UNIVERSITY OF OSLO
Department of Informatics

Dynamic Updates in the Creol Framework

Master's Thesis

Morten Ofstad

April 27, 2005

Abstract

In the present day, distributed systems are becoming more and more important. It can be difficult to stop such systems to perform an update, either because they are critical infrastructure, like air traffic control systems, or simply because we don't have direct control of the nodes. As a result of this, *dynamic updates* are of great interest, but so far no standard approach has emerged that strikes a good balance between the partly conflicting goals of safety, flexibility and performance. This thesis presents a specification of a dynamic update system in the Creol framework for distributed, object oriented programming. The system is designed to dynamically update classes, and includes a procedure for converting existing object instances to an updated version. A method is presented for controlling when and how an update is applied, and the necessary conditions for a correct update are discussed. The system presented is a low-overhead solution, lazily transferring state from one version to the next. It is very flexible, even allowing changes to the inheritance hierarchy. The system is given a formal specification in rewriting logic, extending Creol's semantics. Consequently, formal reasoning about the correctness of updates is possible.

Preface

This thesis was written over a period of two years, from May 2003 to May 2005. During this time I had the chance to go to Amsterdam for one year and study logic at the excellent Institute for Logic, Language and Computation at the University of Amsterdam. I would like to thank my advisor, Einar Broch Johnsen for showing great flexibility with the practical arrangements while I was away and while I have been working part time, for all his constructive criticism and for entrusting me with such an interesting subject. I thank my colleagues at Hue A/S, Bjørn, Diderich, Johan, Paul, Stein and Thomas, who have been very accommodating during the last 6 months while I have been distracted with work on this thesis. I would also like to thank Stefan Slobach at the ILLC who was a great inspiration to me during my stay in Amsterdam, and to all the friends I made while I was there. Finally, I thank my parents and my partner Inger Lill Skuland for their love and support throughout this period.

Contents

Introduction

This thesis seeks to answer how a mechanism for dynamic updates in an object oriented programming language for distributed systems can be built. The programming language used in this thesis is Creol [14, 15], which has been developed at the Department of Informatics at the University of Oslo to study dynamic updates of distributed object systems in a formal framework. The reason why we want to replace or upgrade parts of an object oriented distributed system is that it can be difficult to stop such a system and perform an update, either because it's a critical system like an air traffic control system or simply because we don't have direct control of the nodes, such as in a peer to peer distributed system.

Problem Statement

The main question this thesis seeks to answer is:

How can we make a safe and flexible mechanism
for dynamically updating Creol programs?

By *safe* we mean that we can determine with absolute certainty that the system will continue to operate correctly after it has been updated. By *flexible* we mean that a wide range of updates can be performed, not just adding attributes and methods to classes. By *dynamic update* we mean updating a program while it is running, with minimal disruption of the services provided by the program.

The goals of safety and flexibility are to some extent conflicting. If we severely limit the range of updates which are allowed, it obviously becomes easier to ensure the correct behaviour of the updated system. Conversely, if we do not make any guarantees about the behaviour of the updated system, it's obviously much easier to design a flexible update mechanism.

After reviewing the available literature on the subject, we will try to give answers to the subproblems which present themselves once we have a more clear understanding of the problem:

- What should be the *unit of replacement* in our system?

- How should *state transfer* from the old version to the new version be accomplished?
- How can we maintain *system responsiveness* while the update is in progress?
- Which conditions are necessary for a *correct* update?

Method

The work we have done builds directly on earlier work by Marte Arnestad [1] who made an operational semantics for the Creol language in the form of a rewriting logic [23] specification which can be interpreted using the Maude [5] tool. Many improvements and extensions to the interpreter have been made by Prof. Olaf Owe and my advisor Einar Broch Johnsen. To investigate the question of how we can make a safe and flexible mechanism for dynamic updates in Creol, we have made a specification for such a system using rewriting logic and used the Maude tool to test this specification. We have also made a simple compiler/translator to check that the proposed syntax for updates in Creol gives an unambiguous grammar and to ease the work of creating examples.

Results

A mechanism for dynamic updates is presented that works with classes as the unit of replacement. A procedure for converting existing objects to the new version is presented that works by following the inheritance hierarchy of an object's class and executing state conversion routines for each updated class in the inheritance hierarchy. A flexible approach to controlling updates is used which categorises the methods of the updated class in three different categories. Parameter conversion is provided to allow old code to make calls on updated objects. The mechanism presented is a low-overhead solution, lazily transferring state from one version to the next. This can be done through several versions, so it's not necessary to wait for one update to complete before applying another update. Finally, necessary conditions for a correct update in our system are discussed.

Thesis Outline

Chapter 1 presents a survey of previous work in the field [2, 12, 13, 22] and introduces the main concepts used in update systems. Two early update systems, Argus [3] and DYMOS [19] are presented in some detail to illustrate different approaches that can be used when designing an update system. Chapter 2 introduces the Creol programming language and its model for concurrent objects. Chapter 3 presents the Creol interpreter, starting with a quick review of rewriting logic which has been used to specify both the interpreter and the update system. The two next chapters present the main subject of this thesis, the Creol update system, and discusses the choices made during its implementation. Chapter 4 deals mainly with state transfer and Chapter 5 with mechanisms for controlling how and when updates are applied. Finally, in Chapter 6, we discuss necessary conditions for the correctness of updates.

Chapter 1

Update Systems

Even from the beginning of research into programming languages and operating systems there has been interest in how to update different components of a running system [10]. We are so used to be able to update programs with new versions as they come out that we don't even recognise this as module replacement in a running (operating) system. However, replacements of this granularity are not that interesting, since they normally require the program to be stopped before the new version can be started. What we are interested in are techniques that offer a finer granularity for replacements so that *parts* of running programs can be updated. Other key features of update systems are which kinds of updates can be performed while preserving program correctness and how program state is transferred from the old to the new version when an update occurs. We will also discuss how these issues relate to distributed and object oriented systems. We will then describe two influential systems, Argus [3] and DYMOS [19], developed in the beginning of the 80s which have fundamentally different designs. These systems provide a good background for understanding the design decisions of more recent systems.

1.1 Basic Concepts of Update Systems

We will begin by reviewing some of the basic concepts used in texts on update systems. Some important concepts are *Granularity*, *State Transfer* and *Quiescence* and we will discuss each in turn. We will then look at three different approaches to *when* an update can take place.

1.1.1 Granularity of Updates

If we choose the whole program as the unit of replacement then the entire program must be stopped in order to perform an update. In general we cannot have an active thread of execution inside the unit we are replacing, since we cannot in all cases construct a function which takes the execution state [1] in one version and maps it to an execution state in the updated version. Because of this, we

[1]By execution state we mean for example program counter and register contents in a register machine abstraction.

would typically like to replace as small a unit as possible. However if we allow updates to happen at a very low level it is likely to incur significant performance penalties to running programs, and it will also be very hard to reason about the correctness of updates.

In order to facilitate reasoning about the correctness of updates, research into program updating has concentrated on replacing units that have a clearly defined interface (e.g. the calling convention used for procedure calls), not individual statements or machine instructions. In procedural systems it is quite common to choose procedures as the smallest unit of replacement, while in object oriented systems it makes sense to choose either unique objects or, more commonly, classes as the smallest unit of replacement. However, in most cases an update will comprise more than one such smallest unit. Since a group of updates might carry mutual dependencies, we need some way to describe the process of updating so that dependencies are resolved in a correct way, or we must choose a bigger unit of replacement so that we don't get dependencies we cannot resolve. Because of this problem it is also common to choose entire modules as the smallest unit of replacement.

1.1.2 State Transfer

All systems for updating running programs share the need for state transfer. This is the process of translating the data that goes with one version of the program into data suitable for the new version. If it is impossible to change the data representation of the program when updating, the system is so severely limited that it will most likely be of little practical value. Consequently, a replacement needs to contain not only the updated program code, but also extra code to do state transfer. To ensure that the state transfer can be done, the replacement must be carried out when the data is in a consistent state, so the timing of the update also becomes an issue. In formal systems it is convenient to specify invariants which hold at certain points of program execution. These points then become candidates for stopping the program, replacing the code with the updated version and carrying out state transfer.

Aside from the difficulties of timing state transfer, it can also be difficult to describe the state transfer functions in a convenient way. In most systems, the implementation language for the application is also used for state transfer, but since the state transfer function needs to be able to refer to different versions, this can cause difficulties when adding updating capabilities to an already existing language. For example if you want to add update capabilities to a language like Java [22] it is not possible to write a function that takes an object of the old version of a class as an argument and returns an object of the new version of the class, since there is no way to talk about versions and the two classes might have different class members. A way of solving this problem is to provide a version independent description of the object (in Java by using the java.io.Serializable interface), but this can involve a lot of extra work for the programmer. In formal systems it can also be difficult to prove the correctness of an update if the state transfer function can contain arbitrary code. However, a separate language to describe state transfer is not convenient either, and adds to the complexity of the system.

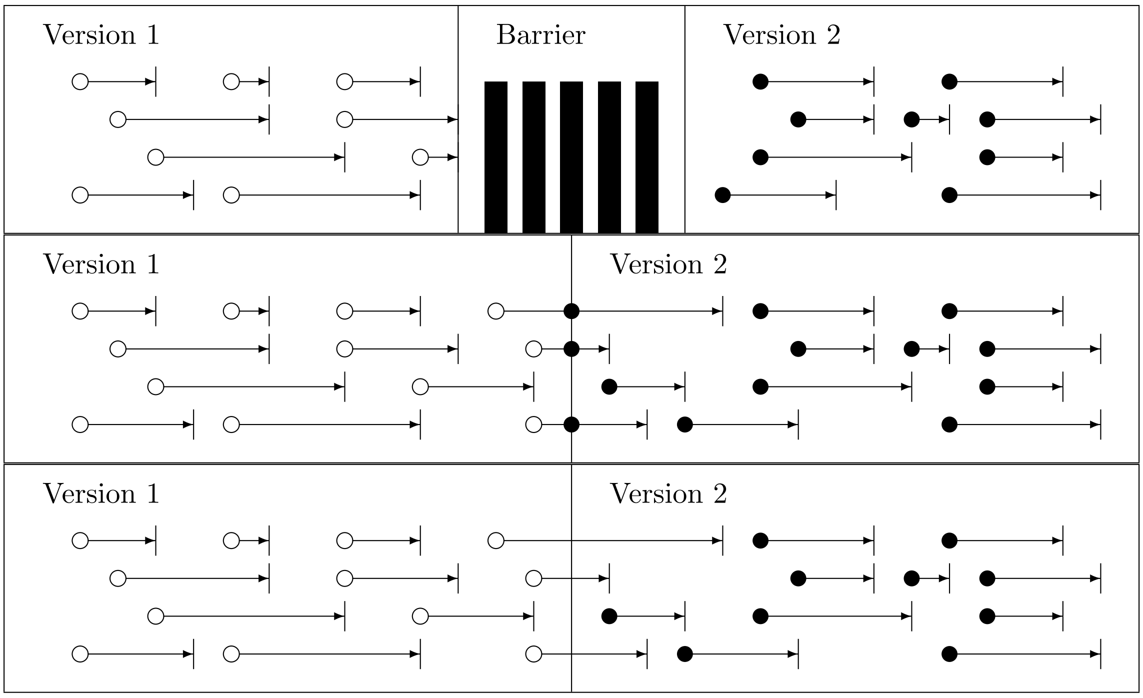


Figure 1.1: Three approaches to update timing

1.1.3 Quiescence

In transaction based systems, a part of the system is said to be quiescent if there are no ongoing transactions that involve that part. If a part is quiescent, that part can be replaced without disrupting the system as a whole as long as the replacement has the same semantics with respect to the rest of the system as the old part. The notion of quiescence can be extended to mean that a part of the system does not have an active thread of execution. In order to replace a unit which is not quiescent you would have to have a state transfer function for all possible points in a program. To generate this kind of state transfer function from one version to the next is impossible in general as it requires information about the semantics of the program which is unavailable to the system. However, some systems provide a way of specifying transfer of the execution state, for example by labelling points in the code where upgrades can take place and inserting matching labels in the updated code to provide a mapping which makes it possible to translate execution state between these points. This is an interesting approach, but requires the implementor of the old version to have anticipated where upgrades will happen.

1.1.4 Timing of Updates

In object oriented systems, if the unit of replacement is chosen to be classes, there is an additional decision to be made as to when we update the instances of the class which has been replaced. We can choose between three main approaches to updating the objects (Figure 1.1). The first approach is to put up a barrier, blocking creation of new instances of the replaced class until all instances of the old version of the class have expired. This approach has the advantage that state transfer is not necessary, but there is no guarantee of when (or even if) an update will be carried out. The second approach is to do explicit state transfer of old objects at the time of the replacement. This is similar to the barrier approach in that you never get two objects that are instances of different versions of the same class. However, creating state transfer functions has its own problems as discussed earlier. The last approach is to allow instances of different versions of a class to coexist. This is the approach taken by Hjálmtýsson and Gray [13], and is perhaps the easiest to implement. They extend this approach slightly by providing objects with a way of discovering whether they are instances of the latest version of the class, making it possible to explicitly migrate objects to the new version.

1.1.5 Updates in Distributed Systems

In a distributed system, it is usually not convenient or even possible to shut down the entire system to perform an update. Therefore, research into updating running systems is of great interest to those who work with distributed systems. There are however some additional challenges to updating distributed systems compared to traditional systems. Since it can be hard or impossible to control that all nodes get the update at the same time, provisions have to be made for different versions to communicate with each other during a transition period without affecting program correctness. It can also be hard to ensure quiescence of the node that you want to update with respect to the rest of the system. In transaction based systems such as Argus, there is usually a built in capability for error recovery, so quiescence can be ensured by cancelling all ongoing transactions before an update is carried out. However, error recovery can in general not be completely automated so a lot of work is left to the programmer. Although fault tolerance is an important subject in distributed systems, it's not necessarily a good idea to rely on the error handling mechanisms of a system to facilitate performing updates.

1.1.6 Updates in Object Oriented Systems

The main problems facing work on updates in object oriented systems are state transfer and inheritance. The state transfer problem is harder for object oriented systems because of the relationship between classes and the objects that are instances of a class. Updating a class means you have to locate instances of this class, a capability which always comes at some cost. It's also much harder to keep track of active threads of execution, since there can be different threads executing in different instances of the class. The common way of building larger systems by reusing code through inheritance also creates additional challenges, because not only instances of the class which has been updated needs to be located and processed, but also instances of any subclass which reuses the code that has been replaced. Formalising the relationships between classes so that correctness criteria for updates can be established has also proven to be hard, and even the most recent works on formalising updates [27] have ignored object oriented systems.

1.2 The Argus Distributed Programming System

Argus is a system for distributed programming, and was one of the first such systems to get support for dynamic replacement [3]. Argus is based on the CLU language [20], and adds distributed computing to this language through modules called *guardians*. A guardian provides some service or encapsulates some resource. Guardians communicate solely through message passing and preform *actions* in response to messages. Though many guardians can exist on a node in the system, a single guardian only exists on one particular node of the system, hence guardians can be thought of as logical nodes in the system. Internally a guardian may consist of any number of processes and objects, but to the outside it looks like a single object.

1.2.1 State and Actions

The *state* of a guardian is the set of objects internal to the guardian. It can be divided into *stable* and *volatile* objects. The stable objects are kept consistent and guaranteed to survive crashes through a database–like system that ensures that changes to these objects are committed to stable storage when actions complete. Should a crash occur, all stable objects are loaded from the last committed version and a special recovery function is called to reinitialise the volatile objects. Atomicity is guaranteed for all actions, including nested actions. When a sub-action commits, the parent action can see the results of this action, but unrelated actions will be unable to see the changes until the top level action finally commits. Argus also provides means of error recovery so that an action can still complete even if a sub-action aborts. This sophisticated system for error recovery is taken advantage of in the implementation of updates in Argus.

1.2.2 The Unit of Replacement

The minimum unit of replacement in the Argus update system was determined to be guardians. This is a natural choice because guardians have a small, well-defined interface to the rest of the system. Furthermore, guardians have a stable state that can be used to resume execution after a crash. The same stable state can also be used to resume execution after an update, and it is also easy to know exactly which state that is subject to state transfer. Replacing smaller units than guardians was deemed to be impractical because it would be too difficult to ensure that any changes in the permanent state would not affect other parts of the guardian. However, bigger units of replacement are considered important because the ability to replace individual guardians will always be restricted by the implementation details of the subsystem it belongs to, along with preserving the interface. When some of the interface is only used by a certain subsystem it should be possible to change that part of the interface as long as the entire subsystem is updated at the same time. This increases the complexity of making replacements, because subsystems might span several nodes and also have a hierarchical structure that single guardians do not. A major part of the work on replacements in Argus has gone into how subsystems are described and determining the implicit interface of the subsystem so that it is possible to change the interfaces used by the guardians that make up the subsystem as long as the (implicit) interface to the subsystem as a whole is unchanged.

1.3 The DYMOS Dynamic Modification System

DYMOS [19] is a system for dynamic modification of running programs based on the StarMod programming language [6]. In DYMOS the programmer modifies and recompiles the source code of procedures and modules that are to be replaced. The programmer then requests the system to change the current core image to incorporate new code and data. All this is done using a command language, so the programmer will first issue an **edit** command, then a **compile** command and finally an **update** command.

1.3.1 The Unit of Replacement

Since DYMOS is based on a procedural language, the natural unit of replacement is procedures. Procedures can be added, modified (including modifications to parameter lists) or deleted. If a procedure is deleted or its parameter list modified, then additional changes to any procedures that call the changed procedure will be required. The update command will return an error if it is issued before all such required changes have been made. It is also possible to supply a *parameter convert* routine that will automatically convert calls of an old procedure to calls of the new procedure. DYMOS is one of the few systems where it is possible to replace procedures which are running. This is accomplished through labelled statements in the code which provide points where an update can take place. Code to convert local variables at a labelled statement can be provided along with the new version, and there is a way to qualify names of variables so that it is possible for the new local variables to have the same name as the old ones. Modules and even types can also be replaced, and can also have conversion code to allow state transfer to take place when they are updated. These features make DYMOS the most flexible of the update systems surveyed.

1.3.2 Timing of Replacements

The **update** command in the DYMOS system can have an optional **when-idle** clause that specifies that the replacement is not done until the specified modules and procedures do not have any active code. It is also possible to specify the order in which changes are introduced when the compilation is done by using the **compile X after Y** command. This makes it easy to state the requirements for a correct update, but unfortunately it is hard to formally prove that a certain update will ever take place. Therefore, a time bound can also be specified for the update command, after which it will return an error if the conditions in the **when-idle** clause have not been met. The system can automatically break down bigger updates into component updates that can be carried out one at a time by looking at the dependencies between different procedures that are available as a result of compilation. This helps to reduce the time it takes for an update to complete since the **when-idle** part of the update can be broken down into separate requirements for each part of the update.

1.4 Formalising Updates

The work by Bloom on Argus was one of the first to present a formal model for updates. He also showed the need for formalising updates by presenting several examples which show the subtle problems that can occur when you make replacements. One of the most interesting ones being that of unique ID generators. In this case you can have two implementations of a specification that both satisfy the requirements of the specification, but are not replacement compatible. Although both generators will generate unique IDs, replacing one with the other can result in the same ID being generated again. It is also demonstrated that reasoning about a single replacement is not enough, because the set of correct states can be changed by earlier replacements. These subtle issues are important to keep in mind for anyone designing a mechanism for replacement in any language or system.

1.4.1 Reasoning with History and Futures

The Argus model for formalising updates reasons about abstractions of each component of the system. Each abstraction has a set of events that can occur at the abstraction's interface, a set of abstract states and a mapping from the set of states to sequences of future events that can occur starting from that state. Intuitively an abstract state is the history of past events, and futures describe allowable sequences of events that can follow that history. This model is based on earlier work by Stark [26] and bears a striking resemblance to the model used in the OUN notation [25] which is a precursor for CREOL. We can then define the concrete state of an implementation as a mapping to a set of abstract states (since several histories can lead to the same concrete state). The concrete futures is then a mapping from the concrete state to event sequences that can follow from that state.

We define how an implementation can satisfy the abstraction by requiring that the concrete futures from each concrete state are a subset of the futures of each abstract state that the concrete state maps to. The condition for correct replacement in this model is that the new instance generates only futures that would be permitted by the replaced abstraction as continuations from the state where the replacement occurred. This model does well in reasoning about distributed systems, but requires that the units to be replaced have a formal specification. Although Bloom's model is not really object oriented, the similarities with the OUN notation suggest that it should be relatively straightforward to extend this kind of formalism to object oriented systems.

1.4.2 A Low Level Model for Updates

The work by Gupta et al. [12] takes another direction by trying to go to the lowest level representation of a program. They model the system as a program Π and a set of states, each state consisting of a variable assignment and an index into Π (the program counter). An update is then a function from Π and a state s to a new program Π' and a new state s'. They argue that such an update is valid if the state s' is reachable from the initial state of Π'. They then build on this simple model to define validity of replacement of procedures. Some suggestions are made as to how this model can be extended to object oriented and distributed systems, but the problem is that the model only defines validity in context of an entire system, so validity of an update can not be established for components without looking at the whole because there are no contracts between the components.

1.4.3 An Algebraic Approach

One can argue that it is impractical to reason about the correctness of updates in a programming language when the semantics of the programming language are not properly formalised. Because of this, Bierman, Hicks et al. [2] have developed an *update calculus* based on the first-order simply-typed lambda calculus with mutually recursive modules and a primitive for updating. Working in such an environment it is much easier to formalise correctness criteria for updates, but unfortunately it is not trivial to extend this language to include other language constructs such as abstract data types, higher order functions, objects or concurrency.

1.5 Summary

When designing a system for dynamic updates, several design decisions must be taken. First of all the unit of replacement must be decided. Then the issue of state transfer has to be dealt with. A way of dealing with code that has active threads of execution must be decided upon and a variety of trade-offs have to be made in the implementation of each of these decisions. Finally the safety of updates within the system must be evaluated, and preferably a formal method for proving correctness should be presented. Although a lot of work has been done in the area of dynamic updates, there are several unresolved problems. One of the major shortcomings so far is the failure to formalise updates as applied to distributed and object oriented systems. Furthermore, no system so far has managed to combine flexibility with low overhead, ease of use and safety in such a way that a standard approach has emerged. In the present day, distributed systems are becoming more and more important, and dynamic updates are also becoming more important as a result. In conclusion, the challenges facing anyone working on dynamic updates are great, but so are the rewards of arriving at a successful design.

Chapter 2

The Creol Programming Language

The CREOL project is a research project to investigate programming constructs and reasoning control in the context of open distributed systems, and in particular the issue of maintenance control, taking an object oriented approach [7]. A part of this project is the development of the Creol programming language which is used to implement specifications given in OUN (Oslo University Notation). The Creol programming language has a formally defined semantics, given in rewriting logic. The full Creol syntax is presented in ISO-EBNF notation in Appendix A. We will in this chapter present the most important parts of the Creol syntax together with an example program.

2.1 The Creol Language

Creol is an experimental high-level object oriented language for distributed concurrent objects. The name is an acronym for Concurrent REflective Object-oriented Language. The language is based on concurrent objects communicating by means of asynchronous method calls, and both active and reactive behaviour of objects is supported. The basic communication mechanisms and operational semantics of the language are described in [14]. The inclusion of multiple inheritance in the language and the operational semantics are described in [15].

2.1.1 Creol Built-in Data Types and Operators

Creol is a strictly typed language, and all variables have to be declared before use. The built-in data types are bool (boolean), int (integer), string and reference. References are typed by an interface which the referenced object implements. If methods from different interfaces are to be invoked on a single object, multiple references must be held to the object. Normal arithmetic (`+`, `-`, `*`, `/`), logical (and, or, not) and comparison (`<`, `>`, `<=`, `>=`, `==`, `/=`) operators work as expected. The `+` operator is overloaded to mean concatenation for string types. The `:=` operator is the assignment operator with the usual semantics.

2.1.2 Statements

There are six types of Creol statements: Assignment statements, if-then-else statements, do-while statements, object creation statements, await statements and method call statements. Object creation is considered a statement since objects cannot safely be created in an initialiser expression, but otherwise looks like an assignment statement that uses an operator 'new' to create an object of a specified class. Await statements are used to synchronise execution as will be explained shortly. Method calls in Creol are categorised as either internal or external and either synchronous or asynchronous. Asynchronous method calls that return something are labelled and the label is used in conjunction with an await statement to fetch the result of the call. There is a strong preference for using tail-recursive calls instead of do-while statements for long running loops because, as we will see later (Section 6.3.3) it is easier to update code that is written this way.

There are three operators that work on statements, in order of increasing precedence we have: The sequential composition operator `;` which means 'first execute the left statement and then the right statement', the nondeterministic choice operator `[]` which means 'execute either left statement or right statement' and the nondeterministic merge operator `||` which means 'execute both left statement and right statement in any order'. Thus we have the equivalence $P \,||\, P' = (P \,;\, P') \,[]\, (P' \,;\, P)$.

2.1.3 Processes and Processor Release Points

Creol objects are concurrent, that means each object has its own (virtual) processor which evaluate local processes. Processes may be *active*, started at creation time by the `run` method, or *reactive*, initiated by method calls from other objects. This is just a conceptual aid, as the language does not differentiate between active and reactive processes. Processes are instances of methods, and each process has its own valuation of the originating method's local variables and parameters.

Only one process can be executed in an object at any given time, and the active process continues to execute until a *processor release point* is reached. Processor release points are explicitly introduced with await statements which wait for *guards*. Guards come in three flavours: Boolean guards which wait until a boolean expression becomes true, wait guards which always release the processor and method return guards which wait for an asynchronous method call to return. Guards can be composed with the `&` operator. Synchronous method calls implicitly create a processor release points, since they are equivalent to making an asynchronous call with an immediately following method return guard.

2.1.4 Interfaces

Creol objects are only allowed to communicate with each other through interfaces. Interfaces can be parametrised on types, and multiple inheritance can be used to compose interfaces. Within a method implementation, the **caller** keyword can be used to obtain a reference to the calling object which is of the type given in the **with** clause.

2.1.5 Classes and Objects

Classes are collections of methods which implement a set of interfaces, in addition to this a class declares which attributes an object of that class has. Multiple inheritance can be used to combine classes, and if we have a shared base class, its attributes and methods are only included once[1]. The class inheritance is completely separate from the interfaces, and since all communication between objects have to use interfaces, this gives a lot of freedom in how the classes are created. Each class has a special 'run' method which takes no arguments and is started when the object is created. When a Creol program is compiled, a class is designated as the main class and an object of this class is created at program start.

Objects are instantiated from classes, and contain a valuation of the class variables and a process queue. When the object is created, one process is initially created for the 'run' method of the class of the object. Other processes are added to the process queue as a result of receiving method calls. A process contains a valuation of the method variables and a list of statements to be executed. After each statement is executed it is removed from the statement list in the process, so the next statement can be executed. When a processor release point is reached either as a result of an await statement or because a process has finished, another process is selected from the process queue and made active.

2.1.6 Inheritance and Method Binding

By default Creol methods have virtual binding, while attributes are statically bound as they are not available through interfaces but only to methods in the class they are defined in and its subclasses. That means attributes with the same name overrides attributes in parent classes, but sometimes we want to refer to these attributes. The same goes for methods, sometimes we want to call the method of a specific class, for example it is common when overriding a method to call the original method in addition to doing some extra processing. For these reasons Creol names of methods or attributes can be *qualified* with a class name to refer to the method or attribute defined in that specific class. The syntax for a qualifying attributes is `identifier@class` to refer to the identifier in the context of the specified class. This syntax is also extended for state transfer to be able to refer specifically to the old versions of attributes as discussed in the next chapter.

2.2 The Dining Philosophers

One of the first example programs developed in the Creol system [16] was an implementation of the Dining Philosophers problem due to Dijkstra [8]. At the time, Creol did not include implementation inheritance so the implementation used two separate classes, Philosopher and Butler. In a later paper, Creol was extended to include implementation inheritance [15], and a new Dining Philosophers example was developed where Butler is a subclass of the Philosopher. We will use the

[1]In contrast to C++ which defaults to including the shared base class once per subclass in the inheritance graph, unless you specify so-called *virtual* inheritance for every subclass.

```
interface Phil
begin
  with Phil
    op borrowStick
    op returnStick
end
```

```
interface Butler
begin
  with Phil
    op getNeighbour(out n:Phil)
end
```

Figure 2.1: Dining Philosophers interfaces

first version of this example to show the basic constructs in Creol, and later in Chapter 5 to show how it can be dynamically updated to the second version of the example.

The interfaces for the dining philosophers example are given in Figure 2.1. We see that the `Phil` interface defines two operations, borrowStick and returnStick. The `with Phil` part of the interface declaration means that these operations can only be used by an object which itself implements the `Phil` interface.

The code for the dining philosophers is given in Figure 2.2.

```
class Philosopher(butler: Butler)
  implements Phil
begin
  var
    hungry: bool := false,
    history: string := "",
    chopstick: bool := true,
    neighbour: Phil

  op think == await not hungry;
              {thinking...}history:=history+"t";
              await wait;!think

  op digest == await not hungry;hungry:=true;
               history:=history+"d";
               await wait;!digest

  op eat == var l : label
            await hungry; l!neighbour.borrowStick;
            await chopstick&l?();{eating...}history:=history+"e";
            hungry:=false; !neighbour.returnStick;
            await wait;!eat

  op run == butler.getNeighbour(;neighbour);
            !think || !eat || !digest

 with Phil
  op borrowStick == await chopstick; chopstick:=false
  op returnStick == chopstick:=true
end

class Butler
  implements Butler
begin
  var p1:Phil, p2:Phil, p3:Phil, p4:Phil, p5:Phil

  op run ==
    p1 := new Philosopher(this);
    p2 := new Philosopher(this);
    p3 := new Philosopher(this);
    p4 := new Philosopher(this);
    p5 := new Philosopher(this);

  with Phil
    op getNeighbour(out n:Phil) ==
      if caller = p1 then n:=p2
      else if caller = p2 then n:=p3
      else if caller = p3 then n:=p4
      else if caller = p4 then n:=p5
      else n:=p1 fi fi fi fi
end
```

Figure 2.2: Dining Philosopher classes

Chapter 3

The Creol Interpreter

We will in this chapter give a quick overview of rewriting logic, followed by a guide to the Creol interpreter specification. This specification uses an intermediate representation called CMC (Creol (virtual) Machine Code), and a simple translation is given that rewrites a Creol program to CMC. As part of the work on this thesis, a simple compiler was implemented using the bison and flex tools to automate the process of translating Creol to CMC. This work also led to some minor revisions of the Creol syntax to make the grammar belong to the LALR(1) class of context free grammars.

3.1 Rewriting Logic

To specify the semantics of our system we use rewriting logic [23] in the syntax of the Maude system [5]. The reason for using rewriting logic is that it is a system with well defined semantics, and it allows us to describe the rules of our programming language on a very high level while still being able to simulate execution of these rules so we can check our specification.

3.1.1 Operations and Terms

First of all we need to define the types we are working with, which are usually referred to as *sorts* in texts on rewriting logic. Given a set S of sorts which we have declared (e.g. things like Integer, Boolean, List ...), we can declare operations which have sort $S^* \times S$, meaning an operation with n arguments of sorts $S_0, S_1 ... S_n$ and a value sort S which is the sort of the result. As a special case we have constants which are operations with no arguments. We declare operations in Maude using the `op` keyword as follows (Nat is the sort of the natural numbers including 0):

```
op zero : -> Nat [ctor] .
op successor : Nat -> Nat [ctor] .
op sum : Nat Nat -> Nat [assoc comm id: zero] .
```

note the `ctor` *attribute* which indicates to the Maude system that this operation is used to construct the ground terms of the system, we will get back to this in a moment. Other attributes that Maude understands is `assoc` to indicate that the operation is associative, and `comm` to indicate it is commutative. The `id:` attribute indicates a constant which acts as the identity element of the operation. These attributes are not strictly necessary, but makes it easier to write specifications because we don't have to use parentheses everywhere if we declare operations to be associative, and it helps Maude to execute the rules of the specification in an efficient way.

With these operations we can construct terms which denote the data our rules will operate on. Since all operations are strongly typed, each term will also have a unique type which can be used to classify it. Terms are built from the operations we have just defined and from a set of variables X with sorts in the following way: Every constant and every variable is a term, and every operation applied to a list of terms with sorts matching the sorts of the arguments for that operation produces a new term which has sort equal to the value sort of the operation we used to build the term. For example `successor(zero)` is a term, and if we declare a variable `var N : Nat.` then `sum(successor(zero), N)` is a term. We call terms which have no variables in them *ground terms.*

3.1.2 Equations and Reduction

Once we have defined the terms, we can start to make computations with these terms by giving equations. Computation proceeds by *matching* the left hand side of an equation to a term, and then replacing that term with the right hand side of the equation. This procedure is called a reduction step, and it's required that repeated application of our equations always result in a normal form, that is a term which cannot be reduced further. It's implicit in this that a computation using our system of equations always terminates, since a non-terminating sequence of reduction steps would mean that some term does not have normal form. In addition to this, it's required that our equations are *confluent.* Confluence means that, given some starting term, no matter which order we apply the reductions in, we end up with the same normal form. More specifically the matching process yields a *substitution* of variables to ground terms which we will write in the following way: $\sigma = \{X_0 \mapsto t, X_1 \mapsto t'...\}$. A reduction step then consists of replacing the term matching the left hand side of an equation with the term obtained by applying the matching substitution to the right hand side of the equation. An example of equations defining the semantics of summation on the natural numbers is as follows:

```
eq sum(zero, N) = N .
eq sum(successor(N), N') = successor(sum(N, N')) .
```

In the Maude system we also have conditional equations on the form `ceq t = t' if condition .` where the condition can be either another equation or a membership test which tests if a term is of a given sort. We will not define formally exactly how this works, as it's only used in a very basic and intuitive way in this thesis.

3.1.3 Rules and Rewriting

The equations are part of the functional sub-language of the more general rewriting logic. The main difference between a rewriting rule $t \to t'$ and an equation $t = t'$ is that the rewriting rules do not have to be terminating or confluent. Conceptually the difference is that the equations specify reductions of a term to an equivalent (but simpler) term, while a rewriting rule specifies the evolution of the system from one state to another. As such, the term that results from a rewriting step does not have to be equivalent to the term that was rewritten. Given a set of rewriting rules `rl t => t' .` (or `crl t => t' if condition .` for conditional rewriting rules), the Maude system's command `rew [n] t` works by reducing t to normal form, then applying some (selected in a pseudo-random fashion) matching rewriting rule, reducing the result to normal form by applying the equations, selecting another matching rewriting rule and so on until no rewriting rule matches or the maximum number of rewrite steps (that is, the bracketed n in the command) have been reached. The rewrite rules are applied to subterms which match the left hand side of the rewrite rule in a non-deterministic way (different execution strategies can be specified using meta-level programming in Maude, we are assuming a non-deterministic strategy while the built-in 'fair' rewrite strategy in Maude is deterministic and not very fair).

3.2 The Interpreter

Because rewriting logic is a system with formal semantics, the Creol interpreter provides a formal operational semantics for execution of Creol programs. We will here review the internal representation of classes, methods, objects and processes as terms in the rewrite theory, and the semantics of method calls. The Maude specification of the interpreter can be found in Appendix B.

3.2.1 Representation of Classes, Methods, Objects and Processes

Creol objects are represented in rewriting logic as terms on the form 3.1, where O and C are the unique object identifier and the class identifier respectively, PR is the active process, W is a set of waiting processes, A is a set of object attributes and N is a counter used to produce unique labels for asynchronous method calls.

$$< O : Ob|Cl : C, Pr : PR, PrQ : W, Att : A, Lcnt : N > \tag{3.1}$$

Creol processes are 4-tuples on the form 3.2 where M is the method identifier of the originating method, B is a boolean which indicates if the process is *newly bound* meaning that no statements have been executed yet, P is a *program* in the form of a sequence of statements joined by composition, non-deterministic choice or non-deterministic merge operators, and L is a set of local variables for this process.

$$process(M, B, P, L) \tag{3.2}$$

The representation of Creol classes are on the form 3.3, they are terms consisting of a unique class identifier C, a list of superclasses S, a list of class attributes (with optional initialisers) IA, a set of methods $MMTD$ and a counter F used to create unique object identifiers when objects of this class are created with a **new** statement.

$$< C : Cl | Inh : S, Att : IA, Mtds : MMTD, Ocnt : F > \tag{3.3}$$

Creol methods are represented as terms on the form 3.4, consisting of a method name R, a set L of local attributes with optional initialisers and a sequence P of statements.

$$< R : Mtdname | Latt : L, Code : P > \tag{3.4}$$

A *configuration* is a set of classes, objects, messages and call queues which together gives the entire state of the running program. This is represented as a single term in rewriting logic.

3.2.2 Semantics of Method Calls

We will now briefly look at the semantics of method calls in the interpreter and we will only consider the asynchronous external call, as the synchronous case can be seen as an asynchronous call which immediately after the call waits for the return and internal calls can be seen as external calls on the current object. Because of these equivalences we only have to consider the most general form of method calls which is the external asynchronous call. The first thing that happens is that the return label is assigned the current method call counter and this number is added to the set of completion messages to keep. This step is only taken if there is a label, so if we are not interested in the reply then we know that we can safely discard the call completion message.

```
eq < O : Ob | Cl: C, Pr: process(M, B, (Q ! M'(I)); P, L),
              PrQ: W, Att: A, Lcnt: N >
   < O : Qu | Ev: MM, Keep: H >
 = < O : Ob | Cl: C, Pr: process(M, B, (Q := int(N)); (! M' (I)); P, L),
              PrQ: W, Att: A, Lcnt: N >
   < O : Qu | Ev: MM, Keep: H ; [N] > .
```

The next thing that happens is that a method invocation message is sent to the object we want to call the method on. Here `OE` is an expression which is evaluated in the context of the class attributes `A` and the local variables `L` to find the object identifier of the receiver. The next part of the invocation message is the method identifier, and finally we have the list of parameters. This list always starts with the object identifier of the caller and the unique call number (which is increased once the invocation message has been emitted). After this follows the list of explicit parameters which are evaluated in the context of the class attributes and the local variables.

```
rl [remote-async-reply] :
  < O : Ob | Cl: C, Pr: process(M, B, ( ! OE . Q(I)); P, L),
             PrQ: W, Att: A, Lcnt: N >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: N + 1 >
  invoc(eval(OE, (A, L)), Q, (O int(N) evalList(I, (A, L)))) .
```

The invocation message then floats around in the configuration until it reaches the message queue of the receiver. A simple transport rule moves it into the queue.

```
rl [invoc-msg] :
  < O : Qu | Ev: MM, Keep: H > invoc(O, M, DL)
=>
  < O : Qu | Ev: MM invoc(O, M, DL), Keep: H > .
```

The next thing that happens is the virtual binding. To accomplish this a bindMtd message is emitted. This message is not meant to correspond to a real message sent over a network, but merely provides an easy way to match the call up with the classes that are floating in the configuration. The first three parameters in the bindMtd message are pretty self explanatory, but the last parameter is a list of classes to search for the method. Initially this is just the most specific class of the object.

```
rl [receive-call-req] :
  < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N' >
  < O : Qu | Ev: MM invoc(O, Q, DL), Keep: H >
=>
  < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N' >
  < O : Qu | Ev: MM, Keep: H >
  bindMtd(O, Q, DL, C[nil]) .
```

The actual binding is performed by searching the classes in the list in a left first, depth first order. Each time the class at the front of the list does not contain the method sought after, its subclasses are prepended to the list of classes which will be searched. It's easy to change the semantics of virtual binding at this point to use a breadth first search or another strategy if desired. The result is a boundMtd message which contains the new process resulting from binding the method.

```
eq bindMtd(O, Q, I, ((C)[DL]) S')
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 = if Q in MMTD then
     boundMtd(O, newProcess(Q @ C, get(Q, MMTD), I))
   else
     bindMtd(O, Q, I, S S')
   fi
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > .
```

Finally the bound method is included in the process queue of the receiving object. Originally this was a rewrite rule so it did not have to happen immediately, but to simplify making updates it's now an equation which ensures that bound method messages are always transient in the configuration.

```
eq boundMtd(O, process(M, B, P, L))
   < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
 = < O : Ob | Cl: C, Pr: PR,
              PrQ: process(M, B, clear(P), L) : W, Att: A, Lcnt: N > .
```

3.2.3 Activating Processes from the Process Queue

Several rules and equations in the interpreter specification deal with moving processes in and out of the process queue. These are of particular importance to our update system, as the points where these rules apply are exactly the points where we will consider it safe to make an update. Thus we want to control the usage of these rules, so an object which is a candidate for being updated does not use these rules, but rather the rules given in the update model. To accomplish this a subsort of object identifiers which correspond to up-to-date objects has been defined, and these rules have been modified to only apply to up-to-date objects.

First of all we have an equation and a rule that implement a mechanism to prioritise internal synchronous calls. Instead of just suspending the calling process and letting the system continue choosing processes to execute non-deterministically, the calling process is swapped with the called process and a special `continue` statement is added to the called process' program. This is done with an equation to make sure it happens before the normal rewriting rules for method calls can be applied. The `continue` statement will ensure that when the called process finishes, it is the calling process that will be loaded into the active process by the `continue` rule.

```
eq < UTD : Ob | Cl: C, Pr: process(M, B, (N ? (J)); P, L),
                PrQ: process(M', true, P',
                             (('caller : UTD), ('label : int(N)), L')) : W,
                Att: A, Lcnt: N' >
 = < UTD : Ob | Cl: C, Pr: process(M', false, P' ; continue(N),
                                   (('caller : UTD), ('label : int(N)), L')),
                PrQ: process(M, B, await N ? ; (N ? (J)); P, L) : W,
                Att: A, Lcnt: N' > .

rl [continue] :
  < UTD : Ob | Cl: C, Pr: process(M, B, continue(N), L),
               PrQ: process(M', B', ((N ?(J)); P), L') : W, Att: A, Lcnt: N' >
=>
  < UTD : Ob | Cl: C, Pr: process(M', false, ((N ?(J)); P), L'),
               PrQ: W, Att: A, Lcnt: N' > .
```

Second we have the conditional rule that suspends processes which are waiting on a guard. It uses a helper function `enabled` that checks the guards for the code. Note that we want processes to be suspended as usual in outdated objects, so there is no requirement that the object identifier is up-to-date with this rule.

```
crl [suspend] :
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
=>
  < O : Ob | Cl: C, Pr: none,
             PrQ: W : process(M, B, clear(P), L), Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
if not enabled(P, (L , A), MM) .
```

Finally we have to rule to load suspended processes when their guard becomes true. This rule also loads newly bound processes as the `enabled` function is automatically true for all statements which are not guards.

```
crl [PrQ-enabled] :
  < UTD : Ob | Cl: C, Pr: none, PrQ: process(M, B, P, L) : W, Att: A, Lcnt: N >
  < UTD : Qu | Ev: MM, Keep: H >
=>
  < UTD : Ob | Cl: C, Pr: process(M, false, P, L), PrQ: W, Att: A, Lcnt: N >
  < UTD : Qu | Ev: MM, Keep: H >
if enabled(P, (A , L), MM).
```

3.2.4 Summary of Changes Made to the Interpreter

In order to accomodate the update system, some changes had to be made to the interpreter. However, as a general rule I have tried to avoid making changes to the interpreter and keeping the changes local to the update module. A short summary of the changes is given here, but the motivation for these changes is discussed in more detail in the following chapters.

- Explicit process terms have been introduced to make things clearer and to keep track of which method a process came from and if it's newly bound or not. Keeping track of which method the process came from is absolutely essential, but the "newly bound" flag is a convenience, as it's possible to compare the program in the process with the program in the method to determine if any execution has taken place.
- Up-to-date object identifiers have been introduced as a subsort of object identifier and the rules which move waiting processes into the active process have been modified to only match on up to date objects.

- Moving processes from boundMtd messages into the waiting process queue of an object has been changed from a rewrite step to an equation. This is allowed because no real communication takes place to do this step (the bindMtd/boundMtd messages would never move from one node to another in a real system). Doing this simplifies updating a configuration since only invoc and comp messages have to be taken into account.

- Several other minor changes have also been made to make the specification more readable, including changes to variable names and introduction of explicit sorts Aid and AidList for attribute identifiers.

3.3 Translation from Creol to CMC

As part of the work on this thesis is a very simple lex/yacc[1] based compiler has been developed. The main purpose of developing this compiler was to speed up the process of making example programs as the manual translation of programs from Creol to the CMC syntax was tedious and error prone. The grammar also proved useful when new syntactic constructs for updates were introduced, as it made it possible to check automatically that no ambiguities were introduced. The work on the compiler has led to some modifications in the Creol grammar, as the precedence of operators had to be clarified and certain parts of the original syntax made the grammar ambiguous with only one token lookahead (which is a common requirement for efficient parsing of the code). The full grammar that the compiler is based on is presented in ISO-EBNF form in Appendix A.

[1]Actually we use the GNU tools: flex and bison

Chapter 4

State Transfer

The process of updating a program can naturally be divided into *state transfer* and *control.* We will in this chapter deal with the problem of state transfer before tackling the problem of how to control the update process in Chapter 5. Before describing the implementation of the state transfer process, we will look at how we can divide the state of a running program into smaller units to enable updates of a finer granularity. First of all we will define the notion of *scope*, and then we will look at state transfer functions defined on these scopes.

4.1 Scopes

We define a scope to be a valuation of a set of variables associated with a program sequence that can refer to these variables. The variables are not available outside of the program sequence associated with the scope. Multiple scopes can be associated with the same program sequence, for example when we have multiple threads of execution invoking the same procedure or multiple class instances in an object oriented system. All scopes are disjoint, and the union of all scopes gives the entire state of a program. There are two kinds of scopes, transient scopes are instantiated when control enters the associated program sequence and are removed once control exits that sequence. This corresponds with the notion of a Creol *process.* Persistent scopes retain their value when control exits the associated sequence. Objects in object oriented systems are special cases of persistent scopes, since instance variables retain their value between method invocations on the same object. In particular Creol *objects* are persistent scopes.

4.2 State Transfer Functions

Once we have an old scope and a corresponding new scope, we need a function to transfer the state between the two versions of the program. These functions can be completely general, allowing them to call other functions, or they can be restricted to simple expressions. There are also issues related to how we program these functions: Do we use the same language as the program is written in, a subset of it or a whole new language? If we use the same language, we probably have to add some constructs to be able to disambiguate references to variables with the same name in the old and new version. We also have to consider if we want to make the old variables read-only or the new variables write-only. These decisions are interrelated, for example if we make old variables read-only and new variables write-only we do not have to disambiguate old and new versions of variables because variables on the left hand side of an assignment always have to refer to the new version and on the right hand always to the old version.

4.3 The Unit of Replacement in Creol

Although it is possible to make program replacements on a lower level than scopes, it's very hard to do any kind of reasoning about the correctness of such updates. The unit of replacement chosen for Creol is a class, although it is trivial to replace only one of the methods in a class, we package an update as an update to the entire class. The reason for choosing this granularity is partly to simplify the implementation since updating individual methods of a class becomes just a special case of updating the entire class and partly that the mechanism for finding the objects which need to have their state converted is class-based.

4.4 Updating Classes

To begin an update in Creol we have a function which takes the current configuration and a configuration containing the updated classes. The function then produces a new configuration by merging the classes of the old configuration with the new classes and outdating the old configuration. The `classes` function simply separates out the classes from the objects, messages and queues. We also make some temporary *ghost* objects which represent the objects that have not yet been converted. This is necessary to help us locate the actual object when we apply several updates in succession and there are many configurations, as will be explained later.

```
op update  : Configuration Configuration -> Configuration .
eq update(CONFIG, CONFIG')
 = merge(classes(CONFIG), CONFIG')
   makeGhosts(CONFIG)
   outdated(CONFIG) .
```

Merging the classes means that we supplement the new classes contained in the update with any unchanged classes from the old configuration. We could have avoided this step by including both old and new classes in the update, but in a real implementation you probably want to make the updates as small as possible since they might be sent over a network.

```
finish list  = "finish",  identifier, { ',', identifier }
restart list = "restart", identifier, { ',', identifier }

parameter convert = "convert", definitions ;

class convert = "convert", "==", [ "var", definitions ], statements,
                [ finish list ], [ restart list ] ;

method signature = "op", identifier,
                   [ ( "(", declarations, ["out", declarations], ")"
                     | "(", [declarations], "out", declarations, ")" ) ] ;

method definition = method signature, [ parameter convert ], "==",
                    [ "var", definitions ], statements ;

class declaration = "class", identifier,
                    [ "[", declarations, "]" ], [ "(", declarations, ")" ],
                    ["implements", identifier, { ',', identifier }],
                    ["inherits", inheritance, { ',', inheritance } ],
                    "begin",
                    [ "var", definitions ],
                    [ class convert ],
                    { method definition },
                    "end" ;
```

Figure 4.1: Creol EBNF syntax extended with class conversion

```
op merge : Configuration Configuration -> Configuration .
eq merge(none, CONFIG') = CONFIG' .
eq merge(< C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > CONFIG, CONFIG')
 = if not (C in CONFIG') then
     < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
   else none fi
   merge(CONFIG, CONFIG') .
```

4.5 Outdating Objects

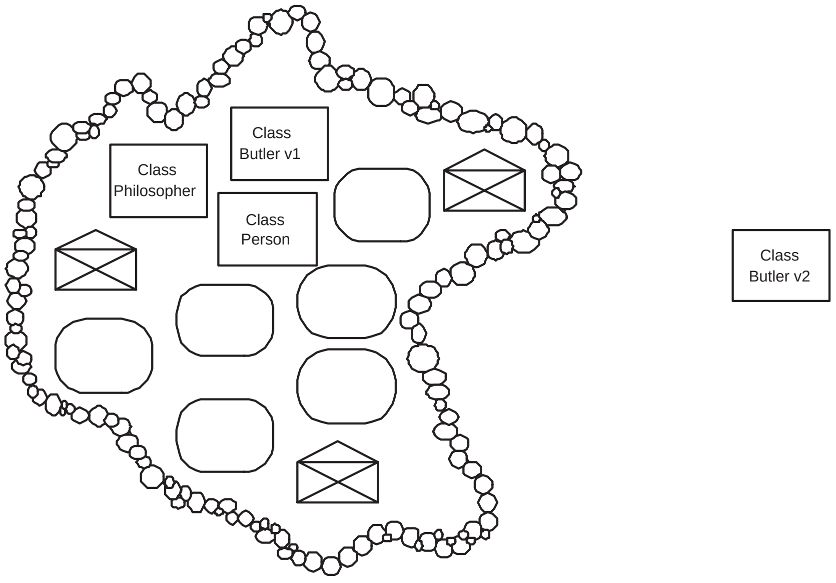
Outdating the old configuration is simply a matter of wrapping it inside a new term so it does not interfere with the new configuration. If we compare this with the methods shown in Figure 1.1, the method employed here is a kind of membrane as opposed to a barrier. We keep objects which are not up-to-date on one side of the membrane until they reach a quiescent state and then we migrate them to the new configuration. This allows for a lazy implementation of state transfer where quiescent objects are kept on the outdated side of the membrane until a method is invoked on them. In our implementation, the non-deterministic nature of the rewriting system means objects are migrated concurrently with executing code in the new configuration. The process is illustrated in Figure 4.2.

In Creol, invariants are satisfied at processor release points, so in order to keep any reasoning control we need to continue executing code in the outdated configuration at least until we reach a processor release point. To be able to control what happens once we reach such a point, we tag the objects and their corresponding queues in the outdated configuration. The rules for activating waiting processes in the interpreter have been modified so they only apply to objects without this tag. Thus the only rules that can activate processes in tagged objects are the ones in the update module. Tagging outdated objects is done by the following equation (note that this is not a rewrite rule so it always happens before any of the rules for activating waiting processes are considered):

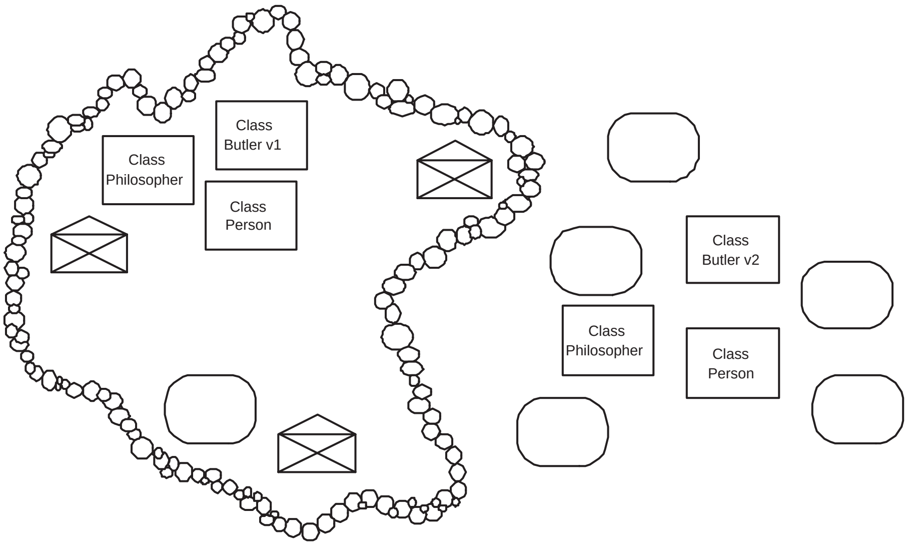
```
subsorts Qid < UTDOid < Oid .
op outdated : Qid -> Oid  [ctor] .

eq outdated(< UTD : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
            < UTD : Qu | Ev: MM, Keep: H >
            CONFIG)
 = outdated(< outdated(UTD) : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
            < outdated(UTD) : Qu | Ev: MM, Keep: H >
            findInheritance(outdated(UTD), (C)[nil], nil, missing)
            CONFIG)
   findInheritance(outdated(UTD), (C)[nil], nil, nil) .
```

In this step we also find the complete list of classes (including version numbers) that the object belongs to, both in the old and new configurations. If these are identical, the object is already up to date and can be moved directly into the new configuration without waiting for a processor

First the old configuration is wrapped in a membrane and put together with the updated classes to form a new configuration



Then the classes which are unchanged and objects which do not belong to any changed class are moved out to the new configuration



Finally objects and messages that need conversion migrate through the membrane concurrently with execution in the new configuration

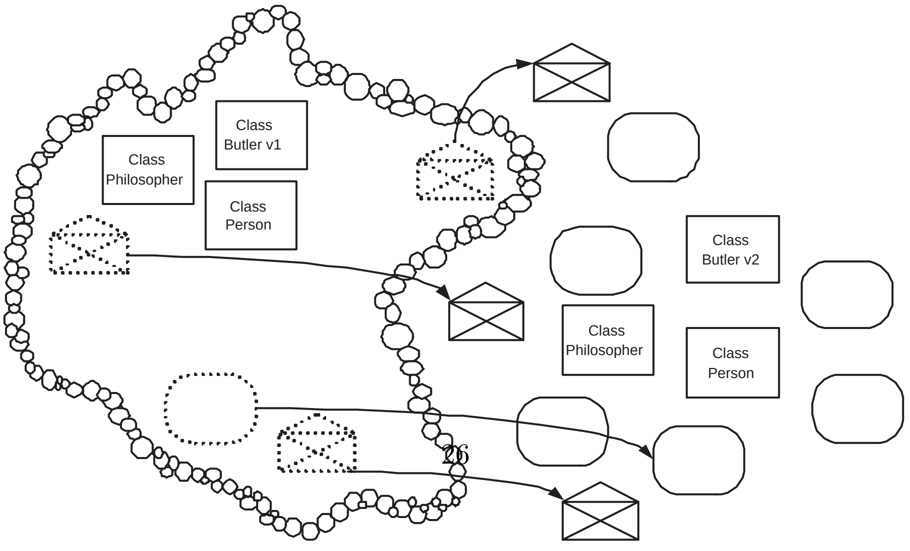


Figure 4.2: Three stages of an update

release point. These lists are also used to determine which parts of the object's state have to be converted and which can simply be copied. We also make a list of methods that should be finished before conversion takes place, which we will get back to in Chapter 5. This list needs to be built in the new configuration and then transferred to the outdated configuration where it will be used to guide execution. This is the role of the `missing` list constructor which is used in the last equation to fill in finishlist in the outdated configuration.

```
op missing : -> AidList [ctor] .
op findInheritance  : Oid InhList InhList AidList -> Msg [ctor] .
eq findInheritance(O,(((C)[DL]) S'), S'', AL)
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 = findInheritance(O,(S S'), ((C # VN)[DL]) S'', AL finishlist(MMTD))
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > .

eq ((C)[DL]) S ((C)[DL]) S' = ((C)[DL]) S S' . *** remove duplicates

eq outdated(findInheritance(outdated(UTD), nil, S, missing)
            CONFIG)
   findInheritance(outdated(UTD), nil, S', AL)
 = outdated(findInheritance(outdated(UTD), nil, S, AL)
            CONFIG)
   findInheritance(outdated(UTD), nil, S', AL) .
```

4.6 Migrating Objects

We now turn to the workings of the membrane. There are two cases, objects which belong to the same versions of all classes in both configurations can be transferred immediately. Objects which belong to one or more classes which have been updated need to have their state transferred. The first case is very simple and is covered by the following rule:

```
rl [migrate-object] :
  outdated(
    < outdated(O) : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
    < outdated(O) : Qu | Ev: MM, Keep: H >
    findInheritance(outdated(O), nil, S, AL)
    CONFIG)
  findInheritance(outdated(O), nil, S, AL)
  < O : Ghost | Cl: C, Old: false >
=>
  < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
  outdated(CONFIG < O : Ghost | Cl: C, Old: true >) .
```

Here we see that the ghost of the object changes place with the actual object, and the state of the ghost changes to indicate that the actual object is in a more recent configuration than the ghost. This information can be used when objects in different configurations need to communicate with each other.

The rule for converting objects is more complex as we need to figure out exactly which classes that the object belongs to that have been updated. The classes that the object belongs to that are unchanged should have their state copied, but we cannot simply copy all state of the old object since the inheritance graph may have changed and some state should be discarded. We make use of a function `combine` which takes the new and old inheritance lists and creates a list where the old version number of each class in the new inheritance graph is recorded. In this way the conversion process can take one class at a time in this list and check if it's up-to-date, if not its state needs to be converted otherwise it is copied. Note that this rewriting rule only applies to objects which have no active process. We also use some simple functions to throw any invocation messages in the queue back out in the outdated configuration while keeping the completions, this will be discussed in Section 4.13.

```
rl [convert-object] :
  outdated(
    < outdated(O) : Ob | Cl: C, Pr: none, PrQ: W, Att: A, Lcnt: N >
    findInheritance(outdated(O), nil, S', AL')
    < outdated(O) : Qu | Ev: MM, Keep: H >
    CONFIG
  )
  findInheritance(outdated(O), nil, S, AL)
  < O : Ghost | Cl: C, Old: false >
=>
  convert(O, combine(S, S'), A, W)
  < outdated(O) : Ob | Cl: C, Pr: none,
                       PrQ: paramcopy(W, 'null, none),
                       Att: ('this : O), Lcnt: N >
  < outdated(O) : Qu | Ev: comps(MM), Keep: H >
  outdated(CONFIG invocs(MM) < O : Ghost | Cl: C, Old: true >) .

op combine : InhList InhList -> InhList .
eq combine((C # VN)[DL], ((C' # VN')[DL']) S)
 = if (C == C') then (C' # VN')[DL'] else combine((C # VN)[DL], S) fi .
eq combine((C # VN)[DL], nil) = (C # VN)[DL] .
eq combine(((C # VN)[DL]) S, S') = combine((C # VN)[DL], S') combine(S, S') .
```

4.7 Quiescence and Internal Synchronous Calls

The rules for internal synchronous calls load the called procedure directly into the active process and move the active process into the process queue. Then a `continue` statement is added at the end of the called procedure which will load the process that made the call when the called procedure has finished. This means that during internal synchronous calls the active process never becomes empty and thus we cannot start converting the object. The semantics of Creol states that *any* call is a potential processor release point (since the called procedure can make asynchronous calls), so we can safely perform conversion at these points. In the full specification of the update system, there is a rule almost identical to that given in the previous section which starts conversion if the active process contains a single `continue` statement. We've also changed the continue rule of the interpreter to require up-to-date objects, since it loads a process from the process queue. When we convert objects which have a `continue` statement in their active process, we simply throw it away since the other rules for loading processes will eventually load the continuation anyway. In effect, the `continue` statement is a hint to the runtime system about priorities of the waiting processes, so we can safely discard it.

4.8 State Conversion

The convert-object rule in Section 4.6 sets up the conversion procedure which then proceeds in stages. For each class the object is an instance of, we either copy the attributes of the object that came from that class if it was not updated, or we load a conversion procedure from the updated class. Notice that the rule which sets up the conversion moves all old attributes and processes out of the object. The updated object is built up again in stages, so that at each stage all superclasses of the class that is currently being processed have already been processed.

In practise there are several methods available for referring to the attributes of the old object when performing the conversion. The most obvious one is maybe to pass a reference to the outdated object as a parameter to the conversion procedure. However, this is inelegant as it requires the type system of the compiler to be aware that you are referring to different versions of a type. It's also problematic because you are normally allowed to call methods on an object you have a reference to, but this is not clearly defined with an outdated object. Finally, in Creol, all references are typed by interface, so it would be rather strange to introduce a special construct for referencing a class. We felt that introducing a special qualifier was a better choice since it clearly indicates the special semantics of referring to another version of an attribute, there is no aliasing of attributes of superclasses, and you can't make method calls on outdated objects. Thus, when the conversion process is started for a certain class we take all the old attributes belonging to that class and put them in the local variables of the conversion process but with their qualifier changed to `old` by a function appropriately named `age`.

```
op age : Subst Cid -> Subst .
eq age((((Q @ C) : D), A) , C')
 = (if C == C' then ((Q @ 'old) : D) else no fi) , age(A, C') .
eq age(((Q : D), A) , C') = ((Q @ 'old) : D) , age(A, C') .
eq age(no, C') = no .
```

The `copy` function copies the attributes from a class, but there is a catch. To support changes in the inheritance graph what we actually do is to use the initialiser list from the class and then assign the old values to any attributes which are already present in the attribute list. This could also be used to provide a default conversion operation for classes which should be converted if none is present in the class, but the current system requires that conversion operations are present for updating a class - in practise this is not a problem since the compiler can automatically generate a conversion function equivalent to the copy function.

```
op copy : Subst Subst -> Subst .
eq copy(((X : D), A) , A')
 = if (X in A') then ((X : val(X, A')), copy(A, A'))
                else ((X : D), copy(A, A')) fi .
eq copy(no, A') = no .
```

When an object is updated with conversion, the rule creates a new process from the conversion procedure in the updated class and gives it the aged attributes from the old class. This process is then executed by the normal interpreter rules until the special endconvert statement is hit. This statement is the subject of another rule which will be covered later.

```
crl [update-object-with-conversion] :
  < C # VN : Cl | Inh: S, Att: IA,
                  Mtds: < Convert | Latt: L, Code: P,
                                    Finish: AL, Restart: AL' > * MMTD,
                  Ocnt: F >
  convert(O, ((C # VN')[DL]) S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: none, PrQ: W, Att: A, Lcnt: N >
=>
  < C # VN : Cl | Inh: S, Att: IA,
                  Mtds: < Convert | Latt: L, Code: P,
                                    Finish: AL, Restart: AL' > * MMTD,
                  Ocnt: F >
  convert(O, S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: process('convert, false,
                                           P ; endconvert(C),
                                           (L, age(A', C))),
                       PrQ: W, Att: (A, evalSS(IA, A)), Lcnt: N >
if VN > VN' .
```

If the class which is currently being processed is unchanged from the old version we use the rule that updates without conversion. The interesting case is when a class has newly been introduced in the inheritance graph of the object by an update to another class, in this case the copy function makes sure the attributes are initialised to default values. However, it is the responsibility of the convert procedure of the class which introduced the new class into the inheritance graph to do any extra initialisation which is required. We can not possibly pass parameters to the class construction because the class which is inheriting from the introduced class has yet to be processed so we would not know what those parameters were. We also see that the process queue has a term added which uses the function paramcopy which we will discuss shortly.

```
crl [update-object-without-conversion] :
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  convert(O, ((C # VN')[DL]) S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: none, PrQ: W, Att: A, Lcnt: N >
=>
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  convert(O, S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: none, PrQ: W : paramcopy(W', C, MMTD),
                       Att: (A, copy(evalSS(IA, A), A')), Lcnt: N >
if VN' = VN .
```

4.9 Conversion of Call Queues

Since the parameters of a method might have changed with an update, a way of converting the parameters of pending method calls[1] must be provided. This is done by declaring a conversion inside the scope of an operation declaration in a class. We have decided to limit the conversion to providing initialiser expressions for the parameters of the new routine in terms of the parameters of the old version (qualified with 'old'), constants and `this`. Early versions of the update system also allowed the use of object attributes in these expressions, but this created problems when there are several updates happening at the same time, as a method call can be to an object two versions away and parameter conversion has to happen twice before the object can receive the call. In this case the object attributes corresponding to the middle version would not be available when converting the parameters of the call. Finally, it was decided that the lazy update mechanism which allows several layers of updates was more important than a more general way of doing parameter conversion.

```
rl [finish-conversion] :
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  convert(O, S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: process('convert, false, endconvert(C), L),
                       PrQ: W, Att: A, Lcnt: N >
=>
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  < outdated(O) : Ob | Cl: C', Pr: none,
                       PrQ: W : paramconv(W', C, MMTD, ('this : O)),
                       Att: A, Lcnt: N >
  convert(O, S', A', W') .
```

[1]In the current Creol interpreter pending method calls correspond to newly bound processes in the process queue.

4.10 Inconvertible State

The remaining state of the object which we can't convert with the methods so far is considered inconvertible. This means processes which are suspended on a guard. There are several ways to deal with this, which we will get back to in the next chapter. For example conversion operations can be made for local variables and the semantics of labels can be extended to provide a mapping between processor release points in different versions of the same function. This has been attempted for example in the DYMOS system [19]. Another option used in the Argus system [3] is to cancel all old code, because distributed applications need to provide error recovery mechanisms anyway so most transactions which are interrupted by an update can be resumed. We will not take this approach as it doesn't work well with internal calls in Creol which are known to always complete and do not necessarily have error recovery code. But no matter what you do, the basic problem remains: There is no one to one mapping between execution state of different versions of code. The simplest solution, which is the one we have adopted as the default for the Creol system is to do nothing and continue executing the old code. This can potentially lead to errors, which we will investigate in Chapter 6.

4.11 Implementation of Parameter Conversion

We keep track of which processes in the process queue of an object are newly bound and thus eligible for conversion. The parameter conversion takes place at the same time as the state transfer, after the state belonging to a certain class has been converted or copied, the processes belonging to that class are subjected to parameter conversion. We have three cases, the first case is the one where we are converting a process that is newly bound (`B == true`), here we make a new process and initialise the local variables with the local variables of the old process subjected to parameter conversion. We use a small helper function `conv` to perform the actual conversion, where we check if there is a parameter conversion defined for the method that the process is an instance of. If there is a conversion, we evaluate the initialiser list within the context of the aged parameters and the class attributes. If the updated method does not have a parameter conversion, we copy old attributes into the new ones with the same name. A small helper function called `toList` makes a list from the substitution because that's what we need to create the new process. The second case deals with filtering processes belonging to other classes which are already processed or will be processed later by discarding them. The third case deals with processes which are not newly bound, which we will just continue executing as is[2].

```
op conv : Subst Mtd Subst -> Subst .
eq conv(L, < M : Mtdname | Latt: L', Code: P' >, A)
 = copy((('caller : 'null), ('label : 'null), L'), L) .
eq conv(L, < M : Mtdname | Latt: L', Code: P', Convert: IL >, A)
 = evalSS(IL, (age(L,'null), A)) .
```

[2]This has potential for creating problems, but as will be seen in the next chapters we can wait for these to complete before converting the object or we can restart them, so we have some possibilities to control the process.

```
op program : Mtd -> ProgList .
eq program(< M : Mtdname | Latt: L', Code: P' >) = P' .
eq program(< M : Mtdname | Latt: L', Code: P', Convert: IL >) = P' .

op paramconv : MProg Cid MMtd Subst -> MProg .
eq paramconv(none, C, MMTD, A) = none .

ceq paramconv(process(M @ C, B, P, L) : W, C, MMTD, A)
  = process(M @ C, true, program(get(M, MMTD)), conv(L, get(M, MMTD), A)) :
    paramconv(W, C, MMTD, A)
 if (M in MMTD) and (B == true) .

ceq paramconv(process(M @ C, B, P, L) : W, C', MMTD, A)
  = paramconv(W, C', MMTD, A)
 if not (C == C') .

ceq paramconv(process(M @ C, B, P, L) : W, C, MMTD, A)
  = process(M @ C, B, P, L) : paramconv(W, C, MMTD, A)
 if (B == false) .
```

If the class has not been updated we simply copy the processes belonging to that class using the following simple function.

```
op paramcopy : MProg Cid MMtd -> MProg .
eq paramcopy(process(M @ C, B, P, L) : W, C', MMTD)
 = if (C == C') and ((M in MMTD) or C == 'null) then
     process(M @ C, B, P, L)
   else
     none
   fi : paramcopy(W, C', MMTD) .
eq paramcopy(none, C, MMTD) = none .
```

4.12 A Simple Example of State Transfer in Creol

A simple recursive procedure to compute the Fibonacci numbers is given in Figure 4.3, it will serve as an example of a program which we can successfully update with the techniques discussed so far. The motivation for doing so is that this method of computing the Fibonacci numbers is extremely slow as it generates an exponential amount of recursive calls. We want to replace the procedure with a linear time procedure to compute the Fibonacci numbers as given in Figure 4.4 without interrupting the system - let's just assume there is a big air traffic control system which cannot be stopped and relies on computing Fibonacci numbers for some purpose.

```
class Fibonacci
begin
var val : int

op fibonacci(n : int out result : int) ==
  var
    a : int, b : int
  if n = 0 then result := 0 else
    if n = 1 then result := 1 else
      fibonacci(n - 1 ; a) ; fibonacci(n - 2 ; b) ; result := a + b
    fi
  fi

op run == fibonacci(15 ; val)

end
```

Figure 4.3: Exponential time method for computing Fibonacci numbers

```
class Fibonacci
begin
var val : int
convert == val := val@old

op helper(a : int, b : int, c : int out result : int) ==
  if c = 0 then result := a else helper(b , a + b, c - 1 ; result) fi

op fibonacci(n : int out result : int) == helper(0, 1, n ; result)

op run == fibonacci(15 ; val)

end
```

Figure 4.4: Linear time method for computing Fibonacci numbers

We first compile the two versions of our `Fibonacci` class with the Creol compiler. When compiling the update we need to indicate that this is a newer version so the class gets the correct version number in the CMC representation.

```
creolc --version 1 <fib_v1.creol >fib_v1.maude
creolc --version 2 <fib_v2.creol >fib_v2.maude
```

We add the line `frew [10000] init . quit` to the end of fib_v1.maude and then run the specification through the Maude tool and write the resulting intermediate configuration to a file with the following command:

```
maude interpreter.maude fib_v1.maude >fib_update.maude
```

we then edit the resulting file a bit by removing Maude's welcome message up to and including `result (sort not calculated):` and replace this by `op fib1 : -> Configuration [ctor] . eq fib1 =`. At the end of the file where Maude has said `Bye.` we replace this with just a period. After this we write `op fib2 : -> Configuration [ctor] . eq fib2 =` and copy the definition of the updated fibonacci class from the file `fib_v2.maude`. Finally we insert the module definition `mod FIB-UPDATE is` and the line `pr UPDATE .` at the top of the file and end the file with `endm`. The result of this is that we now have a file which can be used to test the effect of applying an update to the configuration reached after 10000 rewrite steps of the original program. This process could of course be automated with a little script, but it doesn't take all that long.

We are now ready to test the effect of applying our update, and start Maude again, this time in interactive mode. We import the interpreter, the update module and our example module and then we ask to rewrite the term `update(fib1, fib2)` which performs the update and continues the process of computing the `fibonacci` function.

```
Maude> in interpreter.maude
Maude> in update.maude
Maude> in fib_update.maude
Maude> frew update(fib1, fib2) .
frewrite in FIB-UPDATE : update(fib1, fib2) .
rewrites: 13541 in 290ms cpu (450ms real) (46693 rewrites/second)
result Configuration:
{Class definition and message queue removed}
< 'Fibonacci1 : Ob | Cl: 'Fibonacci,Pr: none,PrQ: none,
                     Att: ('this : 'Fibonacci1),
                          ('val @ 'Fibonacci) : int(610), Lcnt: 1067 >
```

For comparison, if we just rewrite the configuration we had when we applied the update (fib1), we end up having done almost 20 times more rewrites before we reach the same answer. We can also see from the Lcnt (which was 1026 in the fib1 configuration) that we've reduced the amount of calls made before the result is arrived at from 949 to 41.

```
Maude> frew fib1 .
frewrite in FIB-UPDATE : fib1 .
rewrites: 232999 in 6490ms cpu (9374ms real) (35901 rewrites/second)
result Configuration:
{Class definition and message queue removed}
< 'Fibonacci1 : Ob | Cl: 'Fibonacci,Pr: none,PrQ: none,
                    Att: ('this : 'Fibonacci1),
                         ('val @ 'Fibonacci) : int(610), Lcnt: 1975 >
```

4.13 Remaining State in the Creol Interpreter

Since Creol is supposed to be used in distributed systems, we have to face the reality that messages can be in transit while the update is taking place. This kind of state has not been taken into account so far, as we have assumed that all method calls have been bound and made into processes at the time of the update. We will now deal with these other pieces of state which exist outside the objects themselves.

4.13.1 Method Invocation Messages

Method invocation messages (which we will just call invocs from now on) and method completion messages (which we will call comps) comprise all the remaining state in the Creol execution system. These messages can exist either floating freely in the configuration (this corresponds to being in transit on the network) or in a message queue for a specific object. To simplify the implementation a bit, we dump all invoc messages in the message queue of an object back into the outdated configuration when we move the object and its queue out of the outdated configuration. This way we don't have to deal with two separate cases. This simplification is not meant to be a guide for how this should be implemented in a real execution system, as it's probably impractical to retransmit all messages over a real network.

An important aspect of the Creol system is that all method calls are virtually bound and all external method calls happen through interfaces. Since we are allowed to change the inheritance hierarchy as part of an update, we have to make sure we know which method the invoc was supposed to bind to in the *old* version before we can convert it. As the object has already moved out of the outdated configuration, we use the ghost of the object to find out which class the object belongs to. Note that an object can never change class by applying an update. When we know which class the object belongs to, we can bind the method in the outdated configuration by using a special `bindInvoc` message. Invoc messages are converted when there is a ghost of the object in the configuration instead of the object itself, and this ghost indicates that it represents an old version of the object. Using ghosts enables us to successfully convert an invoc through several update membranes before the actual object is reached.

```
rl [bind-outdated-unqualified-invoc-msg] :
  < O : Ghost | Cl: C, Old: true >
  invoc(O, Q, DL)
=>
  < O : Ghost | Cl: C, Old: true >
  bindInvoc(invoc(O, Q, DL), C[nil]) .

rl [bind-outdated-qualified-invoc-msg] :
  < O : Ghost | Cl: C, Old: true >
  invoc(O, M @ C', DL)
=>
  < O : Ghost | Cl: C, Old: true >
  bindInvoc(invoc(O, M, DL), C'[nil]) .
```

The bindInvoc message is then processed by a rule mirroring closely the bindMtd rule of the interpreter. However, binding an invoc does not result in a process like binding a method does. Instead it results in a boundInvoc message which consists of the fully qualified method name (with version number of the class) and a Subst containing the bound variables. This Subst is essential when we want to refer to the (named) old parameters in the parameter conversion of the method we want to call. The boundInvoc message thus serves to complete the virtual binding and bind the parameters to their names. After we have done this we are ready to actually convert the invoc and migrate it into the updated configuration.

```
op bindInvoc  : Msg InhList -> Msg [ctor] .
op boundInvoc : Oid Cid Process -> Msg [ctor] .
ceq bindInvoc(invoc(O, M, DL), ((C)[DL']) S')
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  = boundInvoc(O, C # VN, newProcess(M @ C, get(M, MMTD), DL))
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 if M in MMTD .

ceq bindInvoc(invoc(O, M, DL), ((C)[DL']) S')
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  = bindInvoc(invoc(O, M, DL), S' S)
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 if not (M in MMTD) .
```

We then have a rule to move bound invocs across the version membrane. If the version of the class the method was found in is the same in both configurations, the invoc can be moved across with no conversion. However, if the class has been updated, we check if the method still exists - if not we discard the invoc. The normal case is that it exists, in which case we apply the conversion found in the method. We use the same helper function conv as in the paramconv function described in Section 4.11 to get back a converted Subst. However, the resulting invoc should have a List of parameter values, not a Subst of bound variables. Thus we also need a helper function toList which strips the variable names from the Subst to again get a List. Note that this list is actually longer than the original parameter list as it also includes initialisers for the local variables of the method, but this is harmless with the current parameter passing semantics. If checks are added to see that methods are called with the correct number of parameters internally in the interpreter, this will have to be changed. As it is, the implementation is simplified a lot by just passing the extra parameters. The `toList` function can be found in Appendix C.

```
rl [outdated-invoc-msg] :
  < C # VN' : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  outdated(boundInvoc(O, C # VN, process(M @ C, B, P, L)) CONFIG)
=>
  if VN == VN' then
    invoc(O, M @ C, toList(L))
  else
    if (M in MMTD) then
      invoc(O, M @ C, toList(conv(L, get(M, MMTD), ('this : O))))
    else
      none
    fi
  fi
  < C # VN' : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  outdated(CONFIG) .
```

4.13.2 Method Completion Messages

The method completion messages represent the last bit of state we have to deal with. Because we can have waiting methods in the outdated configuration preventing us from converting an object (we will get back to this in the next chapter), we have to transport completion messages both ways through the membrane. Again we use the ghost objects to guide us when we decide if a completion message needs to be transported. Ghosts which represent old versions of objects mean the completion messages need to be transported to a newer version and ghosts which represent objects that have not yet been converted indicate that we need to transport completion messages to an older version. Note that we do not convert method completion messages, this prevents us from changing the number of out parameters and also prevents us from changing the return types (except in a covariant way). The reason for not converting these is that since they need to move both ways through the membrane the added complexity is bigger than what it is for converting invocs.

```
rl [outdated-reply-msg] :
  outdated(< O : Ghost | Cl: C, Old: true > comp(O int(N) DL) CONFIG)
=>
  comp(O int(N) DL)
  outdated(< O : Ghost | Cl: C, Old: true > CONFIG).
```

```
rl [reply-outdated-msg] :
  comp(O int(N) DL)
  < O : Ghost | Cl: C, Old: false >
  outdated(CONFIG)
=>
  < O : Ghost | Cl: C, Old: false >
  outdated(comp(O int(N) DL) CONFIG) .
```

4.14 Finishing the Update Process

When all objects and messages have moved out of the outdated configuration, we are finished. There can be no more activity in a configuration consisting of only classes and ghosts, so we can at this point get rid of the old configuration. This kind of garbage collection of classes that are no longer in use would also apply to a real implementation of the update system. In our specification it is taken care of by the very simple rule:

```
ceq outdated(CONFIG) = none if objects(CONFIG) = none .
```

Chapter 5

Controlling Updates

If we continue executing old code after an update has taken place, we risk errors caused for example by accessing a class attribute which has changed type or has been removed. Even if we don't allow arbitrary modifications, we will in many cases have a hard time proving the invariants hold when we mix execution of old and new code. To get back some control without having to simply discard all inconvertible state, we will in this chapter see how the programmer can control when an update is applied and what happens to the inconvertible state.

5.1 Transaction Based Updates

One of the first systems considered for controlling updates in Creol was based on defining transactions at the language level to have an idea of when an object could be considered quiescent. Transactions were defined using predicates over the communication history of the object, so an update would not take place while a transaction was in progress. In that way we could avoid loading methods that started new transactions and just allow methods to execute that were contributing to completing transactions that were already in progress. However, such a system would not be a good fit with Creol since it's based on a purely reactive object model and although you can define 'transactions' between the object and itself in order to reach a quiescent state, it will always be a bad fit. Another reason why this idea was discarded was that it was unnecessarily strict. In most cases the state transfer will enable the converted object to complete transactions started with the old object so there is really no need to wait until all transactions have completed.

5.2 Guided Execution

The process described in the previous section is in fact just a special case of a more general technique which we will call *guided execution*. By this we mean that in order to reach a state we can accept as quiescent, we need to selectively execute some processes which contribute to moving toward this goal while preventing new processes to be started which do not contribute toward this goal. If we formulate quiescence in terms of a predicate over the communication history of the object and the state (like the assumption and invariant) we have one piece of the puzzle, but it still does not give us a procedure for selecting which processes are allowed to run in order to satisfy this predicate. However, we can formulate predicates for each of the methods in the class which decides if processes instantiated from this method are allowed to run[1]. This is a very general approach which allows almost complete control of the update procedure, but it inevitably has the overhead of maintaining the communication history and checking the predicates. For our work this was felt to be too expensive and also complicated to implement, so we went with the simpler approach.

5.3 Finishing Code which Creates Objects

The first implementation of the Creol update system suffered from a big problem when it came to finishing code which creates objects. Since we allow parameters to object creation to be added as long as all code that creates objects is also updated, we need a clear strategy for finishing code which creates objects. The chosen strategy is to create the old version of the object and then immediately subject it to conversion. However, since the first version of the update system mixed classes from the old and new versions in the same configuration, it was no longer possible to correctly find the inheritance graph in the old version (if the class the object was instantiated from had an updated superclass, that is). It was also quite problematic to separate between objects created by old code which was finishing and by new code, since the object creation process happens in several stages.

It was this problem which triggered the re-implementation where the old configuration is wrapped inside an `outdated` term which acts as a membrane that objects can migrate through into the new version. With this approach the problem completely vanishes as we are finishing the code inside the old configuration where everything works as expected and the object created can be converted just like any other object. Note also that the `update` function only wraps the old configuration in this term, it does not in any way process the objects to be updated so there is nothing which can be missed by objects created after the `update` function has finished.

5.4 Controlling Updates in the Creol System

The current system lets the programmer divide the processes into three different categories based on which method a process is an instance of. The categories are called *continue*, *restart* and *finish*. We will now describe the semantics of converting processes in each of these categories. Note that processes which are newly bound have already been converted, so the text in this section only applies to processes which are otherwise inconvertible.

First of all, processes which are instances of methods in the *continue* category are allowed to continue running with no conversion in the updated object. This approach is basically the same as

[1]We have to be able to project the communication history on a specific object X which is performing the method call in question, so some special semantics must probably be introduced for these predicates...

that which is adopted in the work on formalising updates in Creol by Johnsen, Owe and Simplot-Ryl [17]. It is possible to continue running old code when the update does not make arbitrary modifications (i.e., no attributes or methods are removed or have their type changed[2]). We will examine this requirement in more detail in the next chapter. This is the default category, so any process which is not marked as belonging to the other two categories fall under this one.

Second, processes which are instances of methods in the *restart* category will be re-instantiated in the updated object. That means a new process is created which has the same parameters as the old one, but the (updated) program is loaded from the method and execution starts from the beginning again. Because the parameters of a Creol method are read-only variables this is easy to implement, but it should be noted that it would be possible for the compiler to automatically create separate local variables for parameters that are written to. We indicate which methods we want to be restarted by a list of method identifiers prepended with the keyword **restart**.

Third, processes which are instances of methods in the *finish* category will cause state conversion to be delayed for the entire object until they are finished executing. This approach is similar to the **when-idle** clause employed in the DYMOS system [19], but operates on an object per object basis instead of delaying the update to the system as a whole. We indicate which methods belong to this category by a list of method identifiers prepended with the keyword **finish**.

A final option which could be useful is to stop all processing of a specific method. This is rather trivial to implement, but the utility of such a construct is questionable unless the language requires error handling to be present for all method calls. We do not include a *discard* category as such, but because it's impossible to restart a process which is an instance of a method that does not exist in the new version we discard the process in that case.

5.5 Implementation of Restarting Processes

Restarting processes is simply done by augmenting the `paramconv` function we saw in Section 4.11 with the ability to convert methods that are not newly bound by restarting them. We will pick the list of methods to restart out of the convert routine which is part of the method set from the class. The function `restartlist` which does this can be found in Appendix C. The important changes to the `paramconv` equations are in the conditions. The first case which converts parameters and loads the updated code into the process is used even if the process is not newly bound when the method is in the restart list. The second case which discards a process if it's not an instance of a method in the class we are currently processing is also used if the method is in the restart list but does not exist in the updated class and thus cannot be restarted. Finally, the third case for methods which are not newly bound and should be just allowed to finish has also had its condition changed so it does not apply to methods in the restart list as these now fall under the first or second case.

```
op paramconv : MProg Cid MMtd Subst -> MProg .
eq paramconv(none, C, MMTD, A) = none .
```

[2]We allow changes to the type if they are *substitutable* as will be defined in Chapter6.

```
ceq paramconv(process(M @ C, B, P, L) : W, C, MMTD, A)
  = process(M @ C, true, program(get(M, MMTD)), conv(L, get(M, MMTD), A)) :
    paramconv(W, C, MMTD, A)
 if (M in MMTD) and ((B == true) or ((M @ C) in restartlist(MMTD))) .

ceq paramconv(process(M @ C, B, P, L) : W, C', MMTD, A)
  = paramconv(W, C', MMTD, A)
 if (not (C == C')) or (not (M in MMTD) and ((M @ C) in restartlist(MMTD))) .

ceq paramconv(process(M @ C, B, P, L) : W, C, MMTD, A)
  = process(M @ C, B, P, L) : paramconv(W, C, MMTD, A)
 if (B == false and not ((M @ C) in restartlist(MMTD))) or
    (not (M in MMTD) and not ((M @ C) in restartlist(MMTD))) .
```

5.6 Implementation of Finishing Processes

First of all we make a function `waiting` which tells us if any processes which are instances of the methods in a list and are not newly bound are present in the given process queue. We then change the `convert-object` rule from Section 4.6 into a conditional rule which has `if not waiting(AL, W)` as its condition. This way the programmer can prevent conversion from happening if we cannot guarantee that it's safe to continue executing old code in the new configuration.

```
op waiting : AidList MProg -> Bool .
eq waiting(nil, W) = false .
eq waiting(X, none) = false .
eq waiting(X, process(Y, B, P, L) : W) = (X == Y and not B) or waiting(X, W) .
eq waiting(X Y AL, W) = waiting(X, W) or waiting(Y AL, W) .
```

Although this makes sure we do not convert an object which has unfinished processes, we have not specified how these unfinished processes can be completed. Recall that the interpreter rules that activate waiting processes were modified to only work on up-to-date objects. For each of these rules we need to make a corresponding rule which specifically allows activating a process which we are waiting to complete.

```
crl [guided-PrQ-enabled] :
  < outdated(O) : Ob | Cl: C, Pr: none,
                       PrQ: process(M, false, P, L) : W, Att: A, Lcnt: N >
  < outdated(O) : Qu | Ev: MM, Keep: H >
  findInheritance(outdated(O), nil, S, AL)
=>
  < outdated(O) : Ob | Cl: C, Pr: process(M, false, P, L),
                              PrQ: W, Att: A, Lcnt: N >
  < outdated(O) : Qu | Ev: MM, Keep: H >
  findInheritance(outdated(O), nil, S, AL)
if enabled(P, (A , L), MM) and waiting(AL, process(M, false, P, L)) .

crl [guided-continue] :
  < outdated(O) : Ob | Cl: C, Pr: process(M, false, continue(N), L),
                       PrQ: process(M', false, ((N ?(J)); P), L') : W,
                       Att: A, Lcnt: N' >
  findInheritance(outdated(O), nil, S, AL)
=>
  < outdated(O) : Ob | Cl: C, Pr: process(M', false, ((N ?(J)); P), L'),
                       PrQ: W,
                       Att: A, Lcnt: N' >
  findInheritance(outdated(O), nil, S, AL)
if waiting(AL, process(M', false, empty, no)) .
```

This allows us to guide the execution in the old configuration toward a state where all waiting processes have completed and we can perform the state conversion. Of course we can make the update process deadlock in various ways when we wait for things to complete, but we will look more into that in the next chapter.

5.7 An Example of Updating

In this section we will show how the Dining Philosophers example presented in Chapter 2 can be updated dynamically to use inheritance. The first update (Figure 5.1) splits the Philosopher class into two classes, the Person class and a derived Philosopher class and transfers some of the state previously belonging to the Philosopher class into the new Person class. The second update (Figure 5.2) changes the Butler class to inherit from Philosopher, and reconfigures the system so the butler object joins the philosophers at the table. Although this is obviously a toy problem, it highlights several features of our update system.

5.7.1 An Example of Dynamic Refactoring

A common critique of object oriented systems is that it is difficult to change the class hierarchy after the first version of the program. A lot of effort has been put into research on *refactoring* existing systems [24], and this example shows that our simple update system may be used to use for this purpose. However, with the current system, there is no link between the `think` and `digest` methods in the Person class and in the Philosopher class. These processes are stopped instead of restarted because there is no corresponding method in the updated Philosopher class, that is why the convert routine explicitly starts these processes again. A more general way of specifying parameter conversion where a method call could be converted to a call on another method would make it much easier to perform this kind of update.

```
class Person
begin
  var
    hungry: bool := false, history: string := ""

  {The methods digest and think are copied from the original Philosopher class}
end

class Philosopher(butler: Butler)
  implements Phil
  inherits Person
begin
  var
    chopstick: bool := true, neighbour: Phil

  convert ==
    hungry@Person := hungry@old; history@Person := history@old+"u";
    butler := butler@old; chopstick := chopstick@old;
    neighbour := neighbour@old;
    !digest ; !think
  finish
    eat, borrowStick, returnStick
  restart
   digest, think

  {The methods digest and think are removed from this class}
  {The methods eat, run, borrowStick and returnStick remain unchanged}
```

Figure 5.1: An example of a class update with state conversion

5.7.2 An Example of Dynamic Reconfiguration

Many update systems deal mainly with changes to the configuration, that means adding and deleting objects and changing the connections between objects. In the Creol update system we don't have this kind of mechanism, but because we can add code to the objects in the system we can introduce methods which allow reconfiguration to take place. These methods can be called from the convert routines of the classes to reconfigure the state of the system when it's updated. We will now develop an example that changes the Butler object to be derived from Philosopher and reconfigures the system so the Butler is also dining with the rest of the Philosophers [3]. Changing the Butler class to inherit from Philosopher along the lines of the first update is trivial, but the reconfiguration requires a little trick. We make an update to the Philosopher class that doesn't actually change the class but just runs a convert routine. The convert routine of the Philosopher class (see Figure 5.2) looks up the neighbour again, thus creating the new configuration. We need to be careful here because there is a subtle problem that can occur when calling a method from the convert routine. An external method call might end up calling a method on the object being converted, causing the update process to deadlock since the object has not been converted yet and cannot complete the method call. But in this particular case we don't have a problem. Because the butler was not a philosopher in the old version of the code, the Philosopher class' convert routine is never called for the butler. If we were to try a similar update after the butler has become a philosopher it would deadlock because of the external call to `getNeighbour`. We could fix this problem by adding a test to check if the butler object was the same as `this` in the convert routine of the Philosopher class, but it's obvious that we have to be careful when making external calls in convert routines.

5.7.3 Multi Level Updates

These two successive updates to the Dining Philosophers example can be used to illustrate another interesting feature of our update system. Because of our membrane approach which enables us to do lazy state conversion, we can in fact apply the second update while the first update is still being processed. In the Maude system we can test this by entering the command `frew [1000] update(update(v1, v2), v3) .` after loading the Dining Philosophers update example. From the resulting configuration we see that both updates have been successfully applied and all the philosophers as well as the butler are dining happily. If we turn on tracing by issuing the `set trace on.` command to Maude, we can clearly see that messages are being sent across the two version membranes as the objects migrate into the final configuration. This kind of lazy state conversion is very attractive in a real world system, as it amortises the cost of the update over a period of time and the system can remain responsive even if there is a large number of objects to be converted when an update is applied.

[3] If we wanted to get rid of one of the philosophers, we could do that in the convert routine for the butler, but the details of how objects are deleted or garbage collected would have to be considered. Since this is not the main subject here, we opt for the simpler example.

```
class Philosopher(butler: Butler)
  implements Phil
  inherits Person
begin
  var
    chopstick: bool := true, neighbour: Phil

  convert ==
    history := history+"u";
    butler := butler@old; chopstick := chopstick@old;
    butler.getNeighbour(;neighbour);
  finish
    eat
{The rest of the class is unchanged}
end

class Butler
  implements Butler
  inherits Philosopher(this)
begin
  var p1:Phil, p2:Phil, p3:Phil, p4:Phil, p5:Phil
  convert ==
    p1 := p1@old; p2 := p2@old; p3 := p3@old; p4 := p4@old; p5 := p5@old;
    hungry@Person := false; history@Person := "";
    butler@Philosopher := this; chopstick@Philosopher := true;
    !run@Philosopher

{The run method is unchanged}
  with Phil
    op getNeighbour(out n:Phil) ==
      if caller = p1 then n:=p2
      else if caller = p2 then n:=p3
      else if caller = p3 then n:=p4
      else if caller = p4 then n:=p5
      else if caller = p5 then n:=this
      else if caller = this then n:=p1
      else n:=null fi fi fi fi fi fi
end
```

Figure 5.2: Using convert routines for reconfiguration

Chapter 6

Correctness of Updates in Creol

Reasoning control in the CREOL framework is based on interface requirements formulated as assumptions and invariants. The assumption associated with an interface is a predicate formulating the obligations placed on the user of that interface, while the invariant represents the promises made by the implementor of the interface. The assumptions and invariants in the system are given as predicates on the communication history of the object. This way of reasoning about object oriented distributed systems comes from the OUN project [25]. The predicates are not tied to the implementation and its state variables at all. The invariant which has to hold for a given class is the conjunction of the invariants of all implemented interfaces in the inheritance hierarchy of that class. In addition to this it's required that the assumptions of all interfaces used with the class are satisfied. Within this framework it's also possible to give some thought to what the correctness of upgrades means. In this chapter we discuss *necessary* conditions for correct updates in our system. Although these conditions seem to suffice in most practical cases, it would be outside the scope of this thesis to identify a set of conditions that would be *sufficient* for ensuring correctness in all cases.

6.1 Substitutability

When we update a class in the system and convert the objects, the first requirement for a correct update is that we can *substitute* the converted objects for the old objects without affecting the parts of the system that were not updated. Substitutability was originally introduced as a notion related to subtyping [21], but one could argue that it is even more relevant as a principle for reasoning about the correctness of updates. If an object still behaves as expected with respect to one of it's interfaces after an update, we say that the old object is *substitutable* with the updated object with respect to that interface. What we mean by 'behaving as expected' in this context is that no calls fail, that the invariant still holds and that no additional assumptions are needed. If these conditions hold, then the system will keep running correctly (i.e., all invariants are respected) after the old object is replaced by the new.

6.1.1 A Formal Definition of Substitutability

We first define the subtype relation $\sqsubseteq$ which is a reflexive partial ordering on the set of types T. The four basic types "int", "bool", "label" and "string" are part of the set and are only related to themselves. The set of types also contains n-tuples of types, and we extend the $\sqsubseteq$ relation by saying that $A \sqsubseteq B$ if the tuples A and B have the same length l and for each $0 \leq i \leq l$ we have $A_i \sqsubseteq B_i$.

We define a similar relation on the set M of method signatures which we will write in the same way using the $\sqsubseteq$ symbol. Method signatures are 3-tuples, (N, I, O) where N is a unique identifier taken from a set of method names, I is a tuple (possibly zero-length) of in parameters and likewise O is a tuple of out parameters. Note that we are not including methods in the set of types as we cannot pass methods around or have variables of method type in the Creol language. If F and G are methods, we have

$$F \sqsubseteq G \iff F_N = G_N \wedge G_I \sqsubseteq F_I \wedge F_O \sqsubseteq G_O \tag{6.1}$$

The set T of types also contains interfaces. An interface is a set of method signatures and two predicates, the assumption and the invariant. For interfaces A and B we have $A \sqsubseteq B$ iff the assumption of A implies the assumption of B and the invariant of B implies the invariant of A[1] . We also require that each of the methods in A has a compatible operation in B where compatibility is defined in the usual way with covariant out parameter types and contravariant in parameter types. Informally we say that the interface A is *substitutable* with interface B when any object implementing A can be replaced by an object implementing B without compromising the correct operation of the system[2].

$$A \sqsubseteq B \iff asm(A) \rightarrow asm(B) \wedge inv(B) \rightarrow inv(A) \wedge \forall F.F \in B \rightarrow \exists F'.F' \in A \wedge F' \sqsubseteq F \tag{6.2}$$

6.2 Handling Arbitrary Changes to Classes and Interfaces

Most update systems only allow additions to be made and code to be replaced, but attributes cannot be removed, method signatures cannot change, attribute types cannot change, etc. This is because the state conversion process can be greatly complicated when these kinds of changes have to be taken into account. In our system we have decided to allow such updates, partly because Creol's strict call-through-interfaces semantics and virtual method binding makes it more tractable and partly because we feel the resulting system is a lot more powerful if these kinds of updates are allowed. These updates introduce *dependencies*, which makes the update of other classes necessary. We will make a distinction between *extending* an interface, a class or a method and *replacing* it. Extending something means that we change it in such a way that no dependencies are created, for example adding an attribute to a class extends the class. Replacing or removing something means we can change it in an arbitrary way, but this creates dependencies in the system.

[1]This is the same as saying that the interfaces must have covariant assumptions and contravariant invariants if we consider the implication between predicates as a kind of sub-relation.

[2]In the dynamic case we have to be more careful, if A is substitutable with B we mean that an object implementing A can be replaced at any time by an object implementing B having the same communication history projected on interface A as the original object without compromising correct operation of the system.

6.2.1 Changes to Classes

There are two kinds of dependencies on a class; subclasses are dependent on superclasses, and a class which creates instances of some specific class C is dependent on C implementing the interface that is the type of the object reference used to access the newly created object. We have to update the classes which create instances of a class C if the interfaces implemented by C are replaced. As mentioned previously, a class can be extended by adding attributes and methods, or by changing the types of attributes and methods to subtypes of their original types. In addition to these basic modifications, we can extend a class by implementing additional interfaces or inheriting from additional superclasses.

Replacing or removing attributes and internal methods (which are not exposed through interfaces) is manageable, because we provide local quiescence (i.e., running code in the object is finished when the conversion routine is called). We require methods that have had their signature changed to have a parameter convert routine. Because we do not provide a mechanism for converting output parameters back to the old version, the only permitted modification of the output parameters is to change the type of an output parameter to a subtype. If any methods depend on the change introduced by replacing an attribute or a method, these methods have to be included in either the finish or the restart category. Subclasses are naturally dependent on the attributes and methods of the superclass, so replacing or removing these mean that any subclasses have to be type checked again and possibly also updated.

6.2.2 Changes to Interfaces

If the list of interfaces a class implements has been replaced, we can have code finishing in an object O which invokes a method that only existed in the previous version of the class. This case can clearly not be resolved with parameter conversion, and even if the update will convert O so it no longer calls the problematic method, we cannot be sure that O will be converted before the object that it is calling the method on. In this case there is no other solution than to require global quiescence. This means that all objects have to have reached a processor release point and finished all methods in their finishlists[3] before any object can be converted. Ideally the update system will also be able to carry out these more general updates, but for distributed systems it can be very costly to ensure global quiescence, since this requires communication to take place between all nodes in the system. We have not tried to implement this in our current system, so we have to require that an update does not remove an exposed method from a class unless it can be proven to be dead code.

In our system we allow an interface to be extended as part of an update. This naturally creates a dependency in any class that implements the extended interface or any of its subinterfaces, as implementations of the new methods need to be provided. Adding superinterfaces to an interface is also a kind of extension, with the same dependencies being introduced as when we extend the interface with new methods. Making the interface invariant stronger or the assumption weaker is also considered an extension of the interface. In addition, we allow replacing method signatures

[3]We also require that there are no messages in transit, depending on the runtime system we may already be sure that there are no messages in transit if the other requirements are satisfied.

in interfaces, creating a dependency on all implementors and all users of the changed interface. Replacing method signatures is possible precisely because we have parameter conversion. If calls relying on the old method signature can be parameter converted, old code can finish using the updated method while the parameters are converted by the update system.

6.3 Correctness of State Conversion

It is not sufficient for an update to satisfy the substitutability requirement. We must also consider liveness and safety properties of the state conversion. First of all we need to prove that given the previous class invariant, after executing the class conversion routine the invariant of the updated class is satisfied. Second we need to prove that the conversion routine will not deadlock through calling methods which depend on the object already having been converted. In addition to this, we need to ensure that converting processes will lead to a correct result. Since we have different update semantics for processes in the different categories, we will now consider necessary correctness conditions for each of these in turn.

6.3.1 Correctness of Continuing Processes

By default, processes which don't belong to either the restart or the finish category belong to the continue category. Processes in this category continue to execute as-is after an update has taken place. However, this approach can sometimes lead to runtime errors. To prevent these errors, we need to type check the original method definition of a continuing process in the context of the updated class hierarchy. If the method definition type checks in the context of the updated class hierarchy, no runtime error can be introduced by continuing to execute a process that is an instance of this method definition. For a deeper analysis we need to prove that assuming the old invariant held at all processor release points in the original method definition, the updated class invariant also holds at all these points. If a method does not satisfy these conditions, processes which are instances of this method must belong to either the finish or the restart category.

6.3.2 Correctness of Restarting Processes

Processes that belong to the restart category load their code from the updated class definition, so type safety is not a problem. The problem is that stopping a process before it would normally finish might break the updated invariant. Of course we only stop processes at processor release points, so the old invariant can be assumed. However, the communication history might get disrupted. An example of this difficulty is the `eat` method in the Dining Philosophers example, a natural invariant for the Philosopher class is that there is at most one more call to the `borrowStick` method than to the `returnStick` method. However, if we restart the `eat` method it could happen that `borrowStick` is called twice without an intermediate call to `returnStick`. Consequently we need to prove that the updated invariant is satisfied in all possible communication histories that can arise from restarting processes at processor release points in order to include those processes in the restart category.

6.3.3 Correctness of Finishing Processes

Processes that belong to the finish category are always finished before conversion takes place, so there is no safety property to be proven. However, we need to prove the liveness of updates given a specific set of methods in the finish category. The problem related to code that must finish is that it can call other methods and get stuck waiting for the replies. Thus it's required that if a process is an instance of a method in the finish category, any method calls (that have returns) made by this process are external or to other methods which are also in the finish category. If the call is internal, it is required that this condition also holds for the called method. Tail-recursive processes are handled correctly because once they call themselves, the process becomes newly bound and thus eligible for parameter conversion. It's still possible for an object to get stuck waiting for a reply from a method call to another object if that object is finishing a process which is waiting for a method call to complete in the first object. This kind of situation is basically the same as a normal deadlock, and the usual techniques to prove that a program is deadlock free can be applied to this problem.

6.4 Security of Updates in a Distributed Environment

Finally, in a distributed environment it's not enough to statically prove that the update is correct before applying it. This would open up all kinds of avenues for attack in a hostile environment such as the Internet. One way of adding security is to use a system of trust where each node in the system will only receive cryptographically signed updates that can be verified to come from some trusted author. This places the complete responsibility for correctness on the author of the update. Another system which is possible is that the updates themselves carry proof of their correctness and the update system then verifies this proof before applying the update. This relies on the fact that verifying a proof is a computationally simple procedure while actually creating the proof is computationally hard. These two ways of providing security are of course more or less orthogonal to each other and a combination of these techniques can be used to provide the right kind of security for a particular application.

Chapter 7

Conclusion and Future Research

The main question this thesis sought to answer was "How can we make a safe and flexible mechanism for dynamically updating Creol programs?". We believe we have answered this question by carefully considering the possible approaches to each of the identified sub-problems and selecting the approach which fits best within the existing Creol framework. We will now give a brief summary of the answers to the sub-problems:

The *unit of replacement* chosen for Creol is the class because the mechanism for finding the objects which need to be updated is class-based, and formal reasoning in the Creol framework is done using classes.

State transfer has been fully integrated with object orientations in Creol by providing a hierarchy of conversion functions that work to convert any object, even allowing for dynamic changes to the class hierarchy. We are not aware of any other update systems which allow this kind of flexibility, most other systems don't address inheritance [3, 12, 19, 27].

Parameter conversion of calls made by old code is introduced to allow a lazy update strategy that minimally impacts *system responsiveness* while the update is in progress. Parameter conversion has been used previously in the DYMOS system [19], but most recent systems do not support this. Some other systems provide lazy updates, but they are either not very flexible [17] or they are very high level [4], requiring atomic transactions or other features not normally found in generic programming languages.

Apart from other work on updates in the Creol framework [17], the only work we are aware of that take into consideration the special challenges posed by concurrency is the Argus system [3] and related transaction based solutions [4]. In this thesis we have made use of the particular properties of Creol programs, which clearly define processor release points and invariants that are satisfied at these points, in order to find points in the program where updates can be applied. Together with the finish semantics corresponding to the when-idle semantics from the DYMOS system and the restart semantics which are inspired by the error-recovery approach taken in the Argus system, this provides a natural and powerful method for controlling updates in a concurrent and distributed setting.

We have made an attempt at clarifying the *conditions for a correct update* in this system in terms of substitutability, safety properties and liveness properties. This seems a natural approach for reasoning about updates in an object oriented system, but a complete investigation of correctness in the system presented remains a subject for future research. In this respect, other systems [12, 27] are more developed, but these systems do not consider inheritance or concurrency and are also less flexible than our approach. As our system has a formal specification in rewriting logic, it is possible to reason about the correctness of updates [1] and we expect that sufficient conditions for correctness of updates with our approach can be established in the future.

In conclusion, we believe the Creol update system presented in this thesis provides a good balance between flexibility and safety, takes inheritance and concurrency into consideration and can perform updates with minimal impact to the responsiveness of the running system.

Future Research

It would be interesting to use the specification given in this thesis to make a low level implementation of the Creol update system where performance results could be investigated. A real implementation of the update system would allow much more complicated systems to be built with Creol and thus create more realistic scenarios for updates. An implementation of the update system could for example be made with the LLVM framework for dynamic compilation [18] and a small runtime using MPI [11] calls for message passing between nodes. I am sure such a system would not only be a valuable tool for programming distributed applications, but would most certainly also highlight areas for improvement in the high level specification of Creol.

It would also be of great importance to make a completely formal investigation of the correctness criteria for updates in my proposed system. Recent work by Stoyle, Bierman, Hicks et al. has introduced a very impressive formal system for updates in C like languages called Proteus [27]. However, this work does not take into account the special properties of object oriented and distributed systems. Judging from this work, it would be a huge endeavour to completely define correctness in my system so in this thesis I have only attempted to provide some pointers to which problems have to be solved.

Several extensions to the update system presented in this thesis are possible. I think the most interesting one would be a more complete investigation of the concept of guided execution. General techniques for reaching a quiescent state would be applicable to almost any update system, and thus potentially valuable. One such method I have only briefly investigated is to formulate regular expressions on the communication history of objects. Since regular expressions can be evaluated by a finite state machine, it's not necessary to keep a record of the communication history to evaluate when we reach a quiescent state. Each state in our state machine can then contain information about which methods will contribute toward reaching a quiescent state and should be allowed to execute when we are seeking quiescence. This is much more powerful that the simple finish or restart semantics used in the system presented here, but it remains to be seen if this extra power is worth the performance overhead added to the method call mechanism.

[1]Other systems may have inherent problems such as representation incoherency [9] or they are based on a programming language that doesn't allow formal reasoning [13, 22]

Other possible extensions include a more general way of specifying parameter conversion where a method call could be converted to a call on another method. I expect this to be a simple extension to make and it would make it much easier to refactor existing class hierarchies, so it seems like a valuable addition to the system. We could also add the capability to do state transfer for completion messages which would allow arbitrary changes to the return types of methods. One way of doing it without requiring extra syntax would be to allow assignments to the out parameters in the parameter conversion. The semantics of this would be to use the assignments to unqualified parameters when converting one way and assignments to parameters qualified with **old** when converting the other way. However the value of such an extension seems questionable based on the findings of Stoyle, Bierman, Hicks et al. [27], where a number of open source applications were analysed to see which kinds of changes were common. They found that very few changes were made to function signatures, and even then the changes consisted primarily of adding new parameters.

Finally an investigation of security concerns would be an interesting undertaking. If we want to create a system which can match the functionality of Windows Update or other such systems for distributing patches, security has to be considered carefully. The goal should be to have updates as an integrated service in the operating system so you would never have to download or install patches to software you are using and would not notice an update taking place except as a pleasant surprise due to new or improved functionality.

Appendix A

Creol EBNF Syntax

```
letter = ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ |
         ’m’ | ’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’ |
         ’y’ | ’z’ | ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’ | ’I’ | ’J’ |
         ’K’ | ’L’ | ’M’ | ’N’ | ’O’ | ’P’ | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ |
         ’W’ | ’X’ | ’Y’ | ’Z’ ;

digit =  ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’ ;

symbol = ’!’ | ’#’ | ’$’ | ’%’ | ’&’ | "’" | ’(’ | ’)’ | ’*’ | ’+’ | ’,’ | ’-’ |
         ’.’ | ’/’ | ’:’ | ’;’ | ’<’ | ’=’ | ’>’ | ’?’ | ’@’ | ’[’ | ’\’ | ’]’ |
         ’^’ | ’_’ | ’‘’ | ’|’ | ’~’ ;

comment delimiter = ’{’ | | ’}’ ;

space = ’ ’ ;

identifier = (letter | ’_’), { letter | number | ’_’ } ;

number = digit, { digit } ;

string = ’"’, { letter | digit | symbol | space | comment delimiter }, ’"’ ;

name = identifier [ "@", identifier ] ;

expression = name | number | string | "true" | "false"
           | "-", expression
           | expression, "+", expression
           | expression, "-", expression
           | expression, "*", expression
           | expression, "/", expression
           | "not", expression
           | expression, "or", expression
           | expression, "and", expression
           | expression, "=", expression
           | expression, "/=", expression
           | expression, ">", expression
           | expression, "<", expression
           | expression, "<=", expression
           | expression, ">=", expression ;

parameters = "(", expression, { ",", expression }, ")" ;

inout parameters = parameters
                 | "(", [ expression, { ",", expression } ],
                   ";", name, { ",", "name" }, ")" ;

method call = name, [ inout parameters ]
            | [ identifier ], "!", name, [ parameters ]
            | name '.' identifier, [ inout parameters ]
            | [ identifier ], "!", name '.' identifier, [ parameters ] ;

method return = identifier, "?", "(", name, { ",", name }, ")";

guard = "wait" | expression | method return ;

statement = name, ":=", expression
          | name, ":=", "new", identifier, parameters
          | "if", expression, "then", statements, [ "else", statments ], "fi"
          | "while", expression, "do", statements, "od"
          | "await", guard, { "&", "guard" }
          | method call
          | method return

statements = statement
           | statements, "[]", statements
           | statements, "||", statements
           | statements, ";",  statements ;

declaration = identifier, ':', identifier ;
definition  = identifier, ':', identifier, [ ":=", expression ] ;

declarations = declaration, { ',', declaration } ;
definitions  = definition,  { ',', definition } ;

inheritance  = identifier, [ parameters ] ;

finish list  = "finish",  identifier, { ',', identifier } ;
restart list = "restart", identifier, { ',', identifier } ;
```

```
parameter convert = "convert", definitions ;

class convert = "convert", "==", [ "var", definitions ], statements,
                [ finish list ], [ restart list ] ;

method signature = "op", identifier,
                   [ ( "(", declarations, ["out", declarations], ")"
                     | "(", [declarations], "out", declarations, ")" ) ] ;

method definition = method signature, [ parameter convert ], "==",
                    [ "var", definitions ], statements ;

interface declaration = "interface", identifier,
                        [ "[", declarations, "]" ],
                        ["inherits", inheritance, { ',', inheritance } ],
                        "begin",
                        [ "types", declarations ],
                        { "with" ( identifier | "any" ), { method signature } },
                        "end" ;

class declaration = "class", identifier,
                    [ "[", declarations, "]" ], [ "(", declarations, ")" ],
                    ["implements", identifier, { ',', identifier }],
                    ["inherits", inheritance, { ',', inheritance } ],
                    "begin",
                    [ "var", definitions ],
                    [ class convert ],
                    { method definition },
                    "end" ;
```

Appendix B

Full Specification of the Interpreter

```
************************ STRUCTURE *********************************

fmod DATA is
pr QID .
pr STRING .
pr INT .

sorts    Nil Aid UTDOid Oid AidList Data DataList Call Expr List .

subsorts Qid < UTDOid < Oid .
subsorts Qid < UTDOid Oid Aid < Data < Expr DataList < List .
subsorts Aid < AidList < DataList .
subsorts Nil < AidList < DataList .
subsort  Call < Expr .

op null :           -> Oid  [ctor] . *** undef. value/none pointer
op _[_] : Qid List  -> Call [ctor] . *** call
op int  : Int       -> Data [ctor] .
op str  : String    -> Data [ctor] .
op bool : Bool      -> Data [ctor] .
op pair : Data Data -> Data [ctor] .
op pair : Expr Expr -> Expr [ctor] .
op list : DataList  -> Data [ctor] .
op list : List      -> Expr [ctor] .

op nil  :           -> Nil [ctor] .
op __   : List List -> List    [ctor assoc id: nil] .
op __   : DataList DataList -> DataList [ctor ditto] .
op __   : AidList  AidList  -> AidList  [ctor ditto] .
op __   : Nil  Nil  -> Nil  [ctor ditto] .

```

```
*** Expressions
ops not_ neg_             : Expr      -> Expr .
ops _+_ _-_ _*_ _/_ _cat_ : Expr Expr -> Expr .
ops _<_ _<=_ _>_ _>=_     : Expr Expr -> Expr .
ops _and_ _or_ _/=_ _=_   : Expr Expr -> Expr .
ops pair                  : Expr Expr -> Expr [ctor] .
ops _.fst _.scd           : Expr -> Expr .  *** first and second part of pair

***  some CMC list functions on expressions
ops head last : Expr      -> Expr . *** first and last element
op rest       : Expr      -> Expr . *** left-rest
op tail       : Expr      -> Expr . *** right-rest
op _++_       : Expr Expr -> Expr . *** concat
op _+-_       : Expr Expr -> Expr . *** append right
op _-+_       : Expr Expr -> Expr . *** append left
op length     : Expr      -> Expr . *** list length
op isempty    : Expr      -> Expr . *** is the list empty
ops has index : Expr Expr -> Expr . *** element test, indexing: list number
op  after     : Expr Expr -> Expr . *** tail of list after index
op remove     : Expr Expr -> Expr . *** remove (first occurrence of)
                                    *** element from list

op _asInt             : Expr -> Int .
op _asBool            : Expr -> Bool .
op _asStr             : Expr -> String .

var N : Nat .
var B : Bool .
var S : String .
eq int(N)  asInt      = N .  *** otherwise error...
eq bool(B) asBool     = B .
eq str(S)  asStr      = S .

endfm

*** Bound variables ***
fmod LIST-QID-VAL is
protecting DATA .

*** Subst: non-repetitive list of BndVar.
*** InitVar InitSubst has Expr where BndVar Subst has Data.
sorts    BndVar Subst InitVar InitSubst .
subsorts BndVar < Subst InitVar < InitSubst .

op _:_ : Aid Data    -> BndVar  [ctor format (! o o o) ] .
op _:_ : Aid Expr    -> InitVar [ctor format (! o o o) ] .
op no  :             -> Subst   [ctor] .
op _,_ : Subst Subst -> Subst   [ctor assoc id: no] .
op _,_ : InitSubst InitSubst -> InitSubst [ctor assoc id: no] .

*** Remove multiple values of same variable in a Susbt
var  A : Aid .
var  L : Subst .
vars D D' : Expr .
eq   (A : D), L, (A : D') = if D' == null then (A : D), L else (A : D'), L fi .

endfm

*** CREOL guards ***
fmod GUARDS is
protecting LIST-QID-VAL .

sorts     Guard Wait Return ExtGuard .
subsorts  Return Expr Wait < Guard  < ExtGuard .

op wait :   -> Wait [ctor] .           *** suspension
op _? : Aid -> Return [ctor] .         *** reply guard
op _? : Nat -> Return [ctor] .         *** low level reply guard
op _?G(_) : Aid List -> ExtGuard .     *** reply guard with side effect
op nothing : -> Guard [ctor] .
op _&_ : Guard Guard -> Guard [ctor id: nothing assoc comm prec 55] .
op _&_ : Guard ExtGuard -> ExtGuard [ctor ditto] .

*** reduction of guards to normalform: [wait &]? [bool &]? return ***
vars E  E' : Expr .
eq   wait & wait = wait .
eq   E & E'      = E and E' .
endfm

*** CREOL program code ***
fmod PROG is
protecting GUARDS .

sorts   Stm Prog ProgList NeProgList . *** Stm is basic statem
*** without a leading guard.
subsort Stm < Prog < NeProgList < ProgList .   *** possibly with Reply < ...

*** Cid is class identifier, Mid method name, Aid attribute name
sorts   Cid Mid Version .
subsort Nat < Version .
op _#_ : Qid Version -> Cid [ctor] .
subsort Qid < Aid < Cid Mid .

op _._   : Expr Qid -> Mid [ctor] . *** remote call
op _@_   : Qid  Cid -> Mid [ctor] . *** method qualified by class name (local)
op _@_   : Qid  Cid -> Aid [ctor] . *** attribute qualified by class name (local)

op empty :                         ->   ProgList [ctor] .
op _;_   : ProgList     ProgList ->   ProgList [ctor assoc id: empty] .
op _;_   : NeProgList   ProgList -> NeProgList [ctor ditto] .
op _;_   : ProgList   NeProgList -> NeProgList [ctor ditto] .
op _;_   : NeProgList NeProgList -> NeProgList [ctor ditto] .

*** CREOL program syntax
*** op _:=_  : Aid Expr -> Stm [ctor] . *** simple assignment
op _:=_ : AidList List -> Stm [ctor]. ***simultaneous assignment, same length
op _:= new_(_) : Aid Cid List -> Stm [ctor] .  *** object creation
op if_th_el_fi : Expr ProgList ProgList -> Stm [ctor] .
op if_th_fi    : Expr ProgList          -> Stm .
op while_do_od : Expr ProgList -> Stm [ctor] .
op _(_;_) : Mid List List -> Stm [ctor] .  *** sync. call (with reply)
op !_(_)  : Mid List -> Stm .            *** async. call (without label)
op _!_(_) : Qid Mid List -> Stm .       *** async. call (with label)
op _?(_)  : Qid List -> Stm [ctor] .            *** async. reply statement
op end : List -> Stm [ctor] .                  *** method return
op continue : Nat -> Stm [ctor] .              *** sync. termination
op _[]_   : ProgList ProgList -> Prog [ctor assoc]. *** non-deterministic progs
op _|||_  : ProgList ProgList -> Prog [comm assoc] . *** interleaved progs
op _MERGER_  : ProgList ProgList -> Prog [assoc] . *** EBJ 18.01.2005
op await_ : Guard    -> Prog [ctor] .
op await_ : ExtGuard -> Prog .        *** reply guards with side effect

*** Low level CMC mechanism for sync. calls (dummy labels using Nat)
op _?(_)  : Nat List -> Stm [ctor].            *** sync. reply statement

*** A Process had a Mid of the method it's an instance of,
*** a Bool indicating if it is newly bound (no execution has been done yet),
*** a ProgList with statements and a Subst of bound variables
sort Process .
op none    : -> Process [ctor] .
op process : Mid Bool ProgList Subst -> Process [ctor] .

*** Multiset of Processes
sort MProg .
subsort Process < MProg .
op _:_ : MProg MProg -> MProg [ctor assoc comm id: none] .

*** Some simplifying equivalences for Programs and Processes
var EL : List .
var M  : Mid .
var B  : Bool .
var L  : Subst .
var P : ProgList .
eq (nil := EL)    ; empty = empty .
eq (await nothing); empty = empty .
eq process(M, B, empty, L)   = (none).Process .
eq (empty ||| P) = P . *** ||| with attr. id: nil gives problems.
endfm

*** CREOL classes ***
fmod CLASS is
protecting PROG .

sorts    Class Mtd MMtd Inh InhList . *** inheritance list
subsorts Call < Inh < Expr .
subsorts Nil Inh < InhList < List .

op  _[_] : Cid  List       -> Inh .    *** initialised superclass
op  __   : InhList InhList -> InhList [ctor assoc id: nil] .

var Ih : Inh . var S : InhList .
eq  Ih S Ih = Ih S .

op <_: Mtdname | Latt:_, Code:_> : Qid Subst ProgList ->
     Mtd [ctor format (m! om m m m m m g m m m! on)] .

subsort Mtd < MMtd .    *** Multiset of methods

op none : -> MMtd [ctor] .
op _*_  : MMtd MMtd -> MMtd [ctor assoc comm id: none] .

op <_: Cl | Inh:_, Att:_, Mtds:_, Ocnt:_> :
     Cid InhList InitSubst MMtd Nat -> Class
     [ctor format (nb! b! ob b b b o g  b o g  b o g  b o  b! on )] .

endfm

*** CREOL objects ***
fmod OBJECT is
protecting CLASS .

sort Object .

op <_: Ob | Cl:_, Pr:_, PrQ:_, Att:_, Lcnt:_> :
Oid Cid Process MProg Subst Nat -> Object [ctor
    format (nr! r! ob r r  or b g  r r g  r m g  r o g  r o r! no )] .
endfm

*** CREOL messages and queues ***
fmod COMMUNICATION is
protecting DATA .
pr PROG .
```

```

sort NatS . *** list of nats

sort Msg MMsg Kid Queue .
subsort Msg < MMsg .

op none : -> MMsg [ctor] .
op __ : MMsg MMsg -> MMsg [ctor assoc comm id: none] .

*** INVOCATION and REPLY
op invoc(_,_,_) : Oid Mid DataList -> Msg  *** invocation
     [ctor format (rg o o o o o o o o no)] .
op comp(_) : DataList -> Msg               *** completion
     [ctor format (rg o o o no)] .
op error   : String -> Msg [ctor] .        *** error
op warning : String -> Msg [ctor] .        *** warning

*** message queue
op empty :         -> NatS  [ctor].
op [_] : Nat       -> NatS  [ctor].
op _;_ : NatS NatS -> NatS [ctor assoc id: empty] .

op <_: Qu | Ev:_, Keep:_> : Oid MMsg NatS -> Queue
                          [format (nm! m! om m m  m o m  m o m! no)] .

endfm

*** STATE CONFIGURATION ***

fmod CONFIG is
protecting OBJECT .
protecting COMMUNICATION .

sort Configuration .

subsorts Object MMsg Queue Class < Configuration .

op none : -> Configuration [ctor] .
op __ : Configuration Configuration -> Configuration
                         [ctor assoc comm id: none] .

endfm

******************** AUXILIARY FUNCTIONS *****************************

fmod FUNKSJONER is
pr COMMUNICATION .
pr OBJECT .
```

```

*** Queue-function ***
op inqueue  : Nat MMsg -> Bool . *** checks if  Msg is in the queue

*** Class/method functions ***
op get      : Qid MMtd -> Mtd .   *** fetches method
op _in_     : Qid MMtd -> Bool .  *** checks if Q is a declared method

*** VarList functions ***
op val      : Aid Subst -> Data .      *** fetches value of prog. var.
op _in_     : Aid Subst -> Bool .      *** checks occurence in list
op _in_     : Aid AidList -> Bool .    *** checks occurence in list
op evalList : List Subst -> DataList . *** maps list to values
op eval     : Expr Subst -> Data .     *** evaluate expression
op evalB    : Expr Subst -> Bool .
op evalI    : Expr Subst -> Int .
op evalS    : Expr Subst -> String .
op enabled  : ProgList Subst MMsg -> Bool .  *** eval guard
op ready    : ProgList Subst MMsg -> Bool .  *** eval guard
op assign   : InitSubst DataList  -> Subst . *** parameter substitution

*** variables
vars A A'     : Aid .
vars Q Q'     : Qid .
vars O O' O'' : Oid .
vars L L'     : Subst .
var  IL       : InitSubst .
vars D D'     : Data .
vars DL DL'   : DataList .
var  St St'   : String .
var  MMTD     : MMtd .
var  MTD      : Mtd .
var  G G'     : Guard .
var  SP SP'   : Prog .
vars P P'     : ProgList .
vars PR PR'   : Process .
vars neP      : NeProgList .
var  MP       : MProg .
vars I J      : List .
vars M M'     ; Mid .
var  MM       : MMsg .
vars X X'     : Expr .
vars B B'     : Bool .
vars N N'     : Nat .
vars C C'     : Int .
var  AL AL'   : AidList .

eq A in no            = false .
eq A in ((A' : D), L)  = if A == A' then true else (A in L) fi .

eq A in nil             = false .
eq A in (A' AL) = if A == A' then true else (A in AL) fi .

eq val(A, no)          = A .  *** or null
eq val(A,((A' : D),L)) = if A == A' then D else val(A, L) fi .

eq evalI(X,L)          = eval(X,L) asInt .
eq evalB(X,L)          = eval(X,L) asBool .
eq evalS(X,L)          = eval(X,L) asStr .
*** evaluates to Data ***

eq eval(null, L)       = null . *** superfluous?
eq eval(A, L)          = val(A, L) .

eq eval(pair(X,X'),L) = pair(eval(X,L),eval(X',L)) .
eq eval(list(J), L)    = list(evalList(J, L)) .
eq evalList(nil, L)    = nil .
eq evalList(X I, L)    = eval(X, L) evalList(I, L) .
eq eval(Q[I], L)       = Q[evalList(I, L)] .   *** call, need also to define Q
eq 'plus [int(C) int(C')] = int(C + C') .
eq 'plus [str(St) str(St')] = str(St + St').
eq eval(index(X,X'),L)= index(eval(X,L),eval(X',L)) .
eq index(list(D DL),int(N))= if N == 1 then D else
                             index(list(DL),int(N - 1))fi .
eq eval(after(X,X'),L)= after(eval(X,L),eval(X',L)) .
eq after(list(  DL),int(0)) = DL .
eq after(list(D DL),int(N)) = if N == 1 then DL else
                              after(list(DL),int(N - 1))fi .
eq eval(remove(X,X'),L)= remove(eval(X,L), eval(X',L)) .
eq remove(list(nil), D) = list(nil) .
eq remove(list(D DL), D') = if D == D' then remove(list(DL), D')
                            else list(D remove(list(DL), D')) fi .

*** data-string
eq eval(X cat X', L) = str(eval(X, L) asStr + eval(X', L) asStr) .

*** data-int
eq eval((neg X),  L) = int(- (eval(X, L) asInt)) .
eq eval((X + X'), L) = int(eval(X, L)asInt  +  eval(X', L)asInt) .
eq eval((X - X'), L) = int(eval(X, L)asInt  -  eval(X', L)asInt) .
eq eval((X * X'), L) = int(eval(X, L)asInt  *  eval(X', L)asInt) .
eq eval((X / X'), L) = int(eval(X, L)asInt quo eval(X', L)asInt) .

*** data-pair
eq pair(D,D').fst = D .
eq pair(D,D').scd = D' .

eq eval(X .fst,L) = eval(X,L).fst .
eq eval(X .scd,L) = eval(X,L).scd .

*** evaluates to booleans
eq eval((X >  X'), L) = bool(eval(X, L)asInt >  eval(X', L)asInt) .
eq eval((X >= X'), L) = bool(eval(X, L)asInt >= eval(X', L)asInt) .
eq eval((X <  X'), L) = bool(eval(X, L)asInt <  eval(X', L)asInt) .
eq eval((X <= X'), L) = bool(eval(X, L)asInt <= eval(X', L)asInt) .
eq eval( X =  X',  L) = bool((eval(X, L) ==  eval(X', L))) .
eq eval( X /= X',  L) = bool((eval(X, L) =/= eval(X', L))) .
eq eval(  not X,   L) = bool(not (eval(X, L) asBool)) .
eq eval(X and X',  L) = bool(eval(X, L)asBool and eval(X', L)asBool) .
eq eval(X or  X',  L) = bool(eval(X, L)asBool or  eval(X', L)asBool) .

*** data-list
eq eval(head(X), L)   = head(eval(X, L)) .
eq eval(last(X), L)   = last(eval(X, L)) .
eq eval(tail(X), L)   = tail(eval(X, L)) .
eq eval(rest(X), L)   = rest(eval(X, L)) .
eq eval(length(X),L)  = length(eval(X, L)) .
eq eval(X ++ X', L)   = eval(X, L) ++ eval(X', L) .
eq eval(X -+ X', L)   = eval(X, L) -+ eval(X', L) .
eq eval(X +- X', L)   = eval(X, L) +- eval(X', L) .
eq eval(isempty(X),L) = isempty(eval(X, L)) .
eq eval(has(X,X'), L) = has(eval(X, L), eval(X', L)) .

eq eval(D, L)         = D [owise] .

*** list-functions
eq head(list(nil))    = null .
eq head(list(D I))    = D .
eq last(list(nil))    = null .
eq last(list(I D))    = D .
eq rest(list(nil))    = null .
eq rest(list(I X))    = list(I) .  ***may use D to enforce bottom up evaluation
eq tail(list(nil))    = null .
eq tail(list(X I))    = list(I) .  ***may use D to enforce bottom up evaluation
eq list(I) ++ list(J) = list(I J) .
eq D -+ list(J)       = list(D J) .
eq list(I) +- D       = list(I D) .
eq has(list(nil), D)  = bool(false) .
eq has(list(J D I),D) = bool(true) .
eq isempty(list(nil)) = bool(true) .
eq isempty(list(D I)) = bool(false) .
eq length(list(nil))  = int(0) .
eq length(list(D I))  = if I == nil then int(1)
                          else length(list(I)) + int(1) fi .
```

```
eq int(C) + int(C')   = int(C + C').          *** needed due to previous line!

*** inspects queue
eq inqueue(N, none)                = false .
eq inqueue(N, comp(O int(N') J) MM) =
     if N == N' then true else inqueue(N, MM) fi .

*** test of guard by ENABLED (EBJ 18.01.2005)
eq enabled(SP ; neP,       L, MM) = enabled(SP, L, MM) .
eq enabled(P []  P',       L, MM) = enabled(P, L, MM) or enabled(P', L, MM) .
eq enabled(P ||| P',       L, MM) = enabled(P, L, MM) or enabled(P', L, MM) .
eq enabled(await(wait & G),L, MM) = false . *** Note: no wait in PrQ!
eq enabled(await (X & G),  L, MM) = evalB(X, L) and enabled(await G, L, MM) .
eq enabled(await((Q ?)& G),L, MM)
 = inqueue(evalI(Q, L), MM) and enabled(await G, L, MM) .
eq enabled(await((N ?)& G),L, MM)
 = inqueue(N, MM) and enabled(await G, L, MM) .
eq enabled(empty,L, MM)           = true .
eq enabled(SP ; P',        L, MM) = true [owise].

eq ready(SP ; neP, L, MM) = ready(SP, L, MM) .
eq ready(Q ?(I),   L, MM) = inqueue(evalI(Q, L), MM) .
eq ready(N ?(I),   L, MM) = inqueue(N, MM) .
eq ready(SP ; P',  L, MM) = enabled(SP ; P', L, MM) [owise].

eq get(Q, < Q : Mtdname | Latt: L, Code: P > * MMTD)
 = < Q : Mtdname | Latt: L, Code: P > .

eq Q in none = false .
eq Q in (< Q' : Mtdname | Latt: L, Code: P > * MMTD)
 = if Q == Q' then true else Q in MMTD fi .

*** makes substitutions
eq assign(IL, nil) = IL .
eq assign(no, DL)  = no . *** needed??
eq assign(((A : X), IL), D DL) = (A : D), assign(IL, DL) .

*** transform wait guards when active code is suspended. remove all waits.
op clear : ProgList -> ProgList .
op clear : Guard    -> Guard .

eq clear(empty)     = empty .
eq clear(SP ; neP)  = clear(SP) ; neP .
eq clear(P [] P')   = clear(P) [] clear(P').
eq clear(await G)   = await( clear(G)).
eq clear(nothing)   = nothing .
eq clear(wait & G)  = clear(G) .
eq clear(X & G)     = X & clear(G).
eq clear((Q ?) & G) = (Q ?) & clear(G).
eq clear((N ?) & G) = (N ?) & clear(G).
eq clear(SP)        = SP [owise].
eq clear(P ||| P')  = clear(P) ||| clear(P').

***    EXPANSION OF LANGUAGE MACROS:
eq await (G' & Q ?G(I)) = await (G' & Q ?); (Q ?(I)) .
eq if X th P fi     = (if X th P el empty fi) .

endfm

*** the CREOL INTERPRETER ***

mod INTERPRET is
pr CONFIG .
pr FUNKSJONER .
pr CONVERSION .

op new_(_) : Cid List -> Msg . *** initialise program message

vars O O'  : Oid .
var  UTD   : UTDOid .
vars B B'  : Bool .
vars C C'  : Cid .
var  Q     : Qid .
vars X Y   : Aid .
var  D D'  : Data .
vars W W'  : MProg .
vars I J K : List .
var  MM    : MMsg .
var  Ms    : Msg .
var  MMTD  : MMtd .
vars M M'  : Mid .
vars E E' OE  : Expr . *** OE is oid-expr.
var  F     : Nat .     *** Ocnt:-values
var  VN    : Nat .     *** Version number
vars N N'  : Nat .
vars G G'  : Guard .
var  AL    : AidList .
vars DL    : DataList .
vars S S'  : InhList .    *** list of (parameterized) superclasses
vars IA    : InitSubst .  *** list of initialized attribute declarations
vars L L' A A' : Subst .  *** local var and attribute var-lists, respec.
vars P P' P'' R R' : ProgList .
vars PR PR' : Process .
vars SP    : Prog .
var  St    : Stm .
vars H H'  : NatS .
```

```

op newProcess : Mid Mtd DataList -> Process .
eq newProcess(M, < Q : Mtdname | Latt: L, Code: P > , D D' DL) =
     process(M, true, P, (('caller : D), ('label : D'), assign(L, DL))) .

**********VIRTUAL BINDING of METHODS With MULTIPLE INHERITANCE

op bindMtd : Oid Qid List InhList -> Msg [ctor] . ***Bind method request
*** Given: callee method params (list of classes to look in)
op boundMtd : Oid Process         -> Msg [ctor] . *** binding result
*** CONSIDER the call O.Q(I).
***  bindMtd(O,Q,I,C S) try to find Q in class C or superclasses, then in S
***  boundMtd(O, process(Q @ C, true, Prog, Latt)) is the result.
***    where C is the actual class of the method

*** Bind run to a no operation program as a default if it's not defined in any
*** class of the object
eq bindMtd(O, 'run, D D' I, nil)
 = boundMtd(O, process('run @ 'null, true, end(nil),
                       ('caller : D), ('label : D'))) .

eq bindMtd(O, Q, I, ((C)[DL]) S')
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 = if Q in MMTD then
     boundMtd(O, newProcess(Q @ C, get(Q, MMTD), I))
   else
     bindMtd(O, Q, I, S S')
   fi
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > .

**********ATTRIBUTE inheritance with multiple inheritance
*** CMC ensures that all attributes names are (globally) different

op findAttr  : Oid InhList Subst -> Msg [ctor] . *** collect attributes
op foundAttr : Oid Subst         -> Msg [ctor] . *** resulting Subst
*** look in InhList, collect attributes in Subst, give result to Oid
eq findAttr(O, nil, A)    = foundAttr(O, A) .  *** collection completed.

eq findAttr(O,(((C)[DL]) S'),A)
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 = findAttr(O,(S S'),(A, evalSS(assign(IA,DL),A)))
   < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > .

op evalSS : InitSubst Subst -> Subst . *** eval Subst Sequentially
eq evalSS( no, A )         = no .
eq evalSS(((X : E),IA), A) = (X : eval(E,A)), evalSS(IA, (A,(X : eval(E,A)))).

************************** Names of new objects *******************
```

```
op newId : Cid Nat -> Qid .
eq newId(C, F)    = qid(string(C) + string(F)) .
***** some code to make string/qid total
op string : Nat    -> String  .
op string : Cid    -> String [ditto] .
op string : Oid    -> String [ditto] .
*** op string : Rat NzNat -> String [ditto] .
op qid    : Qid    -> Qid .
op qid    : String -> Qid [ditto] .
eq string(F)      = string(F,10) .
eq qid(Q)         = Q .
eq qid(string(Q)) = Q .
eq qid(string(C)) = qid(C) .
*** eq qid(S:String)  = 'invalid [owise] .

*********************** INTERPRETER RULES ***************************

*** assign ***

rl [assign] :
  < O : Ob | Cl: C, Pr: process(M, B, (X AL := E I); P, L),
             PrQ: W, Att: A, Lcnt: N >
=>
  if X in L then
    < O : Ob | Cl: C, Pr: process(M, B, (AL := evalList(I, (A, L))); P,
                                  (L, (X : eval(E, (A,L))))),
               PrQ: W, Att: A, Lcnt: N >
  else if X in A then
    < O : Ob | Cl: C, Pr: process(M, B, (AL := evalList(I, (A, L))); P, L),
               PrQ: W, Att: (A, (X : eval(E, (A, L)))), Lcnt: N >
    else error("variable_does_not_exist: " + string(X) + " in " + string(O))
    fi
  fi .

*** if_then_else ***
rl [if-el] :
  < O : Ob | Cl: C, Pr: process(M, B, if E th P el P' fi ; R, L),
             PrQ: W, Att: A, Lcnt: N >
=>
  if evalB(E, (A , L)) then
    < O : Ob | Cl: C, Pr: process(M, B, P ; R, L), PrQ: W, Att: A, Lcnt: N >
  else
    < O : Ob | Cl: C, Pr: process(M, B, P' ; R, L), PrQ: W, Att: A, Lcnt: N >
  fi .

*** while ***
rl [while] :
  < O : Ob | Cl: C, Pr: process(M, B, while E do R od ; P, L),
              PrQ: W, Att: A, Lcnt: N >
=>
  < O : Ob | Cl: C,
              Pr: process(M, B, (if E th (R ; (while E do R od)) fi); P, L),
              PrQ: W, Att: A, Lcnt: N > .

***  Object Creation: by new-msg and by new-statements
rl [new-start-req] :
  (new C (DL))
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
=>
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: (F + 1) >
  < newId(C,F): Ob | Cl: C, Pr: process('init @ 'null, false,
                                      (0 ? (nil));  'run (nil ; nil), no),
                        PrQ: none, Att: no, Lcnt: 1 >
  < newId(C, F): Qu | Ev: none, Keep: empty >
  findAttr(newId(C,F), C[DL], ('this : newId(C,F))) .

rl [new-start] :
  foundAttr(O, A)
  < O : Ob | Cl: C, Pr: process(M, B, (0 ?(nil)); P, L),
              PrQ: W, Att: A', Lcnt: 1 >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: 1 > .

rl [new-req] :
  < O : Ob | Cl: C, Pr: process(M, B, (X := new C'(I)); P, L),
              PrQ: W, Att: A, Lcnt: N >
  < C' # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
=>
  < O : Ob  | Cl: C, Pr: process(M, B, (X := newId(C', F)); P, L),
               PrQ: W, Att: A, Lcnt: N >
  < C' # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: (F + 1) >
  < newId(C',F): Ob | Cl: C', Pr: process('init @ 'null, false,
                                        (0 ?(nil)); 'run (nil ; nil), no),
                         PrQ: none, Att: no, Lcnt: 1 >
  < newId(C',F): Qu | Ev: none, Keep: empty >
  findAttr(newId(C',F), (C')[evalList(I,(A,L))], ('this : newId(C',F))) .

*** Non-deterministic choice ***

crl [nondet-p1] :
  < O : Ob | Cl: C, Pr: process(M, B, (P [] P'); R, L), PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P ; R, L), PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
if ready(P, (L, A), MM) .
```

```

crl [nondet-p2] :
  < O : Ob | Cl: C, Pr: process(M, B, (P [] P'); R, L), PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P' ; R, L), PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
if ready(P', (L, A), MM) .

*** Merge ***
crl [merge] : *** merge is comm, so this rule considers both P and P'.
  < O : Ob | Cl: C, Pr: process(M, B, (P ||| P'); R, L),
             PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
=>
  < O : Ob | Cl: C, Pr: process(M, B, (P MERGER P'); R, L),
             PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
if ready(P, (L, A), MM) .

rl [merger] :
  < O : Ob | Cl: C, Pr: process(M, B, ((SP ; P) MERGER P'); R, L),
             PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
=>
  < O : Qu | Ev: MM, Keep: H >
  if enabled(SP, (L, A), MM) then
    < O : Ob | Cl: C, Pr: process(M, B, (SP ; (P MERGER P')); R, L),
               PrQ: W, Att: A, Lcnt: N >
  else
    < O : Ob | Cl: C, Pr: process(M, B, (P' ||| (SP ; P)); R, L),
               PrQ: W, Att: A, Lcnt: N >
  fi .

eq empty MERGER P = P .

*** Suspension ***

crl [suspend] :  *** all kinds of code P
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
=>
  < O : Ob | Cl: C, Pr: none,
             PrQ: W : process(M, B, clear(P), L), Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
if not enabled(P, (L , A), MM) .

*** Guards ***
```

```

crl [boolguard] :
  < O : Ob | Cl: C, Pr: process(M, B, await E ; P, L), PrQ: W, Att: A, Lcnt: N >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: N >
if evalB(E, (A , L)) .

*** Reduction of label to number in guard
eq < O : Ob | Cl: C, Pr: process(M, B, await (X ? & G); P, L),
              PrQ: W, Att: A, Lcnt: N' >
 = < O : Ob | Cl: C, Pr: process(M, B, await (evalI(X, L)? & G); P, L),
              PrQ: W, Att: A, Lcnt: N' > .

rl [replyguard-inQ] : *** removal of reply guard
  < O : Ob | Cl: C, Pr: process(M, B, await(N ? & G); P, L),
             PrQ: W, Att: A, Lcnt: N' >
  < O : Qu | Ev: MM comp(O int(N) K), Keep: H >
=>
  < O : Ob | Cl: C, Pr: process(M, B, await G ; P, L),
             PrQ: W, Att: A, Lcnt: N' >
  < O : Qu | Ev: MM comp(O int(N) K), Keep: H > .

*** Evaluate guards in suspended processes ***

crl [PrQ-enabled] :
  < UTD : Ob | Cl: C, Pr: none, PrQ: process(M, B, P, L) : W, Att: A, Lcnt: N >
  < UTD : Qu | Ev: MM, Keep: H >
=>
  < UTD : Ob | Cl: C, Pr: process(M, false, P, L), PrQ: W, Att: A, Lcnt: N >
  < UTD : Qu | Ev: MM, Keep: H >
if enabled(P, (A , L), MM).

*** Notify waiting processes when call completion is in the queue
eq < O : Ob | Cl: C, Pr: PR, PrQ: process(M, B, await (N ? & G); P, L) : W,
              Att: A, Lcnt: N' >
   < O : Qu | Ev: MM comp(O int(N) DL), Keep: H >
 = < O : Ob | Cl: C, Pr: PR, PrQ: process(M, B, (await  G ; P), L) : W,
              Att: A, Lcnt: N' >
   < O : Qu | Ev: MM comp(O int(N) DL), Keep: H > .

*** METHOD CALLS ***

eq   ! Q(I) =   ! 'this . Q(I) . *** could alternatively use X@ this class

*** receive invocation message ***
rl [receive-call-req] :
  < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N' >
  < O : Qu | Ev: MM invoc(O, Q, DL), Keep: H >
```

```
=>
  < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N' >
  < O : Qu | Ev: MM, Keep: H >
  bindMtd(O, Q, DL, C[nil]) .

rl [receive-call-req] :
  < O : Qu | Ev: MM invoc(O, Q @ C, DL), Keep: H >
=>
  < O : Qu | Ev: MM, Keep: H >
  bindMtd(O, Q, DL, C[nil]) .

eq boundMtd(O, process(M, B, P, L))
   < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
 = < O : Ob | Cl: C, Pr: PR,
             PrQ: process(M, B, clear(P), L) : W, Att: A, Lcnt: N > .

*** local sync call with continue
eq < UTD : Ob | Cl: C, Pr: process(M, B, (N ? (J)); P, L),
                PrQ: process(M', true, P',
                             (('caller : UTD), ('label : int(N)), L')) : W,
                Att: A, Lcnt: N' >
 = < UTD : Ob | Cl: C, Pr: process(M', false, P' ; continue(N),
                                   (('caller : UTD), ('label : int(N)), L')),
                PrQ: process(M, B, await N ? ; (N ? (J)); P, L) : W,
                Att: A, Lcnt: N' > .

rl [continue] :
  < UTD : Ob | Cl: C, Pr: process(M, B, continue(N), L),
               PrQ: process(M', B', ((N ?(J)); P), L') : W, Att: A, Lcnt: N' >
=>
  < UTD : Ob | Cl: C, Pr: process(M', false, ((N ?(J)); P), L'),
               PrQ: W, Att: A, Lcnt: N' > .

rl [local-async-qualified-req] :
  < O : Ob | Cl: C, Pr: process(M, B, ( ! Q @ C'(I)); P, L),
             PrQ: W, Att: A, Lcnt: N >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: N + 1 >
  invoc(O, Q @ C', (O int(N) evalList(I, (A, L)))) .

*** REMOTE METHOD CALLS ***
eq < O : Ob | Cl: C, Pr: process(M, B, (Q ! M'(I)); P, L),
             PrQ: W, Att: A, Lcnt: N >
   < O : Qu | Ev: MM, Keep: H >
 = < O : Ob | Cl: C, Pr: process(M, B, (Q := int(N)); (! M' (I)); P, L),
             PrQ: W, Att: A, Lcnt: N >
   < O : Qu | Ev: MM, Keep: H ; [N] > .
```

```
rl [remote-async-reply] :
  < O : Ob | Cl: C, Pr: process(M, B, ( ! OE . Q(I)); P, L),
             PrQ: W, Att: A, Lcnt: N >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: N + 1 >
  invoc(eval(OE, (A, L)), Q, (O int(N) evalList(I, (A, L)))) .

*** Reduce sync. call  to async. call with reply
eq < O : Ob | Cl: C, Pr: process(M, B, (M'(I ; J)); P, L),
              PrQ: W, Att: A, Lcnt: N >
   < O : Qu | Ev: MM, Keep: H >
=  < O : Ob | Cl: C, Pr: process(M, B, ! M'(I); (N ?(J)); P, L),
              PrQ: W, Att: A, Lcnt: N >
   < O : Qu | Ev: MM, Keep: H ;[N] > .

*** emit reply message ***
rl [reply] :
  < O : Ob | Cl: C, Pr: process(M, B, (end(J)); P, L), PrQ: W, Att: A, Lcnt: N >
=>
  < O : Ob | Cl: C, Pr: process(M, B, P, L), PrQ: W, Att: A, Lcnt: N >
  comp(evalList('caller 'label J, (A, L))) .

*** reduce label
eq < O : Ob | Cl: C, Pr: process(M, B, (X ?(J)); P, L),
              PrQ: W, Att: A, Lcnt: N >
 = < O : Ob | Cl: C, Pr: process(M, B, (evalI(X,L)?(J)); P, L),
              PrQ: W, Att: A, Lcnt: N > .

*** blocking reply sentence ***
eq < O : Ob | Cl: C, Pr: process(M, B, (N ? (J)); P, L),
              PrQ: W, Att: A, Lcnt: N' >
   < O : Qu | Ev: MM comp(O int(N) DL), Keep: H >
 = < O : Ob | Cl: C, Pr: process(M, B, (J := DL); P, L),
              PrQ: W, Att: A, Lcnt: N' >
   < O : Qu | Ev: MM, Keep: H > .

*** Transport rules: include new message in queue ***
rl [invoc-msg] :
  < O : Qu | Ev: MM, Keep: H > invoc(O, M, DL)
=>
  < O : Qu | Ev: MM invoc(O, M, DL), Keep: H > .

rl [reply-msg] :
  < O : Qu | Ev: MM, Keep: H ;[N]; H' > comp(O int(N) DL)
=>
  < O : Qu | Ev: MM comp(O int(N) DL), Keep: H ; H' > .

op _in_ : Nat NatS -> Bool .
```

```
eq N in empty = false .
eq N in H ; [N'] = if N == N' then true else N in H fi .

crl [reply-msg] :
  < O : Qu | Ev: MM, Keep: H > comp(O int(N) DL)
=>
  < O : Qu | Ev: MM, Keep: H >
if N in H == false .

endm
```

Appendix C

Full Specification of the Update System

```
*** DYNAMIC UPDATES ***

mod UPDATE is
protecting INTERPRET .

vars C  C'  : Cid .
vars VN VN' VN'' : Nat .
vars F  F'  : Nat .
vars IA IA' : InitSubst .   *** list of initialized attribute declaratins
var  IL      : InitSubst .
vars S  S'  S'' : InhList . *** list of (parameterized) superclasses

vars B  B'   : Bool .
vars O  O'   : Oid .
vars Q  Q'   : Qid .
vars W  W'   : MProg .
vars L  L'   : Subst .
vars A  A'   : Subst .
vars X  Y    : Aid .
vars M  M'   : Mid .
vars N  N'   : Nat .
vars D  D'   : Data .
vars DL DL'  : DataList .
vars AL AL'  : AidList .
vars P  P' R : ProgList .
var  J       : List .
var  PR      : Process .
var  H  H'   : NatS .
vars E  E'   : Expr .
vars G G'  : Guard .

var MMTD   : MMtd .
var MMTD'  : MMtd .
var CLASS  : Class .
var GHOST  : Ghost .
var OBJECT : Object .
var MSG    : Msg .
var MSG'   : Msg .
var MM     : MMsg .
var STM    : Stm .
var QUEUE  : Queue .
var UTD    : UTDOid .

vars CONFIG CONFIG' : Configuration .

*** Ghosts are used to help messages find objects which are in a different
*** configuration

sort Ghost .
subsort Ghost < Configuration .
op <_: Ghost | Cl:_, Old:_> : Oid Cid Bool -> Ghost [ctor] .

*** Add convert routine methods and methods with parameter conversion

op <_: Mtdname | Latt:_, Code:_, Convert:_> :
     Qid Subst ProgList InitSubst -> Mtd
     [ctor format (m! om m m m m m g m m m m m m! on)] .

op < Convert | Latt:_, Code:_, Finish:_, Restart:_> :
     Subst ProgList AidList AidList -> Mtd [ctor] .

eq get(Q, < Q : Mtdname | Latt: L, Code: P, Convert: IL > * MMTD)
 = < Q : Mtdname | Latt: L, Code: P, Convert: IL > .

eq Q in (< Convert | Latt: L, Code: P, Finish: AL, Restart: AL' > * MMTD)
 = Q in MMTD .
eq Q in (< Q' : Mtdname | Latt: L, Code: P, Convert: IL > * MMTD)
 = if Q == Q' then true else Q in MMTD fi .

eq newProcess(M, < Q : Mtdname | Latt: L, Code: P, Convert: IA > , D D' DL)
 = process(M, true, P, (('caller : D), ('label : D'), assign(L, DL))) .

*** updating starts the conversion process
op update  : Configuration Configuration -> Configuration .
eq update(CONFIG, CONFIG')
 = merge(classes(CONFIG), CONFIG')
   makeGhosts(CONFIG)
   outdated(CONFIG) .

*** merging classes includes the outdated classes that don’t exist in
*** the update in the new configuration
op merge : Configuration Configuration -> Configuration .
eq merge(none, CONFIG’) = CONFIG’ .
eq merge(< C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > CONFIG, CONFIG’)
 = if not (C in CONFIG’) then
     < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
   else none fi
   merge(CONFIG, CONFIG’) .

*** outdated configuration  becomes a term in the new configuration
op outdated : Configuration -> Configuration [ctor] .

*** convert messages control state conversion for objects
op convert : Oid InhList Subst MProg -> Msg [ctor] .

*** outdated are marked so they cannot load processes without good reason
op outdated : Qid -> Oid  [ctor] .
op endconvert : Cid -> Prog [ctor] .

*** if an object has the same version of all classes in the old configuration
*** as in the new configuration then move it
rl [migrate-object] :
  outdated(
    < outdated(O) : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
    < outdated(O) : Qu | Ev: MM, Keep: H >
    findInheritance(outdated(O), nil, S, AL)
    CONFIG)
  findInheritance(outdated(O), nil, S, AL)
  < O : Ghost | Cl: C, Old: false >
=>
  < O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
  < O : Qu | Ev: MM, Keep: H >
  outdated(CONFIG < O : Ghost | Cl: C, Old: true >) .

*** if an object is ready to be converted, then convert it
crl [convert-object] :
  outdated(
    < outdated(O) : Ob | Cl: C, Pr: none, PrQ: W, Att: A, Lcnt: N >
    findInheritance(outdated(O), nil, S’, AL’)
    < outdated(O) : Qu | Ev: MM, Keep: H >
    CONFIG
  )
  findInheritance(outdated(O), nil, S, AL)
  < O : Ghost | Cl: C, Old: false >
=>
```

```
  convert(O, combine(S, S'), A, W)
  < outdated(O) : Ob | Cl: C, Pr: none,
                       PrQ: paramcopy(W, 'null, none),
                       Att: ('this : O), Lcnt: N >
  < outdated(O) : Qu | Ev: comps(MM), Keep: H >
  outdated(CONFIG invocs(MM) < O : Ghost | Cl: C, Old: true >)
if not waiting(AL, W) .

crl [convert-object-continue] :
  outdated(
    < outdated(O) : Ob | Cl: C, Pr: process(M, B, continue(N'), L),
                         PrQ: W, Att: A, Lcnt: N >
    findInheritance(outdated(O), nil, S', AL')
    < outdated(O) : Qu | Ev: MM, Keep: H >
    CONFIG
  )
  findInheritance(outdated(O), nil, S, AL)
  < O : Ghost | Cl: C, Old: false >
=>
  convert(O, combine(S, S'), A, W)
  < outdated(O) : Ob | Cl: C, Pr: none,
                       PrQ: paramcopy(W, 'null, none),
                       Att: ('this : O), Lcnt: N >
  < outdated(O) : Qu | Ev: comps(MM), Keep: H >
  outdated(CONFIG invocs(MM) < O : Ghost | Cl: C, Old: true >)
if not waiting(AL, W) .

*** combine sets up an inheritance list which reflects which classes are old
*** and which are new, but using the new inheritance graph.

op combine : InhList InhList -> InhList .
eq combine((C # VN)[DL], ((C' # VN')[DL']) S)
 = if (C == C') then (C' # VN')[DL'] else combine((C # VN)[DL], S) fi .
eq combine((C # VN)[DL], nil) = (C # VN)[DL] .
eq combine(((C # VN)[DL]) S, S') = combine((C # VN)[DL], S') combine(S, S') .

*** update rule to convert class attributes using convert routine
crl [update-object-with-conversion] :
  < C # VN : Cl | Inh: S, Att: IA,
                  Mtds: < Convert | Latt: L, Code: P,
                                    Finish: AL, Restart: AL' > * MMTD,
                  Ocnt: F >
  convert(O, ((C # VN')[DL]) S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: none, PrQ: W, Att: A, Lcnt: N >
=>
  < C # VN : Cl | Inh: S, Att: IA,
                  Mtds: < Convert | Latt: L, Code: P,
                                    Finish: AL, Restart: AL' > * MMTD,
                  Ocnt: F >
  convert(O, S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: process('convert, false,
                                           P ; endconvert(C),
                                           (L, age(A', C))),
                       PrQ: W, Att: (A, evalSS(IA, A)), Lcnt: N >
if VN > VN' .

*** finish convert routine by converting parameters
rl [finish-conversion] :
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  convert(O, S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: process('convert, false, endconvert(C), L),
                       PrQ: W, Att: A, Lcnt: N >
=>
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  < outdated(O) : Ob | Cl: C', Pr: none,
                       PrQ: W : paramconv(W', C, MMTD, ('this : O)),
                       Att: A, Lcnt: N >
  convert(O, S', A', W') .

*** update rule to copy class attributes for unchanged superclasses
crl [update-object-without-conversion] :
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  convert(O, ((C # VN')[DL]) S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: none, PrQ: W, Att: A, Lcnt: N >
=>
  < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  convert(O, S', A', W')
  < outdated(O) : Ob | Cl: C', Pr: none, PrQ: W : paramcopy(W', C, MMTD),
                       Att: (A, copy(evalSS(IA, A), A')), Lcnt: N >
if VN' = VN .

*** when all classes have been processed the object is up to date
eq convert(O, nil, A', W')
   < outdated(O) : Ob | Cl: C, Pr: none, PrQ: W, Att: A, Lcnt: N >
   < outdated(O) : Qu | Ev: MM, Keep: H >
 = < O : Ob | Cl: C, Pr: none, PrQ: W, Att: A, Lcnt: N >
   < O : Qu | Ev: MM, Keep: H > .

*** mark outdated objects and find inheritance in old and new configuration
*** to prepare for conversion
eq outdated(< UTD : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
            < UTD : Qu | Ev: MM, Keep: H >
            CONFIG)
 = outdated(< outdated(UTD) : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N >
            < outdated(UTD) : Qu | Ev: MM, Keep: H >
            findInheritance(outdated(UTD), (C)[nil], nil, missing)
```

```
                CONFIG)
    findInheritance(outdated(UTD), (C)[nil], nil, nil) .

*** fill in finishlist in outdated configuration
eq outdated(findInheritance(outdated(UTD), nil, S, missing)
                CONFIG)
    findInheritance(outdated(UTD), nil, S', AL)
 = outdated(findInheritance(outdated(UTD), nil, S, AL)
                CONFIG)
    findInheritance(outdated(UTD), nil, S', AL) .

*** the findInheritance takes a class and finds all superclasses
op missing : -> AidList [ctor] .
op findInheritance  : Oid InhList InhList AidList -> Msg [ctor] .
eq findInheritance(O,(((C)[DL]) S'), S'', AL)
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 = findInheritance(O,(S S'), ((C # VN)[DL]) S'', AL finishlist(MMTD))
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > .

eq ((C)[DL]) S ((C)[DL]) S' = ((C)[DL]) S S' . *** remove duplicates

*** separates invocs and comps
ops invocs comps : MMsg -> MMsg .

eq invocs(invoc(O, M, DL) MM) = invoc(O, M, DL) invocs(MM) .
eq invocs(comp(O int(N) DL) MM) = invocs(MM) .
eq invocs(none) = none .

eq comps(invoc(O, M, DL) MM) = comps(MM) .
eq comps(comp(O int(N) DL) MM) = comp(O int(N) DL) comps(MM) .
eq comps(none) = none .

*** separates classes and objects (including messages and queues)
ops classes objects : Configuration -> Configuration .

eq classes(CLASS CONFIG) = CLASS classes(CONFIG) .
eq classes(OBJECT CONFIG) = classes(CONFIG) .
eq classes(QUEUE CONFIG) = classes(CONFIG) .
eq classes(MSG CONFIG) = classes(CONFIG) .
eq classes(GHOST CONFIG) = classes(CONFIG) .
eq classes(outdated(CONFIG') CONFIG) = classes(CONFIG) .
eq classes(none) = none .

eq objects(CLASS CONFIG) = objects(CONFIG) .
eq objects(OBJECT CONFIG) = OBJECT objects(CONFIG) .
eq objects(QUEUE CONFIG) = QUEUE objects(CONFIG) .
eq objects(MSG CONFIG) = MSG objects(CONFIG) .
eq objects(GHOST CONFIG) = objects(CONFIG) .
eq objects(none) = none .

*** check for class in configuration
op _in_ : Cid Configuration -> Bool .
eq C in (none).Configuration = false .
eq C in < C’ # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F > CONFIG
 = C == C’ or C in CONFIG .

*** makeGhosts makes a ghost for every object in the system
op makeGhosts : Configuration -> Configuration .
eq makeGhosts(< O : Ob | Cl: C, Pr: PR, PrQ: W, Att: A, Lcnt: N > CONFIG)
 = < O : Ghost | Cl: C, Old: false > makeGhosts(CONFIG) .
eq makeGhosts(< O : Ghost | Cl: C, Old: false > CONFIG)
 = < O : Ghost | Cl: C, Old: false > makeGhosts(CONFIG) .
eq makeGhosts(outdated(CONFIG’) CONFIG) = makeGhosts(CONFIG) .
eq makeGhosts(GHOST CONFIG) = makeGhosts(CONFIG) .
eq makeGhosts(CLASS CONFIG) = makeGhosts(CONFIG) .
eq makeGhosts(QUEUE CONFIG) = makeGhosts(CONFIG) .
eq makeGhosts(MSG CONFIG) = makeGhosts(CONFIG) .
eq makeGhosts(none) = none .

*** Check for unfinished processes corresponding to the methods in the list
op waiting : AidList MProg -> Bool .
eq waiting(nil, W) = false .
eq waiting(X, none) = false .
eq waiting(X, process(Y, B, P, L) : W) = (X == Y and not B) or waiting(X, W) .
eq waiting(X Y AL, W) = waiting(X, W) or waiting(Y AL, W) .

*** Extract finishlist from methods
op finishlist : MMtd -> AidList .
eq finishlist(< Convert | Latt: L, Code: P, Finish: AL, Restart: AL’ > * MMTD)
 = AL .
eq finishlist(< M : Mtdname | Latt: L’, Code: P’, Convert: IL > * MMTD)
 = finishlist(MMTD) .
eq finishlist(< M : Mtdname | Latt: L’, Code: P’ > * MMTD) = finishlist(MMTD) .
eq finishlist(none) = nil .

*** Extract restartlist from methods
op restartlist : MMtd -> AidList .
eq restartlist(< Convert | Latt: L, Code: P, Finish: AL, Restart: AL’ > * MMTD)
 = AL’ .
eq restartlist(< M : Mtdname | Latt: L’, Code: P’, Convert: IL > * MMTD)
 = restartlist(MMTD) .
eq restartlist(< M : Mtdname | Latt: L’, Code: P’ > * MMTD) = restartlist(MMTD) .
eq restartlist(none) = nil .

*** age and copy of attributes
op age : Subst Cid -> Subst .
```

```
eq age((((Q @ C) : D), A) , C')
 = (if C == C' then ((Q @ 'old) : D) else no fi) , age(A, C') .
eq age(((Q : D), A) , C') = ((Q @ 'old) : D) , age(A, C') .
eq age(no, C') = no .

op copy : Subst Subst -> Subst .
eq copy(((X : D), A) , A')
 = if (X in A') then ((X : val(X, A')), copy(A, A'))
                else ((X : D), copy(A, A')) fi .
eq copy(no, A') = no .

op toList : Subst -> List .
eq toList(((X : D), A)) = D toList(A) .
eq toList(no) = nil .

*** convert and copy of parameters
op conv : Subst Mtd Subst -> Subst .
eq conv(L, < M : Mtdname | Latt: L', Code: P' >, A)
 = copy((('caller : 'null), ('label : 'null), L'), L) .
eq conv(L, < M : Mtdname | Latt: L', Code: P', Convert: IL >, A)
 = evalSS(IL, (age(L,'null), A)) .

op program : Mtd -> ProgList .
eq program(< M : Mtdname | Latt: L', Code: P' >) = P' .
eq program(< M : Mtdname | Latt: L', Code: P', Convert: IL >) = P' .

op paramconv : MProg Cid MMtd Subst -> MProg .
eq paramconv(none, C, MMTD, A) = none .

ceq paramconv(process(M @ C, B, P, L) : W, C, MMTD, A)
  = process(M @ C, true, program(get(M, MMTD)), conv(L, get(M, MMTD), A)) :
    paramconv(W, C, MMTD, A)
 if (M in MMTD) and ((B == true) or ((M @ C) in restartlist(MMTD))) .

ceq paramconv(process(M @ C, B, P, L) : W, C', MMTD, A)
  = paramconv(W, C', MMTD, A)
 if (not (C == C')) or (not (M in MMTD) and ((M @ C) in restartlist(MMTD))) .

ceq paramconv(process(M @ C, B, P, L) : W, C, MMTD, A)
  = process(M @ C, B, P, L) : paramconv(W, C, MMTD, A)
 if (B == false and not ((M @ C) in restartlist(MMTD))) or
    (not (M in MMTD) and not ((M @ C) in restartlist(MMTD))) .

op paramcopy : MProg Cid MMtd -> MProg .
eq paramcopy(process(M @ C, B, P, L) : W, C', MMTD)
 = if (C == C') and ((M in MMTD) or C == 'null) then
     process(M @ C, B, P, L)
   else
```

```
     none
   fi : paramcopy(W, C', MMTD) .
eq paramcopy(none, C, MMTD) = none .

*** guided execution rules -- allow waiting processes to be loaded
crl [guided-PrQ-enabled] :
  < outdated(O) : Ob | Cl: C, Pr: none,
                       PrQ: process(M, false, P, L) : W, Att: A, Lcnt: N >
  < outdated(O) : Qu | Ev: MM, Keep: H >
  findInheritance(outdated(O), nil, S, AL)
=>
  < outdated(O) : Ob | Cl: C, Pr: process(M, false, P, L),
                              PrQ: W, Att: A, Lcnt: N >
  < outdated(O) : Qu | Ev: MM, Keep: H >
  findInheritance(outdated(O), nil, S, AL)
if enabled(P, (A , L), MM) and waiting(AL, process(M, false, P, L)) .

crl [guided-continue] :
  < outdated(O) : Ob | Cl: C, Pr: process(M, false, continue(N), L),
                       PrQ: process(M', false, ((N ?(J)); P), L') : W,
                       Att: A, Lcnt: N' >
  findInheritance(outdated(O), nil, S, AL)
=>
  < outdated(O) : Ob | Cl: C, Pr: process(M', false, ((N ?(J)); P), L'),
                       PrQ: W,
                       Att: A, Lcnt: N' >
  findInheritance(outdated(O), nil, S, AL)
if waiting(AL, process(M', false, empty, no)) .

*** do not pass around the 'outdated' tag -- only objects should have it
eq invoc(O, M, outdated(O') DL) = invoc(O, M, O' DL) .
eq invoc(outdated(O), M, DL) CLASS
 = invoc(O, M, DL) CLASS .
eq comp(outdated(O) int(N) DL) CLASS
 = comp(O int(N) DL) CLASS .

*** make sure invocs and comps in queues matches the object idea including tag
eq < O : Qu | Ev: MM invoc(outdated(O), M, DL), Keep: H >
 = < O : Qu | Ev: MM invoc(O, M, DL), Keep: H > .
eq < O : Qu | Ev: MM comp(outdated(O) int(N) DL), Keep: H ; H' >
 = < O : Qu | Ev: MM comp(O int(N) DL), Keep: H ; H' > .

eq < outdated(O) : Qu | Ev: MM invoc(O, M, DL), Keep: H >
 = < outdated(O) : Qu | Ev: MM invoc(outdated(O), M, DL), Keep: H > .
eq < outdated(O) : Qu | Ev: MM comp(O int(N) DL), Keep: H ; H' >
 = < outdated(O) : Qu | Ev: MM comp(outdated(O) int(N) DL), Keep: H ; H' > .

*** bind outdated invocs to prepare for conversion
```

```
rl [bind-outdated-unqualified-invoc-msg] :
  < O : Ghost | Cl: C, Old: true >
  invoc(O, Q, DL)
=>
  < O : Ghost | Cl: C, Old: true >
  bindInvoc(invoc(O, Q, DL), C[nil]) .

rl [bind-outdated-qualified-invoc-msg] :
  < O : Ghost | Cl: C, Old: true >
  invoc(O, M @ C', DL)
=>
  < O : Ghost | Cl: C, Old: true >
  bindInvoc(invoc(O, M, DL), C'[nil]) .

*** bindInvoc makes sure all invocs are qualified and parameters bound
op bindInvoc  : Msg InhList -> Msg [ctor] .
op boundInvoc : Oid Cid Process -> Msg [ctor] .
ceq bindInvoc(invoc(O, M, DL), ((C)[DL']) S')
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  = boundInvoc(O, C # VN, newProcess(M @ C, get(M, MMTD), DL))
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 if M in MMTD .

*** couldn't bind in this class, look in next class in the inheritance graph
ceq bindInvoc(invoc(O, M, DL), ((C)[DL']) S')
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  = bindInvoc(invoc(O, M, DL), S' S)
    < C # VN : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
 if not (M in MMTD) .

*** move invocs across version barrier
rl [outdated-invoc-msg] :
  < C # VN' : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  outdated(boundInvoc(O, C # VN, process(M @ C, B, P, L)) CONFIG)
=>
  if VN == VN' then
    invoc(O, M @ C, toList(L))
  else
    if (M in MMTD) then
      invoc(O, M @ C, toList(conv(L, get(M, MMTD), ('this : O))))
    else
      none
    fi
  fi
  < C # VN' : Cl | Inh: S, Att: IA, Mtds: MMTD, Ocnt: F >
  outdated(CONFIG) .

*** move comps across version barrier
```

```
rl [outdated-reply-msg] :
  outdated(< O : Ghost | Cl: C, Old: true > comp(O int(N) DL) CONFIG)
=>
  comp(O int(N) DL)
  outdated(< O : Ghost | Cl: C, Old: true > CONFIG).

rl [reply-outdated-msg] :
  comp(O int(N) DL)
  < O : Ghost | Cl: C, Old: false >
  outdated(CONFIG)
=>
  < O : Ghost | Cl: C, Old: false >
  outdated(comp(O int(N) DL) CONFIG) .

*** Transport rules: include new message in outdated queue ***
rl [invoc-msg-outdated] :
  < outdated(O) : Qu | Ev: MM, Keep: H > invoc(O, M, DL)
=>
  < outdated(O) : Qu | Ev: MM invoc(outdated(O), M, DL), Keep: H > .

rl [reply-msg-outdated] :
  < outdated(O) : Qu | Ev: MM, Keep: H ;[N]; H' > comp(O int(N) DL)
=>
  < outdated(O) : Qu | Ev: MM comp(outdated(O) int(N) DL), Keep: H ; H' > .

*** garbage collect outdated configurations with no objects in them
ceq outdated(CONFIG) = none if objects(CONFIG) = none .

*** local sync call in convert routine ***
eq < outdated(O) : Ob | Cl: C, Pr: process('convert, B, (N ? (J)); P, L),
              PrQ: process(M', true, P',
                           (('caller : O), ('label : int(N)), L')) : W,
              Att: A, Lcnt: N' >
 = < outdated(O) : Ob | Cl: C, Pr: process(M', false, P' ; continue(N),
                              (('caller : O), ('label : int(N)), L')),
              PrQ: process('convert, B, await N ? ; (N ? (J)); P, L) : W,
              Att: A, Lcnt: N' > .

rl [continue-convert] :
  < outdated(O) : Ob | Cl: C, Pr: process(M, B, continue(N), L),
                       PrQ: process('convert, B', ((N ?(J)); P), L') : W,
                       Att: A, Lcnt: N' >
=>
  < outdated(O) : Ob | Cl: C, Pr: process('convert, false, ((N ?(J)); P), L'),
                       PrQ: W,
                       Att: A, Lcnt: N' > .

endm
```

Bibliography

[1] Marte Arnestad. En abstrakt maskin for Creol i Maude. Master's thesis, Department of Informatics, University of Oslo, November 2003. In norwegian. Available from `http://heim.ifi.uio.no/~creol`.

[2] Gavin Bierman, Michael W. Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *Proceedings of USE 2003: the Second International Workshop on Unanticipated Software Evolution*, April 2003.

[3] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, MIT, 1983. Also as MIT LCS Tech. Report 303.

[4] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, August 2002.

[6] R.P. Cook. *Mod – a language for distributed programming. *IEEE Transactions on Software Engineering*, SE6(6):563–571, November 1980.

[7] Creol homepage. `http://www.ifi.uio.no/~creol`.

[8] Edsger W. Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[9] Dominic Duggan. Type-based hot swapping of running modules. In *International Conference on Functional Programming*, pages 62–73, 2001.

[10] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 470–476. IEEE Computer Society Press, 1976.

[11] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, Message Passing Interface Forum, 1994.

[12] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[13] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes - a lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, pages 65–76, 1998.

[14] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. In *Proceedings of the 2nd IEEE International Conference on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, September 2004.

[15] Einar Broch Johnsen and Olaf Owe. Inheritance in the presence of asynchronous method calls. In *Proceedings of the 38th Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society Press, January 2005.

[16] Einar Broch Johnsen, Olaf Owe, and Marte Arnestad. Combining active and reactive behavior in concurrent objects. In *Proceedings of the Norwegian Informatics Conference (NIK'03)*, pages 193–204. Tapir Academic Publisher, November 2003.

[17] Einar Broch Johnsen, Olaf Owe, and Isabelle Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In *Proceedings of the 7th IFIP International Conference on Formal Methods for Object-based Distrubuted Systems (FMOODS 2005)*, 2005. To appear.

[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[19] Insup Lee. *DYMOS: A Dynamic Modification System.* PhD thesis, University of Wisconsin - Madison, 1983.

[20] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[21] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[22] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361. Springer-Verlag, 2000.

[23] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[24] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois, 1992.

[25] O. Owe and I. Ryl. On combining object orientation, openness and reliability. In *Proceedings of the Norwegian Informatics Conference (NIK'99).* Tapir Academic Publisher, November 1999.

[26] E. W. Stark. Foundations of a theory of specification for distributed systems. Technical Report MIT/LCS/TR-342, MIT, 1984.

[27] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, January 2005.