /* Inherited attributes */
  InheritsList il = creolclass.getInheritsList();
  while(il != null) {
    declareAttributes(att, classdef.getClass(il.first().getClassname↩
        ↪()));
    il = il.rest();
  }

  /* Attributes of this class */
  VdeclList vl = creolclass.getAttributes();
  Vdecl vdecl;
  while(vl != null) {
    vdecl = vl.first();
    if(vdecl.getInitial() != null)
      att.put(vdecl.getName(), vdecl.getInitial().evaluate(att));
    else
      att.put(vdecl.getName(), null);
    vl = vl.rest();
  }
}
```

Figure 5.7: The methods used to create the persistent variables of the object.

```
1  public CreolMethod getMethodDef(String above, String below, String ↩
       ↪methodname, Sig sig, Co co) {
    classdefLock.readLock().lock();
    String classname;
    CreolMethod cm = null;
5   CreolClass cc;
    LinkedList<String> classes = new LinkedList<String>();
    classes.add(above);

    while(classes.size() > 0) {
10    classname = classes.remove();

      if(!isBelow(classname,below)) continue;

      cc = classdef.getClass(classname);
15    cm = cc.getMethod(methodname);
      if(cm != null) {
        if(sig.matches(cm.getSig(), subtypeGraph) && subtypeGraph.↩
            ↪isSubtypeOf(co.getValue(), cm.getCo().getValue())) {
            classdefLock.readLock().unlock();
            return cm;
20      }
      }

      /* Add all inheriated classes to the classes list */
      InheritsList il = cc.getInheritsList();
25    while(il != null) {
        classes.addFirst(il.first().getClassname());
        il = il.rest();
      }
    }
30  return null;
   }
```

Figure 5.8: The method Central.getMethodDef() returns the method definition.

Service: Get method definition

In Chapter 4, the method `Central.getMethodDef()` was a simple look-up of a method. Now, as we have introduced multiple inheritance, this method is rather complicated. The method's arguments are changed: the class name is replaced by an above constraint, and the below-constraint, the signature and the cointerface are added; see Figure 5.8.

We have chosen to use a linked list of strings where we first add the class name given by the *above* argument; inherited classes are added to this list at the end of the method (lines 24-27). The inherited classes are added at the front of the list so that we get a depth first search. The below-constraint is observed by skipping classes which are not below the below-class, that is, a jump is made to the top of the while loop (line 12).

We have assumed that within a class, all method names are unique; therefore, methods are fetched by the method name and the class name (lines 14-15). Then, if the specified class has a method with the requested method name, this method is checked to see if the signature and cointerface match, and if so, the method is returned.

Note that the class definition set is protected by the lock `classdefLock`. As we do not alter the class definitions, we use a read-lock to allow multiple readers to call `Central.getMethodDef()` at the same time. (The reason for protecting the class definitions is to prepare for class definition updates; for class updates a write-lock must be used to enforce unique access.)

5.4 Example Run: Authorization Policies

To test the new look-up mechanism, we use the authorization policies example from Section 5.1.1. Recall that the HLAuth class inherits a class DB and controls the access to the database. We create a small database of customers by implementing the DB class; see Figure 5.9. We have only one customer: Ole Hansen. His customer number is 123, and his sensitive information is his telephone number. For simplicity, all agents are granted high access; still, the result depends on whether openH or openL is used.

To test the database, we make a class TestAgent which contacts the database and asks for information about customer 123 (both high and low access) and customer 456 (only low access). See Figure 5.10.

In the example run, we create an instance of the HLAuth class; say db = new HLAuth(). Further, we create an instance of the TestAgent class. The

```
class DB
begin
  op access(in key:int, high:bool out y:Data) ==
     if key = 123
     then
       if high then
         y := ''123␣is␣customer␣Ole␣Hansen.␣Phone␣no:␣90807060.''
       else
         y := ''123␣is␣customer␣Ole␣Hansen.''
       fi
     else
       y := ''Unknown␣customer.''
     fi

  op clear(in x:Agent out ok:bool) == ok := true
end
```

Figure 5.9: A small customer database.

```
class TestAgent(db:HighLow) implements Agent
begin
  op run ==
     var result:String, ok:bool;
     db.openH(;ok);
     if ok
     then
        db.access(123; result);
        print ''Result␣1:␣'' + result;
        db.closeH;
     fi;

     db.openL;
     db.access(123; result);
     print ''Result␣2:␣'' + result;
     db.access(456; result);
     print ''Result␣3:␣'' + result;
     db.closeL
end
```

Figure 5.10: An agent which tests the database.

object identifier of the database is given as an parameter: agent = new TestAgent(db). The following is the output on the console:

```
$ java AuthorizationPolicies
Result 1: 123 is customer Ole Hansen. Phone no: 90807060.
Result 2: 123 is customer Ole Hansen.
Result 3: Unknown customer.
```

As expected, the given information about customer 123 depends on the access level, and customer 456 is unknown.

5.5 Summary

In this chapter we first took a look at challenges related to multiple inheritance and how these are solved in the Creol language, by the pruned binding strategy for method binding and qualified names for accessing superclass methods and superclass attributes. We have shown how to change the model to support multiple inheritance: we extended the method calls with information about the above- and below-class, the signature, and the cointerface, and we added the subtype graph. The Central's services were changed: parameters and attributes of superclasses are included as new object persistent variables, and a new look-up mechanism is introduced. The implementation was changed the same way, and we tested it with the authorization policies example.

Chapter 6

CVM Intercommunication and Remote Objects

The objects in the Creol language and the communication between these objects are constructed in such a way that the objects can be distributed among different *nodes*; e.g., different machines. However, the Creol language as presented in Chapter 2 has no concept of nodes, machines or virtual machines. Therefore, we need to extend the Creol language to be able to have true distributed objects. In Section 6.1 we extend the Creol language with a notation for virtual machines and remote objects. The new constructs imply that the model and the implementation must be changed; the changes to the model are presented in Section 6.2 and the changes to the implementation are presented in Section 6.3.

6.1 New Creol Language Constructs

We extend the Creol language with a notation for virtual machines and remote objects. We try to incorporate the new constructs as seamlessly as possible, and in such a way that "old" programs don't have to be rewritten but have the same semantics in the extended language.

6.1.1 Virtual Machines

The virtual machines on which we want to distribute the objects must somehow be identified. We choose to give each virtual machine a unique identifier. In addition, we specify the machine name where the virtual machine runs; this is necessary to communicate over a network; e.g., the Internet.

We use a construct similar to the *define* construct in languages such as, e.g., C and C++. The keyword #CVM is used followed by the virtual machine identifier and the machine name of the machine where the virtual machine exists:

#CVM *<cvm identifier> <machine name>*

We may for instance declare a virtual machine cvm1 which exists on the machine einn.ifi.uio.no:

#CVM cvm1 "einn.ifi.uio.no"

The line above specifies that a virtual machine with identifier cvm1 is to be created on the machine with host name einn.ifi.uio.no. The identifier cvm1 can be used to refer to this virtual machine.

Note: On the Internet, it is common to specify a communication port by the host name (or ip address) and a *port number*. We will use a standard port number for all virtual machines to keep things simple and only use the host names. Even so, on each host there may be more than one CVM, as each has its own unique identifier.

6.1.2 Remote Objects

We want the Creol objects to be distributed among different virtual machines. In relation to an object o_1, we say that an object o_2 is *local* if it is on the same virtual machine as o_1 and *remote* if it is on another virtual machine.

We introduce a new statement type "new remote object" into the Creol language. This statement is very similar to the "new object" statement, except that it specifies the virtual machine on which the object is to be created. The new statement is created by appending a "new object" statement with @ followed by the virtual machine identifier. The syntax is

v := new *classname*(E) @ *CVM-id*

where v is an object variable and E an expression list. A new object of class *classname* is created on the Creol virtual machine identified by *CVM-id* and v is assigned the new object's identifier. For instance, the statement prod := new Prod(bb) @ cvm1 creates a new producer on the previously declared virtual machine cvm1, and prod is assigned the object identifier.

Note that there is no difference between the local and remote object identifiers, that is, remote object identifiers are used in just the same way as local object identifiers to invoke methods and can also be sent between objects as parameters. Note also that an object created by the new remote object statement may be local, as the specified virtual machine might be the same as the object on which the statement is executed.

New predicates over object identifiers

As objects are now distributed and these objects' identifiers can be sent between objects, it is possible that an object's actions depend on where other objects are. For instance, if an object has access to multiple databases, a local database is preferred in favor of a remote database.

We introduce predicates *isLocal*, *inCVM* and *inSameCVM* with the following syntax and semantics:

- isLocal(o) for an object identifier o. It returns true if and only if the object identified by o is local, that is, o exists on the same CVM as the object in which the call is made.
- (o inCVM cvm) for an object identifier o and a virtual machine identifier cvm. It returns true if and only if the object identified by o exists on the virtual machine identified by cvm.
- (o_1 inSameCVM o_2) for object identifiers o_1 and o_2. It returns true if and only if the objects identified by o_1 and o_2 exists on the same virtual machine.

Examples: The expression (prod inSameCVM cons) checks if a producer and a consumer object is on the same virtual machine; (prod inCVM cvm1) checks if the producer exists on the cvm1 virtual machine; isLocal(prod) checks if the producer is a local object.

Initial objects

We must somehow define the initial objects of the program and where these objects are to be created. Therefore, we introduce a program initialization statement

#initobject *cvm-id classname*(*parameters*)

```
1  interface DB
   begin
    with Server
     op get(in filename:String out ok:bool, file:File)
5  end

   interface Server
   begin
    with Client
10   op get(in filename:String out ok:bool, file:File)
   end

   interface Client
   begin
15 end
```

Figure 6.1: Distributed databases and servers: Interfaces.

which specifies that an object of class *classname* is to be created on the virtual machine identified by *cvm-id*. For instance,

```
#initobject cvm1 Starter(42)
```

declares that the cvm1 virtual machine should create an object of class Starter at startup.

Note that all virtual machines run the same program; however, their behaviors are different because they start with different initial objects and because objects can be created on a specified virtual machine.

6.1.3 Example: File Downloads

We now illustrate how the new constructs by an example. Multiple Internet servers share a distributed database of files, which can be downloaded by users. Each server offers the same files, but may have different presentations of the files and use different languages. The servers can be located on different machines, and the servers do not necessarily have access to all databases.

Popular and heavily accessed files are stored in many or all of the databases, not-so-popular files are stored in one or a few of the databases. Therefore, when a server gets a file request, it might have to check multiple databases before it locates the file. It may even not have the file in any of the databases it has access to, in which case it returns no file.

Clients communicate with servers and a server communicates with its databases. This way the clients do not need to know more than one

```
1  class ServerC(dbList:List[DB]) implements Server
   begin
     op local(in inList:List[DB] out outList:List[DB]) ==
        var restList:List[DB];
5       if inList = nil
        then outList := nil
        else local(rest(inList); restList);
             if isLocal(first(inList))
             then outList := append(first(inList), restList)
10           else outList := restList
             fi
        fi
     op remote(in inList:List[DB] out outList:List[DB])==/*similar*/
     op tryGet(in list:List[DB], filename:String
15             out ok:bool, file:File) ==
        if list = nil
        then (ok, file) := (false, null)
        else first(list).get(filename; ok, file);
             if not(ok)
20           then tryGet(rest(list), filename; ok, file)
             fi
        fi

    with Client
25   op get(in filename:String out ok:bool, file:File) ==
        var dbList2:List[DB];
        local(dbList; dbList2);
        tryGet(dbList2, filename; ok, file);
        if not(ok)
30      then remote(dbList; dbList2);
             tryGet(dbList2, filename; ok, file)
        fi
   end
```

Figure 6.2: The implementation of a server.

server to access multiple databases. We define interfaces DB, Server and Client; see Figure 6.1.

The implementation of the server is given in Figure 6.2. The server has a list of its databases. When a server receives a request for a file, it first checks local databases before it checks remote databases. It has an internal method local which selects the local databases by using the predicate *isLocal* (on line 8). Similarly, it has a method remote which selects the databases which is *not* local. Further, the server has a method tryGet which takes as arguments a list of databases and a filename, and tries to download the specified file from one of the databases.

We create a number of databases and servers, both locally and remote; see Figure 6.3. We assume the database is implemented by a class DB-

```
#CVM cvm1 ''einn.ifi.uio.no''
#CVM cvm2 ''tva.ifi.uio.no''
#CVM cvm3 ''tva.ifi.uio.no''
#initobject cvm1 Starter()

class Starter
begin
  op run ==
     var db:DB, dbList:List[DB]=nil,
         s:Server, sList:List[Server]=nil;
     db := new DBclass(..);
     dbList := add(dbList, db);
     db := new DBclass(..) @ cvm2;
     dbList := add(dbList, db);
     db := new DBclass(..) @ cvm3;
     dbList := add(dbList, db);
     s := new Server(dbList);
     sList := add(sList, s);
     s := new Server(dbList) @ cvm3;
     sList := add(sList, s);
     ...
end
```

Figure 6.3: Initializing the databases and servers.

class. We declare three virtual machines: one on “einn” and two on “tva”. The initial object is specified on line 4: an instance of class Starter is created on the cvm1 virtual machine.

The class Starter declares three databases: one on each virtual machine (the first database is a new local object, which happens to be in cvm1). Two servers are created: one local and one on the cvm3 virtual machine; both are given a list of the databases.

6.2 Extending the Model

The model as defined in Chapter 3 has most of what is needed to define virtual machines and remote objects. We abstract away the physical machines’ host names given by the #CVM preprocessor construct; hence, the introduction of this construct has no influence on the model. The virtual machines of the initial objects are already defined in the model; therefore, the #initobject construct imposes no changes. What is missing is the statement “new remote object” and a way to invoke the “new object” service of other virtual machines.

We introduce two new message types newRemoteObject and newRemoteObjId. They are used to forward newObject messages and newObjId

messages, respectively. The syntax is:

- newRemoteObject(*CVM-id*$_{to}$, *CVM-id*$_{from}$, newObject(...))
- newRemoteObjId(*CVM-id*$_{to}$, *CVM-id*$_{from}$, newObjId(...))

For both message types, the last argument is the wrapped message. The Central is extended with a service "new remote object"; this is as the "new object" service except that it is invoked from objects in other virtual machines and the object identifier is sent back. It uses the new message types.

When a new remote object statement v := new A() @ cvm is executed, the Creol object creates a message

newRemoteObject(cvm$_{to}$, cvm$_{from}$, newObject(obj, ...))

where the new object message is as for a new object statement. This message is transported to the virtual machine cvm$_{from}$'s out-queue, then to the cvm$_{to}$'s in-queue and at last to the Central in cvm$_{to}$. A new object is created just as for the "new object" service. Then a message

newRemoteObjId(cvm$_{from}$, newObject(obj, ...))

is sent back to cvm$_{from}$ so the identifier can be returned to the creator object.

6.3 Extending the Implementation

In Section 6.1 we extended the Creol language with a concept of virtual machines and a new statement to create remote objects; in Section 6.2 we made some small changes to the model to include the new statement. For the implementation part, the communication between the virtual machines is another challenge. Since our implementation is based on the Java platform, we will use Java's remote method invocation (RMI) [13] for CVM intercommunication.

The use of Java RMI has a great impact on the virtual machine as a whole; therefore, we start by presenting Java RMI in Section 6.3.1. Then we give an overview of the changes in Section 6.3.2 and details in Section 6.3.3.

6.3.1 Background: Java RMI

A Java program runs on a single Java Virtual Machine (JVM). Java has streams and sockets[1] which can be used by a program to communicate over the Internet with other programs on other JVMs; however, using streams and sockets involves lots of details and is a tedious and low-level way of programming. For example, objects must be transformed into streams and vice-versa. These transformations are called marshalling and demarshalling of data.

The Java Remote Method Invocation (RMI) framework [13] is designed to make it easier to communicate between JVMs. Java RMI automatically generates code which takes care of:

- Launching and configuration: setting up connections between JVMs.
- Marshalling and demarshalling of data, and the mechanism of invoking a method in another JVM.

Remote objects

Remote objects in Java are described as follows [31]:

> In the Java platform's distributed object model, a remote object is one whose methods can be invoked from another Java virtual machine, potentially on a different host. An object of this type is described by one or more remote interfaces, which are interfaces written in the Java programming language that declare the methods of the remote object.

A remote interface extends the `Remote` interface and declares a set of *remote methods*. Each remote method must declare `RemoteException` in its `throw` clause; `RemoteException` is the common superclass for a number of communication-related exceptions that may occur during the execution of a remote method call. The class `UnicastRemoteObject` is used for exporting a remote object and obtaining a stub that communicates to the remote object. An alternative, which we will use, is to create a subclass of `UnicastRemoteObject`.

To illustrate, we use the bounded buffer example from Chapter 2. We want the bounded buffer's methods to be invoked from other JVMs. We define a remote interface `Buffer` which extends the `Remote` interface; see Figure 6.4. Note that the methods of the `Buffer` interface are declared

[1] A *stream* is a continuous flow of data, designed to be processed sequentially. A *socket* is an access point, usually the combination of an IP address and a port number.

```
1  interface Buffer extends Remote {
     public void append(Object x) throws RemoteException;;
     public Object remove() throws RemoteException;;
   }
5
   class BoundedBuffer extends UnicastRemoteObject implements Buffer {
     private static final long serialVersionUID = 42;
     ...
     public BoundedBuffer(int n) throws RemoteException {
10       ...
     }
     public void append(Object x) {
         ...
     }
15   public Object remove() {
         ...
     }
   }
```

Figure 6.4: A remotely accessible bounded buffer.

to throw a RemoteException. The class BoundedBuffer implements the Buffer interface, and extends the class UnicastRemoteObject. The constructor of UnicastRemoteObject may throw a RemoteException; hence, the constructor of BoundedBuffer is declared to throw RemoteException. The BoundedBuffer class has an attribute serialVersionUIO which serves the purpose of version control for remote objects (check if the client and the server have the same version of the BoundedBuffer class).

Stubs and RMI compiler

Objects which call methods on objects on another JVM are referred to as *client objects*; the objects which are called are referred to as *server objects*. Client objects communicate with server objects through a *stub*. A stub is a client-side object which represents a single server object inside the client's JVM. The stub implements the same methods as the server object, maintains a socket connection to the server object's JVM and is responsible for marshalling and demarshalling data on the client side. It is automatically generated by the RMI compiler. The class defining the server object is compiled and a new class file is created and given the extension "_Stub". The command

```
$ rmic BoundedBuffer
```

creates a stub for the bounded buffer, that is, a file BoundedBuffer_Stub.-class. The RMI compilation must be done after the usual Java compilation

and before the program is started.

Remote object registry

On the Internet (and other networks), programs use *ports* to identify connection points. The program "rmiregistry" listens to a specified port. Servers can bind objects to this port by giving a String to identify the object. Similarly, clients can get references to objects which are already bound to a port and a name. We let the rmiregistry program listen to port number 8181:

```
$ rmiregistry 8181
```

To bind a server object, we use the method Naming.rebind(). Note that an exception is thrown if the URL is malformed, if the rmiregistry is not started (with the specified port) or another network related exception occurs. We create a bounded buffer and bind this to the name "buffer" on port number 8181 (assuming the machine name is "tva.ifi.uio.no"):

```
Buffer b = new BoundedBuffer(10);

try {
  Naming.rebind(‘‘rmi://tva.ifi.uio.no:8181/buffer’’, b);
}
catch(RemoteException e) {
  /* do appropriate */
}
catch(MalformedURLException me) {
  /* do appropriate */
}
```

Now, a reference to the buffer is fetched by the call Naming.lookup() (possibly on another host):

```
Buffer buffer;
try {
  buffer = (Buffer) Naming.lookup(‘‘rmi://einn.ifi.uio.no↩
      ↪:8181/buffer’’);
}
catch(RemoteException e) {
  /* do appropriate */
}
catch(MalformedURLException me) {
  /* do appropriate */
}
```

The reference buffer can now be used to call the buffer's methods; e.g., buffer.append(new Integer(42)).

6.3.2 Overview of the Changes

Now we are ready to present the changes in the implementation. We first give an overview of the changes; we identify five changes:

- The introduction of a virtual machine identifier and a mapping from each virtual machine to the actual machine in which it runs.
- Each initial object has a specified virtual machine.
- The new service "New remote object".
- Communication between virtual machines.
- The new statement v := *classname*(E) @ *CVM-id*.

The identifier of a given virtual machine is given at start-up, that is, as a command-line argument. The mapping from virtual machine identifiers to host names is part of the program; each #CVM declaration is stored in the CreolProgram object which represent the program.

The initial objects are stored in the same way as before, except that each initial object is extended with the virtual machine identifier. At initialization, the virtual machine's identifier is compared to that of the initial objects; only initial objects with matching identifier are created.

The Central is extended with the new service "New remote object", implemented by a method *newRemoteObject*(). The interface CreolObject-Services is also extended with this method.

As in the model, our implementation has an in-queue and an out-queue for each virtual machine. The in-queue is remotely accessible, and the out-queue at one virtual machine is responsible for communicating with the in-queue of another virtual machine. The out-queue will obviously transport messages, but in addition it will forward the invocation of the new remote object service to the specified virtual machine. It may seem unnatural that the queues have anything to do with services; however, in order to reduce the complexity we concentrate all communication between virtual machines through the queues.

The new queues are implemented by classes CVMinQueue and CVM-outQueue. We use Java RMI to get the in-queue remotely accessible; therefore, the class CVMinQueue is a subclass of UnicastRemoteObject.

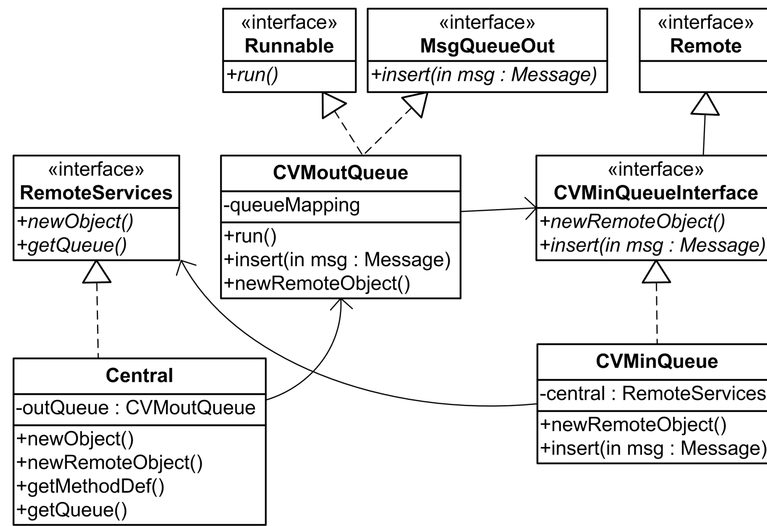

Figure 6.5: UML class diagram [23] showing new classes, interfaces and associations.

CVMinQueue also implements an interface CVMinQueueInterface, as remote method invocations are always through interfaces. The out-queue is active, that is, the class CVMoutQueue implements the interface Runnable and we start a new thread at initialization.

The implementation of the "new remote object" statement is very similar to the "new object" statement, except that it invokes the Central.*newRemoteObject*() method instead of Central.*newObject*() method.

Recall the UML class diagram from Chapter 4 (Figure 4.1). Figure 6.5 shows a UML diagram after the changes.

6.3.3 Detailed Changes

We now discuss the details of the changes in the implementation. We start with the queues, as this is the most difficult part.

The in-queue

We have an in-queue for each virtual machine. This queue serves the purpose of forwarding messages to the addressed Creol objects and to invoke the "new object"-service of the Central. We define an interface CVMinQueueInterface with methods *insert*() and *newRemoteObject*(). The class CVMinQueue implements CVMinQueueInterface and hence the interface's methods. In addition, it is a subclass of UnicastRemoteObject.

```
1  public interface CVMinQueueInterface extends Remote {
     public ObjVal newRemoteObject(String cname, DataList par)
         throws RemoteException;
     public void insert(Message msg) throws RemoteException;
5  }

   public class CVMinQueue extends UnicastRemoteObject
                           implements CVMinQueueInterface {
    private static final long serialVersionUID = 1;
10  private RemoteServices central;

    public CVMinQueue(RemoteServices central) throws RemoteException {
        super();
        this.central = central;
15  }
    public void insert(Message msg) {
        MsgQueueOut msgQ = central.getQueue(msg.getDestination());
        msgQ.insert(msg);
    }
20  public ObjVal newRemoteObject(String cname, DataList par) {
        return central.newObject(cname, par);
    }
   }
```

Figure 6.6: The virtual machine's in-queue.

See Figure 6.6. Further, we define an interface RemoteServices with methods *newRemoteObject*() and *getQueue*(); this interface is implemented by the Central and defines the services available for the in-queue.

As described in Section 6.3.1, we make the queue remotely accessible by compiling the queue with the program `rmic` (Remote Method Invocation Compiler):

```
$ rmic CVMinQueue
```

This generates the stub which is necessary for remote method invocation; the stub is explained in Section 6.3.1.

The out-queue

The out-queue is implemented by the class CVMoutQueue. It implements the interface MsgQueueOut (see Chapter 4, Section 4.2.3) and it has public methods *insert*() and *newRemoteObject*(). The latter method is used by the Central to forward the "new remote object" call.

Recall that we use the Creol object's execution thread to forward messages from the Creol object's out-queue to the addressed in-queue. We

```
1  public class CVMoutQueue implements MsgQueueOut, Runnable {
    private HashMap<String,String> nameMapping;
    private HashMap<String,CVMinQueueInterface> queueMapping;
    private LinkedList<Message> queue;
5   private Lock msgLock, mappingLock;
    private Condition msgArrived;

    public CVMoutQueue(HashMap<String,String> nameMapping) {
        super();
10      this.nameMapping = nameMapping;
        this.queueMapping = new HashMap<String,CVMinQueueInterface>();
        queue = new LinkedList<Message>();
        msgLock = new ReentrantLock(true);
        msgArrived = msgLock.newCondition();
15      mappingLock = new ReentrantLock(true);
    }
    ...
   }
```

Figure 6.7: The class CVMoutQueue.

consider this to be a good solution because it does not involve any extra threads and because it only gives a small delay. The sending of messages to objects in other virtual machines involves bigger delays. Therefore, we do not use the Creol object's execution thread; the out-queue is an active object which forwards messages to the addressed virtual machine. Hence, the class CVMoutQueue implements the Runnable interface. For the "new remote object" service, the situation differs, as the Creol object must wait for the object identifier anyway. Therefore, for this service we use the Creol object's execution thread.

The out-queue must store a mapping from virtual machine identifiers to the host names; for this it uses a hash map nameMapping. This mapping is used to set up a connection to a virtual machine's in-queue. Once a connection has been set up, it stores a reference to the virtual machine's in-queue in a hash map queueMapping. Further, it has a list of messages (the attribute queue) and fair locks to protect queueMapping and queue. To signal the arrival of a message, we use a condition variable called msgArrived. See Figure 6.7 for the attributes and the constructor of class CVMoutQueue.

We define a method *setupConnection*() to set up a connection between an out-queue and an in-queue; see Figure 6.8. As discussed previously, the host name is given in the nameMapping hash map, and we use port number 8181. The actual binding to the in-queue of the specified virtual machine is done by the call Naming.lookup(host) (line 8). The remote virtual machine may not have been started yet; if so, either a connect

```
1  private CVMinQueueInterface setupConnection(String cvm) {
    CVMinQueueInterface inqueue = null;
    String mname = nameMapping.get(cvm);
    String host = "rmi://" + mname + ":8181/" + cvm;
5
    while(inqueue == null) {
      try {
        inqueue = (CVMinQueueInterface) Naming.lookup(host);
      }
10    catch(ConnectException ce) {
        try { Thread.sleep(1000); } catch (InterruptedException ie) {}
      }
      catch(NotBoundException nbe) {
        try { Thread.sleep(1000); } catch (InterruptedException ie) {}
15    }
      catch(Exception e) {
          e.printStackTrace();
          System.exit(1);
      }
20  }
    return inqueue;
   }
```

Figure 6.8: The method CVMoutQueue.setupConnection().

exception or a not bound exception is thrown. We do not want to make any assumptions about the order in which virtual machines are initialized; therefore, we catch these exceptions and go for sleep for a while, waiting for the remote virtual machine to be ready for connections. At success, the in-queue of the remote virtual machine is returned.

Message transportation from a virtual machine's out-queue to another virtual machine's in-queue is performed by the out-queue's execution thread; see Figure 6.9.

Creol objects insert messages by the method *insert*(). This method is quite simple: the message is appended to the end of the message queue and the out-queue thread is notified using the msgArrived condition variable.

The out-queue's active behavior is defined by a method *sendMessages*() and started in the *run*() method. The out-queue waits for and receives messages by the method *nextMsg*(). It checks if there exists a connection to the addressed virtual machine, and if so this connection is used, if not a new connection is set up (line 31). Then the message is inserted in the in-queue by using the *insert*() method of the in-queue (line 37). Note that the actual sending of a message is outside a critical region; thus, Creol objects can insert messages while the out-queue sends messages, enabling parallel execution.

```
public void insert(Message msg) {
    msgLock.lock();
    queue.addLast(msg);
    msgArrived.signal();
    msgLock.unlock();
}

private Message nextMsg() {
    msgLock.lock();
    while(queue.size() == 0) {
        try { msgArrived.await(); }
        catch (InterruptedException ie) {}
    }
    Message msg = queue.remove();
    msgLock.unlock();
    return msg;
}

private void sendMessages() {
    CVMinQueueInterface inqueue;
    String destCVM;
    Message msg;
    while(true) {
        msg = nextMsg();
        destCVM = msg.getDestination().getCvm();
        mappingLock.lock();
        if(queueMapping.containsKey(destCVM)) {
            inqueue = queueMapping.get(destCVM);
        }
        else {
            inqueue = setupConnection(destCVM);
            queueMapping.put(destCVM,inqueue);
        }
        mappingLock.unlock();

        try {
            inqueue.insert(msg);
        }
        catch (RemoteException re) {
            re.printStackTrace();
            System.exit(1);
        }
    }
}

public void run() {
    sendMessages();
}
```

Figure 6.9: The transportation of messages between virtual machines.

```
public ObjVal newRemoteObject(String cname,DataList par,String at){
    CVMinQueueInterface inqueue;
    ObjVal result = null;
    mappingLock.lock();
    if(queueMapping.containsKey(at)) {
        inqueue = queueMapping.get(at);
    }
    else {
        inqueue = setupConnection(at);
        queueMapping.put(at,inqueue);
    }
    mappingLock.unlock();

    try {
        result = inqueue.newRemoteObject(cname, par);
    }
    catch (RemoteException re) {
        re.printStackTrace();
        System.exit(1);
    }
    return result;
}
```

Figure 6.10: The method Central.newRemoteObject()

The out-queue has a method *newRemoteObject*() used by the Central to invoke the corresponding service in a remote virtual machine; see Figure 6.10. As for the sending of messages, the queueMapping hash map is checked to see if there is a connection to the remote virtual machine, if not, a new connection is set up. Then a call to the in-queue's *newRemoteObject*() method is made, and the result of this method call is returned.

Creol program representation

We extend the Creol program representation with a method getCVMtoMachineMapping(). This method returns a mapping from virtual machine identifiers to the host name of the physical machine on which the virtual machine runs. The mapping is given in a hash map and used by the CVM out-queue for setting up connections to remote virtual machines.

Each initial object is extended with the name of the virtual machine on which it will be created, that is, the class InitObject is extended with a method getCvm() which returns the virtual machine identifier.

The CVM

Each virtual machine is identified by a String, and the constructor of class CVM is therefore extended with an argument `String id`. The creation of initial objects is changed: we only create the objects with matching virtual machine identifier. The CVM also initializes the in-queue and the out-queue. The class CVM is given in Figure 6.11.

As mentioned previously, we want the new constructs to have as little impact on the virtual machine as possible. The introduction of virtual machine queues imposes the use of the program `rmiregistry`. For programs which only run on one virtual machine, it is not necessary to set up queues; therefore, we use a special virtual machine identifier “cvm” to specify that we only have one virtual machine and thus should not create any virtual machine queues.

In case the virtual machine queues are needed, a new out-queue and a thread for the out-queue are created. Then the Central is created and a reference to the out-queue is given as a parameter to the Central. The method *remoteSetup*() creates an in-queue and this in-queue is bound to the port number 8181 with the virtual machine identifier. Finally, the inital objects are created; this is straight forward.

The Central: A new service

The Central offers a new service: “New remote object”. Therefore, the interface CreolObjectServices and the class Central are extended with a method *newRemoteObject*(). The Central uses the virtual machine’s out-queue to forward the method call to the target virtual machine. In case the specified cvm is this cvm, the *newRemoteObject*() method simply calls *newObject*() in the same central. See Figure 6.12.

Virtual machine’s initialization

Recall that we declare virtual machines by `#CVM` initialization statements. For each of the host in these definitions, we must start `rmiregistry` on the specified host. We must also start an instance of a virtual machine for each of the specified CVM. For a given initialization statement `#CVM cvmid ”hostname”`, we start rmiregistry and the virtual machine as follows:

```
$ rmiregistry 8181 &
[1] 15976
$ java Program cvmid
```

where `Program` is a Java class with the representation of the Creol program we want to run.

```
public class CVM {
 public CVM(String name, CreolProgram creolprogram) {
  Central central;
  if(name.equals("cvm")) {
      central = new Central(name, creolprogram.getClassDefinitions↩
          ↪(), creolprogram.getSubtypeGraph(), null);
  }
  else {
      CVMoutQueue outqueue =
          new CVMoutQueue(creolprogram.getCVMtoMachineMapping());
      Thread t = new Thread(outqueue, name + ".CVMouQueue");
      t.start();
      central = new Central(name, creolprogram.getClassDefinitions↩
          ↪(), creolprogram.getSubtypeGraph(), outqueue);
      remoteSetup(name, central);
  }

  InitObject io;
  InitObjectList iol = creolprogram.getInitObjects();

  /* Create initial objects */
  while(iol != null) {
      io = iol.first();
      if(io.getCvm().equals(name)) {
          central.newObject(io.getClassname(), io.getParameters());
          iol = iol.rest();
      }
  }
 }

 private void remoteSetup(String cvm, Central central){
   CVMinQueueInterface inqueue = null;
   try {
       inqueue = new CVMinQueue(central);
   } catch(RemoteException re) {
       e.printStackTrace();
       System.exit(1);
   }
   String host = "rmi://"  + ":8181/" + cvm;
   try {
       Naming.rebind(host, inqueue);
   } catch (Exception e) {
       e.printStackTrace();
       System.exit(1);
   }
 }
}
```

Figure 6.11: The class CVM.

```
1   public ObjVal newRemoteObject(String classname, DataList ↩
        ↪actualParameters, String at) {
        if(at.equals(cvmname)) {
            return newObject(classname, actualParameters) throws ↩
                ↪RemoteException;;
        }
5       else {
            return outqueue.newRemoteObject(classname, actualParameters↩
                ↪, at) throws RemoteException;;
        }
    }
```

Figure 6.12: The method Central.newRemoteObject().

6.4 Example Run: Distributed Santa Claus Problem

Recall the Santa Claus problem from Chapter 2. In Chapter 4 we changed it somewhat to get some output on the console: we introduced a print statement and the Santa Claus class was changed to print to the console whenever he interacts with the reindeer or elves. We now make similar changes to the reindeer and elves. Further, we introduce three virtual machines called “cvm1”, “cvm2” and “cvm3” and specify the hosts on which they are to be run; see Figure 6.13. The Santa Claus is created locally whereas the reindeer and elves are created on remote virtual machines.

The virtual machine “cvm1” is run on host “viisi.ifi.uio.no”, “cvm2” on host “kolme.ifi.uio.no”, and “cvm3” on host “kaksi.ifi.uio.no”. We start the remote object registry on each of these machines (`rmiregistry 8181`) and then we run the Santa Claus problem. The initial object and thus the Santa Claus are created on “cvm1”; the following is the output on the “cvm1” console:

```
$ java SantaClauseProblemRemote cvm1
Santa Claus delivers toys.
Santa Claus talks to elves: (cvm3,0), (cvm3,1), (cvm3,2)
Santa Claus delivers toys.
Santa Claus talks to elves: (cvm3,3), (cvm3,4), (cvm3,0)
Santa Claus talks to elves: (cvm3,2), (cvm3,1), (cvm3,3)
  ...
```

On host “kolme.ifi.uio.no” we start the virtual machine “cvm2”. The reindeer are the only objects on this machine. We get the following output on the console:

```
$ java SantaClauseProblemRemote cvm2
Reindeer ’(cvm2,1)’ delivers toys!
Reindeer ’(cvm2,4)’ delivers toys!
```

```
1  #CVM cvm1 ''viisi.ifi.uio.no''
   #CVM cvm2 ''kolme.ifi.uio.no''
   #CVM cvm3 ''kaksi.ifi.uio.no''
   #initobject cvm1 Starter()
5
   class Starter
   begin
     op run ==
        var sc:SantaClaus, r:Reindeer, e:Elf;
10      sc := new SantaClaus();
        r := new Reindeer(sc) @ cvm2;
        r := new Reindeer(sc) @ cvm2;
         ...
        r := new Reindeer(sc) @ cvm2;
15      e := new Elf(sc) @ cvm3;
        e := new Elf(sc) @ cvm3;
         ...
        e := new Elf(sc) @ cvm3
   end
```

Figure 6.13: A distributed version of the Santa Claus problem.

```
Reindeer '(cvm2,3)' delivers toys!
Reindeer '(cvm2,2)' delivers toys!
Reindeer '(cvm2,0)' delivers toys!
Reindeer '(cvm2,5)' delivers toys!
Reindeer '(cvm2,6)' delivers toys!
Reindeer '(cvm2,8)' delivers toys!
Reindeer '(cvm2,7)' delivers toys!
  ...
```

The elves are created on virtual machine “cvm3”; we get the following output on the “cvm3” console:

```
$ java SantaClauseProblemRemote cvm3
Elf '(cvm3,1)' talks to Santa!
Elf '(cvm3,2)' talks to Santa!
Elf '(cvm3,0)' talks to Santa!
Elf '(cvm3,3)' talks to Santa!
Elf '(cvm3,4)' talks to Santa!
Elf '(cvm3,0)' talks to Santa!
Elf '(cvm3,2)' talks to Santa!
Elf '(cvm3,1)' talks to Santa!
Elf '(cvm3,3)' talks to Santa!
  ...
```

Consider the three first lines on the “cvm3” console. As expected, the three first elves which talk to Santa correspond to the first group of elves Santa talks to (see the “cvm1” console).

6.5 Summary

In this chapter we have introduced a notion of virtual machines into the Creol language, and the possibility to create objects on remote virtual machines. Only small changes in the model had to be done; we introduced message wrappers to forward `newObject` and `newObjId` messages to other virtual machines. For the implementation part, the changes were more extensive. We presented Java RMI which was used for communication between virtual machines. We saw that remote queues could be used in the same way as local queues. Finally, we gave an example run of a distributed version of the Santa Claus problem.

Chapter 7

Conclusion

In this chapter we summarize the contributions of this thesis and give suggestions for further research.

7.1 Contributions

This thesis has presented a new run-time model for Creol and a prototype of the model has been implemented on the Java Platform. To explore the thesis' contributions in detail, we review the questions presented in Chapter 1.

- Can a run-time model for Creol be defined which supports code sharing?

We have described a model which can represent Creol programs. A computation of the model is defined as operations on a state using rules defined by pre- and postconditions. The model does not impose any restrictions to Creol. In particular, multiple inheritance is supported.

We have made a prototype implementation of the model on the Java platform. By using Java threads the prototype faithfully implements the model with regard to concurrency. This in turn implies the possiblity of true concurrency on mulitprocessor systems. The multithreading has also contributed to make the model and the implementation similar. The prototype demonstrates that the model can serve as a basis for low-level implementations of Creol run-time environments.

For the sake of simplicity, the model itself does not have code sharing; however, the code is changed in a way that makes code sharing possible in an implementation. Our prototype implementation has code sharing.

- Can Creol and its operational model be extended to support real distribution?

We have introduced new Creol constructs used to specify how Creol objects may be distributed over multiple virtual machines. Instead of using a host name, each virtual machine is given a unique identifier. This abstraction makes programming with explicit virtual machines, which may be found somewhat low-level, more elegant. We claim that the new Creol constructs are intuitive and easy to use.

Further, we have extended the model to support the new Creol constructs. Java RMI is used to implement distribution in the prototype, and different virtual machines can be located on different hosts on the Internet. Thus, the prototype supports real distribution in the sense that objects are distributed among different physical machines.

7.2 Further Work and Research

The work presented in this thesis is part of the Creol project[1]. Hence, further work and research concerning our model and implementation should be seen in the context of this project. With this in mind, the following areas may be suggested for further research.

Verifying the correctness of the model

In Chapter 2 we gave an informal definition of the semantics of Creol. In Chapter 3 the semantics was defined precisely by a computational model. However, we have not shown that the model is correct, that is, computes Creol programs in accordance with the semantics of the language. Our example-runs in the prototype implementation behaved as expected. This *indicates* that the model is correct. To verify the correctness of the model such indications are not enough. We must validate that the model is sound with respect to the formal semantics of the language.

Creol has an abstract operational semantics which is formally defined in Rewriting Logic (RL) [24]. The RL semantics define the valid computations of Creol programs. The possible executions of a program in our model must also be possible in the RL semantics, given some appropriate notion of state transformation. This can hopefully be proved by induction over computations.

[1] `http://www.ifi.uio.no/~creol/`

A full proof of the model's correctness requires a lot of work. Nevertheless, a comparison of the two models is interesting. In many aspects, our model and the RL semantics are very similar; however, there are some noteworthy differences:

- *Status flag:* In our model, the objects have a status flag which explicitly expresses the current task of the object. The RL semantics does not have a status flag.
- *The merge statement:* In order to support code sharing, we have introduced a set of suspended program statement lists used to implement the merge statement. We have also introduced a statement joinMerge. This solution is very different from that of the RL semantics, where code is manipulated directly.
- *The return of method calls:* We have introduced a statement return which sends the completion message to the caller. The sending of a completion message is defined differently in the RL semantics.
- *Explicit processor release points:* The evaluation of the wait guard is done differently in our model than in the RL semantics.
- *Multiple inheritance:* The method look-up mechanism is differently defined; in our model we use functions whereas the RL semantics use messages.

A first validation of our model should focus on these subjects. For instance, it must be shown that the introduction of the status flag does not impose any problems. In particular, it must be checked that the objects can under no circumstances go into dead-lock because of the use of a status flag.

Creol compiler

The way we represent Creol programs as Java objects is low-level, and despite that it is straight-forward it is error-prone to manually translate from Creol programs to the Java representation. To really get the benefits from having a virtual machine and to experiment with Creol programs, we need a compiler which can do this automatically and in addition do type checking.

A prototype compiler for Creol has been developed [10]; this compiler creates Creol machine code for the virtual machine in Maude [2]. This work can be used as a starting point for a compiler for our virtual machine in Java.

Dynamic updates

The Creol run-time environment in Maude has been extended with a mechanism for dynamic updates [26]. Classes and interfaces are updated at run-time. Due to virtual binding of methods, the code which is executed by method calls, can change over time. It would be interesting to extend our implementation to support dynamic updates.

We have made a model with centralized code, that is, each virtual machine has the class definitions stored in a central. This way of organizing the code should make dynamic updates feasible. However, introducing new class attributes will be a challenge as an object has to be extended with these attributes before it executes code which refers to these attributes.

CVM variables and run-time creation of virtual machines

We have used a preprocessor construct `#CVM` to define virtual machines. This construct is static and not very flexible. For instance, it is not possible to send references to virtual machines between objects. Introducing virtual machine variables into the Creol language may facilitate new possibilities in program design.

Further, our virtual machines are defined statically. It would be interesting to create new virtual machines dynamically at run-time. The Creol language must be extended with a new construct for creating new virtual machines. Implementing run-time initialization of virtual machines on the Java platform may be a challenge.

Process scheduling

In our model we have not specified how to schedule processes; a ready process is selected nondeterministically. In our implementation, the first ready process is selected by using a first-in first-out queue. Processes in the queue are re-checked for readiness; this re-checking is inefficient. Especially when it comes to recursive calls this reevaluation is inefficient as many processes wait for method returns. A study of process scheduling in Creol may lead to better performance in program execution.

In Creol, each object has a single execution thread and hence processes execute one at a time. In combination with the processor release points (the `await` statement), this mutual exclusion has similarities to conditional critical regions [15]. Brinch Hansen states that it does not seem possible to implement conditional critical regions efficiently and that

the root of the problem is the unbounded reevaluation of Boolean expressions until they are true [4]. However, the await statement in Creol also includes the testing for method returns. The return of a method call can be treated as a conditional variable, that is, the return can signal the process which waits for the return. This can dramatically reduce the number of reevaluated processes.

We also believe that the evaluation of boolean expressions can be done more efficiently; e.g., by only reevaluating a guard if at least one of the variables has changed.

Bibliography

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, Reading, Mass., 2000.

[2] M. Arnestad. En abstrakt maskin for Creol i Maude. Master's thesis, Department of Informatics, University of Oslo, Nov. 2003. In Norwegian. Available from `http://heim.ifi.uio.no/~creol`.

[3] P. Brinch Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[4] P. Brinch Hansen. The Invention of Concurrent Programming. In *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 3–61. Springer-Verlag New York, Inc., 2002.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.

[6] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Some features of the SIMULA 67 language. In *Proceedings of the Second Conference on Applications of Simulations*, pages 29–31. Winter Simulation Conference, 1968.

[7] J. Dovland, E. B. Johnsen, and O. Owe. Reasoning about Asynchronous Method Calls and Inheritance. In *Proc. of the Norwegian Informatics Conference (NIK'04)*, pages 213–224. Tapir Academic Publisher, Nov. 2004.

[8] J. Dovland, E. B. Johnsen, and O. Owe. Verification of Concurrent Objects with Asynchronous Method Calls. In *Proceedings of the IEEE International Conference on Software Science, Technology & Engineering(SwSTE'05)*, pages 141–150. IEEE Computer Society Press, Feb. 2005.

[9] B. Eckel. *Thinking in Java.* Prentice Hall, Third edition, 2003.

[10] J. H. Fjeld. Compiling Creol Safely. Master's thesis, Department of Informatics, University of Oslo, May 2005.

[11] P. Golde, A. Hejlsberg, and S. Wiltamuth. *The C# Programming Language*. Addison Wesley Professional, 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Third edition, 2005.

[13] W. Grosso. *Java RMI*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[14] C. A. R. Hoare. Monitors: An Operating Systems Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[15] C. A. R. Hoare. Towards a Theory of Parallel Programming. In *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 231–244. Springer-Verlag New York, Inc., 2002.

[16] E. B. Johnsen and O. Owe. Object-Oriented Specification and Open Distributed Systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *Lecture Notes in Computer Science*, pages 137–164. Springer-Verlag, 2004.

[17] E. B. Johnsen and O. Owe. A Dynamic Binding Strategy for Multiple Inheritance and Asynchronously Communicating Objects. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Proc. 3rd International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 274–295. Springer-Verlag, 2005.

[18] E. B. Johnsen and O. Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modelling*, 2006. To appear.

[19] E. B. Johnsen, O. Owe, and E. W. Axelsen. A Run-Time Environment for Concurrent Objects with Asynchronous Method Calls. In N. Martí-Oliet, editor, *Proc. 5th International Workshop on Rewriting Logic and its Applications (WRLA'04), Mar. 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 375–392. Elsevier, Jan. 2005.

[20] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A Dynamic Class Construct for Asynchronous Concurrent Objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal*

Methods for Open Object-Based Distributed Systems (FMOODS'05), volume 3535 of *Lecture Notes in Computer Science*, pages 15–30. Springer-Verlag, June 2005.

[21] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A Type-Safe Object-Oriented Model for Distributed Concurrent Systems. Research Report 327, Department of Informatics, University of Oslo, June 2005.

[22] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Second edition, 1999.

[23] K. S. Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Publishing, Second edition, 1999.

[24] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[25] J.-F. Monin. *Understanding Formal Methods*. Springer Verlag, 2002.

[26] M. Ofstad. Dynamic Updates in the Creol Framework. Master's thesis, Department of Informatics, University of Oslo, Apr. 2005.

[27] O. Owe and I. Ryl. The Oslo University Notation: A Formalism for Open, Object Oriented, Distributed Systems. Research Report 270, Department of Informatics, University of Oslo, 1999.

[28] M. Persson. Kompilator fra OUN til Java. Master's thesis, Department of Informatics, University of Oslo, Feb. 2002. In Norwegian.

[29] N. Plat and P. G. Larsen. An Overview of the ISO/VDM-SL Standard. *SIGPLAN Not.*, 27(8):76–82, 1992.

[30] B. Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley Verlag, Boston, 2000.

[31] Sun Microsystems. Java Technology. `http://java.sun.com/`.

[32] J. A. Trono. A New Exercise in Concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.

Appendix A

Creol Examples

In the following sections, we list the complete Creol code for the examples “The Bounded Buffer”, “The Santa Claus Problem” and “Authorization policies”.

A.1 The Bounded Buffer

```
1   /* First the interfaces */
    interface BufferP
    begin with Producer
      op append(in d:Data)
5   end

    interface BufferC
    begin with Consumer
      op remove(out d:Data)
10  end

    interface Buffer inherits BufferP, BufferC
    begin
    end
15
    interface Consumer
    begin
    end

20  interface Producer
    begin
    end

    /* The implementation of the buffer */
25  class BoundedBuffer(max:int) implements Buffer
    begin
      var buffer:List[Data]=empty, n:int=0
```

```
  with Producer
  op append(in d:Data) == await n < max;
     buffer := add(buffer,d); n := n + 1

  with Consumer
  op remove(out d:Data) == await n > 0;
     d := first(buffer); buffer := rest(buffer); n := n - 1
end

/* The producer creates the natural numbers: */
class Prod(b:BufferA) implements Producer
begin
  op run == !loop(0)
  op loop(in i:int) == b.append(i); !loop(i+1)
end


/* The consumer fetches and simply prints the number: */
class Cons(b:BufferR) implements Consumer
begin
  op run == !loop
  op loop == var y:Data; b.remove(;y); print(y); !loop
end

/* A starter class */
class Starter
begin
  op run == var b:Buffer, p:Producer, c:Consumer;
     b := new BoundedBuffer(10);
     p := new Prod(b);
     c := new Cons(b)
end
```

A.2 The Santa Claus Problem

```
/* The interfaces */
interface SantaClausR
begin with ReinDeer
  op backFromHoliday
end

interface SantaClausE
begin with Elf
  op haveProblem
end

interface SantaClaus inherits SantaClausR, SantaClausE
begin
end

```

```
interface Reindeer
begin with SantaClausR
  op harness
  op unharness
end

interface Elf
begin with SantaClausE
  op enterOffice
  op leaveOffice
end

/* The classes */
class SantaClausC implements SantaClaus
begin
  var ct_rd:nat=0, wait_rd:List[Reindeer]=empty,
      harnessed_rd:List[Reindeer]=empty, ct_elves:nat=0,
      wait_elves:List[Elf]=empty, inoffice_elves:List[elf]=empty

  op run == !loop

  op loop ==
      (await ct_rd = 9; deliverToys(;) []
       await (ct_elves >= 3 /\ ct_rd != 9); talkToElves(;));
      !loop

  op deliverToys ==
      var t1:Label,..,t9:Label;
      ct_rd := 0;
      t1!first(wait_rd).harness;
      harnessed_rd := add(harnessed_rd, first(wait_rd));
      wait_rd := rest(wait_rd);
      ...
      t9!first(wait_rd).harness;
      harnessed_rd := add(harnessed_rd, first(wait_rd));
      wait_rd := rest(wait_rd);
      await t1? /\ ... /\ t9?;
      <<Pick up and deliver Toys>>;
      !first(harnessed_rd).unharness;
      harnessed_rd := rest(harnessed_rd);
      ...
      !first(harnessed_rd).unharness;
      harnessed_rd := rest(harnessed_rd);


  op talkToElves ==
      var t1:Label,t2:Label,t3:Label;
      <<Open door>>;
      ct_elves := ct_elves - 3;
      t1!first(wait_elves).showIn;
      inoffice_elves := add(inoffice_elves, first(wait_elves);
      wait_elves := rest(wait_elves);
      t2!first(wait_elves).showIn;
      inoffice_elves := add(inoffice_elves, first(wait_elves);
```

```
70        wait_elves := rest(wait_elves);
          t3!first(wait_elves).showIn;
          inoffice_elves := add(inoffice_elves, first(wait_elves);
          wait_elves := rest(wait_elves);
          await t1? /\ t2? /\ t3?;
75        <<Close door>>;
          <<Talk to elves>>;
          <<Open door>>;
          t1!first(inoffcie_elves).showOut;
          inoffice_elves := rest(inoffice_elves);
80        t2!first(inoffcie_elves).showOut;
          inoffice_elves := rest(inoffice_elves);
          t3!first(inoffcie_elves).showOut;
          inoffice_elves := rest(inoffice_elves);
          await t1 /\ t2 /\ t3;
85        <<Close door>>

      with Reindeer
      op backFromHoliday ==
          ct_rd := ct_rd + 1;
90        wait_rd := add(wait_rd,caller)

      with Elf
      op haveProblem ==
          ct_elves := ct_elves + 1;
95        wait_elves := add(wait_elves,caller)
    end


    class ReindeerC(sc:SantaClausR) implements Reindeer
100 begin
      op run == !holiday
      op holiday == <<Go on holiday>>; !sc.backFromHoliday
      op deliverToys == <<Deliver Toys>>

105   with SantaClausR
      op harness == !deliverToys
      op unharness == !holiday
    end

110
    class ElfC(sc:SantaClausE) implements Elf
    begin
      op run == !work
      op work == <<Do work>>; !sc.haveProblem
115   op talkToSanta == <<Talk to Santa>>

      with SantaClausE
      op showIn == <<Go into Santa's office>>; !talkToSanta
      op showOut == <<Leave Santa's office>>; !work
120 end

    class LeaderElfC(sc:SantaClaus, elves:List[Elves])
      inherits ElfC(sc) implements Elf
```

```
begin
125   op run == !loop
      op work == (<<Lead the elves>> ||| <<Make toys>>); !sc.↩
          ↪haveProblem
    end

    class Christmas
130 begin
      op run ==
        var sc:SantaC, r1:Reindeer,...,r9:Reindeer, e1:Elf,...,e7:Elf;
        sc := new SantaClausC;
        r1 := new ReindeerC(sc);
135     r2 := new ReindeerC(sc);
        ...
        r9 := new Reindeer(sc);
        e1 := new ElfC(sc);
        e2 := new ElfC(sc);
140     ...
        e6 := new ElfC(sc);
        e7 := new LeaderElfC(sc, [e1,e2,...,e6])
    end
```

A.3 Authorization Policies

```
1   interface High
    begin
     with Agent
      op openH(out ok:Bool)
5     op access(in key:int out y:Data)
      op closeH
    end

    interface Low
10  begin
     with Agent
      op openL
      op access(in key:int out y:Data)
      op closeL
15  end

    interface HighLow inherits High, Low
    begin
    end
20
    class DB
    begin
      op access(in key:int, high:bool out: y:Data) == /* ... */
      op clear(in x:Agent out ok:bool) == /* ... */
25  end

    class SAuth
```

```
var gr:Agent=null
begin
  op grant(in x:Agent) == await (gr = null); gr := x
  op revoke(in x:Agent) == if gr = x then gr := null fi
  op auth(in x:Agent) == await (gr = x)
end

class MAuth
var gr:Set[Agent]=empty
begin
  op grant(in x:Agent) == gr := gr U { x }
  op revoke(in x:Agent) == gr := gr \ { x }
  op auth(in x:Agent) == await (x in gr)
end

class HAuth implements High inherits SAuth, DB
begin
  op acc(in x:Agent, key:int out y:Data) ==
     auth(x);
     await access@DB(key, true; y)
 with Agent
  op openH(out ok:Bool) ==
     await clear(caller; ok);
     if ok then grant(caller) fi
  op access(in key:int out y:Data) ==
     acc(caller, key; y)
  op closeH == revoke(caller)
end

class LAuth implements Low inherits MAuth, DB
begin
  op acc(in x:Agent, key:int out y:Data) ==
     auth(x);
     await access@DB(key, false; y)
 with Agent
  op openL == grant(caller)
  op access(in key:int out y:Data) ==
     acc(caller, key; y)
  op closeL == revoke(caller)
end


class HLAuth implements HighLow inherits LAuth, HAuth
begin with Agent
  op access(in key:int out y:Data) ==
     if caller=gr@SAuth
     then acc@HAuth(caller, key; y)
     else acc@LAuth(caller, key; y)
     fi
end
```

Appendix B

Java Representation

The Java representation of Creol programs is low-level and boring details; therefore, we will not discuss it in detail. However, we list the Java representation of the Bounded Buffer example in the next section.

Note that in case of a branching statement, e.g., the if-statement, a method toListRep() is called in order to set some "next statement"-pointers. See Appendix C for further notes.

B.1 The Bounded Buffer

```
1  public class BoundedBuffer extends StandardMachines
                              implements CreolProgram {
    public static void main(String[] argv) {
        if(argv.length == 0)
5           new CVM("cvm", new BoundedBuffer());
        else
            new CVM(argv[0], new BoundedBuffer());
    }

10  public ClassDefinitionSet getClassDefinitions() {

     ClassDefinitionSet classdef = new ClassDefinitionSet();

     /* BoundedBuffer class */
15   CreolClass boundedbuffer =
      new CreolClass(
       null,
       new VdeclList(new Vdecl("buffer", new DataList(null)),new ↩
           ↪VdeclList(new Vdecl("max",new IntVal(10)),new VdeclList(↩
           ↪new Vdecl("n",new IntVal(0))))),
       null,
20     new HashMap<String, CreolMethod>());
     boundedbuffer.addMethod("append",
```

```
      new CreolMethod(
        new VarList("d"),
        new Await(new IntL(new IntVar("n"), new IntVar("max")),
25       new Assignment("buffer", new DataListAdd("buffer", new ↩
            ↪DataVar("d")),
         new Assignment("n", new Plus(new IntVar("n"), new IntVal(1))↩
            ↪,
         new Return(null)))),
        null,
        new Sig(new StringList("Data"), null),
30      new Co("Producer")));
    boundedbuffer.addMethod("remove",
      new CreolMethod(
        null,
        new Await(new IntL(new IntVal(0), new IntVar("n")),
35       new Assignment("d", new DataListFirst(new DataListVar("↩
            ↪buffer")),
         new Assignment("buffer", new DataListRest(new DataListVar("↩
            ↪buffer")),
         new Assignment("n", new Minus(new IntVar("n"), new IntVal(1)↩
            ↪),
         new Return(new VarList("d"))))),
        new VarList("d"),
40      new Sig(null,new StringList("Data")),
        new Co("Consumer")));
    classdef.addClass("BoundedBuffer", boundedbuffer);

    /* Prod class */
45  CreolClass prod =
      new CreolClass(new VarList("b@Prod"),null,null,new HashMap<↩
         ↪String, CreolMethod>());
    prod.addMethod("run",
       new CreolMethod(
         null,
50       new Call(null,"loop",null,"Prod",new ExprList(new IntVal(1))↩
            ↪,new Sig(new StringList("int"),null),new Co("Void")),
         null,
         new Sig(null, null),
         new Co("Void")));
    prod.addMethod("loop",
55    new CreolMethod(
         new VarList("i"),
         new VarDecl("t",null,
          new Print("###␣Producer␣sends:␣'", new IntVar("i"), "'\n",
          new ExternalCall("t", new ObjVar("b@Prod"), "append", new ↩
             ↪ExprList(new IntVar("i")),new Sig(new StringList("int"↩
             ↪),null),new Co("Producer"),
60        new Await(new ReplyGuard("t"),
          new Reply("t", null,
          new Call(null,"loop",null,"Prod",new ExprList(new Plus(new ↩
             ↪IntVar("i"), new IntVal(1))),new Sig(new StringList("↩
             ↪int"),null),new Co("Void"),
          new Return(null))))))),
         null,
```

```
      new Sig(new StringList("int"), null),
      new Co("Void")));
  classdef.addClass("Prod", prod);

  /* Cons class */
  CreolClass cons = new CreolClass(new VarList("b@Cons"),null,null,↩
     ↪new HashMap<String, CreolMethod>());
  cons.addMethod("run",
    new CreolMethod(null,
      new Call(null,"loop",null,"Cons",null,new Sig(null,null),new↩
          ↪ Co("Void")), null, new Sig(null, null), new Co("Void")↩
          ↪));
  cons.addMethod("loop",
    new CreolMethod(
      null,
      new VarDecl("t",null,
       new VarDecl("y", null,
       new ExternalCall("t", new ObjVar("b@Cons"), "remove", null,↩
          ↪new Sig(null, new StringList("Data")),new Co("Consumer↩
          ↪"),
       new Await(new ReplyGuard("t"),
       new Reply("t", new VarList("y"),
       new Print("###␣Consumer␣prints:␣'", new IntVar("y"), "'\n",
       new Call(null,"loop",null,"Cons",null,new Sig(null,null),↩
          ↪new Co("Void"),
       new Return(null)))))))),
      null,
      new Sig(null, null),
      new Co("Void")));
  classdef.addClass("Cons", cons);

  /* Starter class */
  CreolClass starter =
     new CreolClass(null, null, null, new HashMap<String,↩
        ↪CreolMethod>());
  starter.addMethod("run",
    new CreolMethod(
     null,
     new VarDecl("b", null,
      new VarDecl("p", null,
      new VarDecl("c", null,
      new NewObject("b", "BoundedBuffer", null,
      new NewObject("p", "Prod", new ExprList(new ObjVar("b")),
      new NewObject("c", "Cons", new ExprList(new ObjVar("b"))))))↩
          ↪)),
     null,
     new Sig(null,null),
     new Co("Void")));
  classdef.addClass("Starter", starter);

  return classdef;
 }

 public InitObjectList getInitObjects() {
```

```
        return new InitObjectList(new InitObject("Starter", null));
    }

    public SubtypeGraph getSubtypeGraph() {
115     HashMap<String, StringList> subtypes = new HashMap<String, ↩
            ↪StringList>();
        subtypes.put("int", new StringList("Data"));
        subtypes.put("nat", new StringList("int"));
        subtypes.put("string", new StringList("Data"));
        subtypes.put("Any", new StringList("Data"));
120     subtypes.put("BufferA", new StringList("Any"));
        subtypes.put("BufferR", new StringList("Any"));
        subtypes.put("Buffer", new StringList("BufferA", new StringList("↩
            ↪BufferR")));
        subtypes.put("Producer", new StringList("Any"));
        subtypes.put("Consumer", new StringList("Any"));
125     SubtypeGraph sg = new SubtypeGraph(subtypes);
        return sg;
    }
}
```

Appendix C

Prototype Notes

Here we present some additional notes and details about the CVM prototype:

- Organization of classes and interfaces in packages.
- Logging.
- How to use the prototype.
- Where to download the prototype.

C.1 Details

This section describes some details for which no room was found in the thesis, but which are important for those who want to use the CVM or who want to change or extend the CVM.

Packages

The CVM prototype consists of many classes and interfaces. These classes and interfaces need some kind of organization. Therefore, we have collected them in different packages. Each of these packages is a sub-package of cvm:

cvm.classdef contains classes and interfaces which are in close relationship with the Creol program definition; e.g., the interface CreolProgram and classes ClassDefinitionSet, CreolClass, CreolMethod, etc.

cvm.code contains the Statement interface and the classes which implement statements; e.g., AssignmentList, Await, NewObject, etc.

cvm.func contains classes and interfaces which define the functional part of Creol; e.g., interface Data, BoolExpr, Label, IntVal, IntEq, etc.

cvm.logg contains two classes for logging purposes: CVMLogger and ObjLogger

cvm.msg contains classes and interfaces defining the message queues and the message types.

cvm.node contains the classes CVM and Central and the interfaces which Central implements.

cvm.object contains classes which define the Creol object and the components of the Creol object; e.g., classes CreolObject, CreolProcess, CompSet, etc.

The test programs are stored in a package cvmtest (this is not a subpackage of cvm).

Logging

The development of a virtual machine is a demanding and difficult job. Often programming errors do not give compilation errors but appear as strange behavior of the program. To reveal and fix errors, the use of a logger has been crucial.

We have used the Apache log4j[1] logger to log the execution of the virtual machine. Figure C.1 shows the Logger class from the log4j package. Each logger has a debug level; this level can be set at run-time and hence the degree of debugging can be set without recompiling. We use a file `config.log4j` to specify the configuration properties; e.g., the debugging level for each type of loggers. This file is read by the prototype program.

The use of a logger is rather simple: a call to the method specifying the appropriate debugging level. For instance, we log the execution of a Creol assignment statement with debug level "info":

```
logger.info(id + ‘‘:=’’ + expr.toString());
```

[1] http://logging.apache.org/log4j/

```
package org.apache.log4j;

public class Logger {
    // Creation & retrieval methods:
    public static Logger getRootLogger();
    public static Logger getLogger(String name);
    // printing methods:
    public void debug(Object message);
    public void info(Object message);
    public void warn(Object message);
    public void error(Object message);
    public void fatal(Object message);
    // generic printing method:
    public void log(Level l, Object message);
}
```

Figure C.1: The `Logger` class.

where `id` is the variable identifier and `expr` is the expression which is evaluated and assigned to `id`. If the debugging level for this logger is set to "info" or below, this will log information to a given file. We will not go into more details about how loggers are set up and used; the interested reader is referred to Apache's home site: `http://logging.apache.org/log4j/`.

We have used a number of loggers. Each logger has a name, and logs to a file in the `logg` directory. The most important loggers are:

- An execution log for each Creol object. The execution of a statement is logged.
  Name: `ExecLog.<objid>`
  Filename: `<objid>.execution.log`

- A variable log for each object. All local variables and attributes between the execution of statements are logged.
  Name: `VarLog.<objid>`
  Filename: `<objid>.variables.log`

- A console log for each object, which log whatever the object writes to the console (using the print statement).
  Name: `ConsoleLog.<objid>`
  Filename: `<objid>.console.log`

- Loggers for the objects' in-queue and out-queue. These loggers log when a message is inserted and removed. There are also loggers for each virtual machine's in-queue and out-queue.

```
public class StandardMachines {
  public HashMap<String,String> getCVMtoMachineMapping() {
    HashMap<String,String> hm = new HashMap<String,String>();
    hm.put("cvm1", "viisi.ifi.uio.no");
    hm.put("cvm2", "kolme.ifi.uio.no");
    ...
    return hm;
  }
}
```

Figure C.2: The StandardMachines class.

```
public class SantaClausProblem extends StandardMachines
                               implements CreolProgram {
  public static void main(String[] argv) {
    if(argv.length == 0)
      new CVM("cvm", new SantaClausProblem());
    else
      new CVM(argv[0], new SantaClausProblem());
  }
  public ClassDefinitionSet getClassDefinitions() { ... }
  public InitObjectList getInitObjects() { ... }
  public SubtypeGraph getSubtypeGraph() { ... }
}
```

Figure C.3: The SantaClausProblem class.

The logging files are stored in the catalog `logg`. The debugging of each type of logger can be set in `config.log4j` by using the logger name; how this is done is self-explained by investigating the file. `<objid>` is the object identifier of the object; e.g., `(cvm1,0)` is the first created object on the virtual machine `cvm1`.

Specifying Host Machines

Recall from Chapter 6 that the CreolProgram interface has a method getCVMtoMachineMapping(). We have a class StandardMachines which has this method so that we do not need to specify new virtual machine for each program we make, see Figure C.2. Hence the classes which implement CreolProgram can extend this class and thus the method does not need to be specified. The class StandardMachines must be modified so that the host names correspond to the machines on which the virtual machines will run.

The Creol program classes have a method main which creates an instance of CVM; see Figure C.3. As discussed in Chapter 6, not all programs

need multiple virtual machines. If no virtual machine is specified at start-up, the special virtual machine identifier “cvm” is used to pass this information to the `CVM` constructor.

C.2 Download and Use

This is a “How to” for downloading, installing, and using the prototype virtual machine.

log4j

The log4j must be downloaded (`http://logging.apache.org/log4j/`). This is a Java jar file. The excact name depends on the version number; for instance it can be `log4j-1.2.12.jar`. This file must be added to the Java class path. How this is done is system dependent. On unix systems with a `bash` shell, write

```
export CLASSPATH=$CLASSPATH:path/log4j-1.2.12.jar
```

either in `bash` or in the file `.bashrc`.

Prototype Download

The prototype is available for download at

```
http://heim.ifi.uio.no/~ivaralm/thesis/
```

All files are available in the cvm directory. The files are also packed in a file called `cvm.tar.gz`. Download this file and unpack by `tar`; e.g., the command

```
~ $ tar -xvzf cvm.tar.gz
```

This command creates a directory `cvm` which contains three sub-directories (`cvm`, `cvmtest` and `logg`) and one file (`config.log4j`). The `cvm` directory corresponds to the `cvm` package and contains the sub-packages. The `cvmtest` directory corresponds to the `cvmtest` package. The `logg` directory is an empty directory for log files. The `config.log4j` file is the logging configuration file. After unpacking the `cvm.tar.gz` file, the `cvm` directory should look something like:

```
~/cvm $ ls
config.log4j  cvm  cvmtest  logg
~/cvm $ ls cvm
classdef  code  func  logg  msg  node  object
```

Compiling

The Java files need to be compiled before execution. For example, if you want to execute the Santa Claus Problem example, compile `cvmtest/-SantaClausProblem.java`:

```
~/cvm $ javac cvmtest/SantaClausProblem.java
```

To be able to run distributed programs, the CVM in-queue stub must first be created, that is, the CVMinQueue.java file must be compiled by the RMI compiler:

```
~/cvm $ rmic cvm.msg.CVMinQueue
```

Run

The Santa Claus problem is now ready for execution:

```
~/cvm $ java cvmtest.SantaClausProblem
```

In case you want to execute a distributed version, the virtual machine identifier must also be given, for instance

```
~/cvm $ java cvmtest.SantaClausProblemRemote cvm1
```

The examples at the end of the chapters 4, 5 and 6 give additional information about how to execute the programs.