A Dynamic Binding Strategy for Multiple Inheritance and Asynchronously Communicating Objects

Einar Broch Johnsen and Olaf Owe

Department of Informatics, University of Oslo, Norway
{einarj,olaf}@ifi.uio.no

Abstract This paper considers an integration of asynchronous communication, virtual binding, and multiple inheritance. Object orientation is the leading paradigm for concurrent and distributed systems, but the tightly synchronized RPC communication model seems unsatisfactory in the distributed setting. Asynchronous messages are better suited, but lack the structure and discipline of traditional object-oriented methods. The integration of messages in the object-oriented paradigm is unsettled, especially with respect to inheritance and redefinition. Asynchronous method calls have been proposed in the Creol language, reducing the cost of waiting for replies in the distributed environment while avoiding low-level synchronization constructs such as explicit signaling. A lack of reply to a method call need not lead to deadlock in the calling object. Creol has an operational semantics defined in rewriting logic. This paper considers a formal operational model of multiple inheritance, virtual binding, and asynchronous communication between concurrent objects, extending the semantics of Creol.

1 Introduction

Object orientation is the leading paradigm for concurrent and distributed systems. The importance of such systems is increasing in society, driving the need for formal models and reasoning support for object-oriented distributed systems. With the current domination of languages such as Java and C++, one may think that there is only one way to understand object-oriented languages. In the setting of distributed systems, these languages may be criticized for their approach to concurrency as well as to communication. An alternative approach is taken in the Creol language: concurrent objects typed by interfaces which communicate by means of asynchronous method calls. This communication model integrates asynchronous message passing with the high-level structuring mechanism of method definition and invocation [20].

In this paper we discuss multiple inheritance in the setting of open distributed systems and consider the combination of multiple inheritance and virtual (or late) binding. Multiple inheritance provides a flexible way to combine class hierarchies, but is generally considered error-prone. Multiple inheritance is often explained in terms of run-time data structures such as virtual pointer tables [38], which are complex and hard to understand. High-level formal treatments are scarce (e.g., [3, 6, 15, 34]) but needed to clarify intricacies, thus facilitating design and correctness reasoning for programs using multiple inheritance. In this paper, an operational semantics capturing multiple inheritance and virtual binding of methods for Creol is defined, extending work reported in [22].

In: F.S. de Boer, M. Bonsangue, S. Graf, W.-P. de Roever (Eds.):
Proc. 3rd. Intl. Symp. on Formal Methods for Components and Objects (FMCO'04),
p. 274–295, LNCS 3657, Springer-Verlag, Nov. 2005.

In particular, common restrictions on the name space of methods to avoid name conflicts either severely limit the use of class inheritance or become impractical in large class hierarchies and break the encapsulation principle for class inheritance. In order to maintain local reasoning control without abandoning common features of virtual binding such as method overloading and overriding, virtual binding of method calls must be handled carefully. For this purpose, a dynamic *pruned binding strategy* is introduced in this paper and formalized with an operational semantics given in rewriting logic. Rewriting logic [29] is chosen due to its high level of abstraction with inherent support for distribution, concurrency, and asynchronous communication, as well as its simulation and model checking facilities through the Maude tool [8, 30]. This allows us to focus the formalization on issues related to inheritance and virtual binding. The strategy is integrated in the Creol interpreter in Maude. An example demonstrates that a form of binding anomaly is avoided with this strategy.

Paper overview. Sect. 2 provides a background discussion on multiple inheritance. Sect. 3 introduces the Creol language. Sect. 4 extends Creol with mechanisms for multiple inheritance and pruned virtual binding, Sect. 5 illustrates the mechanisms, and Sect. 6 defines its operational semantics. Sect. 7 considers related work and Sect. 8 concludes the paper.

2 Inheritance: Reuse of Behavior and Reuse of Code

Inheritance is a powerful feature of object orientation, but its exact role as a structuring mechanism for programs varies between different object-oriented languages. With *single* inheritance, a class is derived from one direct ancestor class, while with *multiple* inheritance there may be several direct ancestors. Inheritance may be understood as a mechanism for sharing and specialization of behavior as well as code. Formal approaches to inheritance tend to favor the first interpretation and understand inheritance in terms of behavioral reuse, obeying the *substitutability* principle: As a subclass is a specialization of a superclass, an object of the subclass may masquerade as an object of the superclass. This interpretation of inheritance has led to an active field of research on behavioral subtyping [2, 14, 27], identifying conditions for safe substitutability. A *type* describes a collection of objects which share the same externally observable behavior. Subtyping provides a powerful structuring mechanism for defining, specializing, and understanding the external behavior of objects.

A *class* describes a collection of objects which share the same internal structure; i.e., attributes and method definitions. Code inheritance provides an equally powerful mechanism for defining, specializing, and understanding the imperative structure of classes through code reuse and modification. Class extension and method redefinition are convenient both for development and understanding of code. Calling superclass methods in a subclass method enables *reuse in redefined methods*, making the relationship between the method versions explicit. Thus, this facility is clearly superior to cut-and-paste programming with regard to the ease with which existing code may be inspected and understood and it is also clearly superior to inheritance mechanisms which do not distinguish between locally defined and inherited definitions. A denotational semantics for code sharing and reuse based on single inheritance is given in [9].

Although many languages identify the subclass and subtype relations, in particular for parameter passing, several authors argue that inheritance relations for code and for behavior should be distinct [2, 5, 10, 37]. From the pragmatic point of view, combining these relations leads to severe restrictions on code reuse which seem unattractive to programmers. From a reasoning perspective, the separation of these relations allows greater expressiveness while providing type safety. In order to solve the conflict between unrestricted code reuse in subclasses, and behavioral subtyping and incremental reasoning control [27, 37], we use behavioral interfaces [19, 21] to type object variables (i.e., references) and external (remote) calls, and allow multiple inheritance for both interfaces and classes. Interface inheritance is restricted to a form of behavioral subtyping, whereas class inheritance may be used freely for code reuse. A class may implement several interfaces, provided that it satisfies their syntactic and semantic requirements. An object of class C supports an interface I if the class C implements I. Reasoning control is ensured by substitutability at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subinterface of I* in a context depending on I, although the latter object may be of another class. Subclassing is unrestricted in the sense that implementation claims and class invariants are not in general inherited.

With distinct inheritance and subtyping hierarchies, class inheritance could allow a subset of the attributes and methods of a class to be inherited. However, this would require considerable work establishing invariants for parts of the superclass that appear desirable for inheritance, either anticipating future needs or while designing subclasses. The *encapsulation principle* for class inheritance states that it should suffice to work at the subclass level to ensure that the subclass is well-behaved when inheriting from a superclass: Code design as well as new proof obligations should occur in the subclass only. Situations that break this principle are called inheritance anomalies [28, 32]. Reasoning considerations therefore suggest that all attributes and methods of a superclass are inherited, but method redefinition may violate the semantic requirements of an interface of the superclass.

2.1 Multiple Inheritance

The focus of this paper is the formalization of an operational semantics for code reuse through class level multiple inheritance. Multiple inheritance seems desirable because it provides much better possibilities for sharing than single inheritance, allowing named features (attributes and methods) from several classes to be integrated. The combination of single inheritance and interfaces is sometimes proposed as an alternative to multiple inheritance, but this approach has some difficulties. In particular virtual binding does not integrate directly with delegation, and the use of private methods as well as program variables from superclasses is problematic. Tempero and Biddle show how the reusability of the Java Core API is adversely affected by the lack of multiple inheritance [39].

Multiple inheritance is found in languages such as C++ [38], CLOS [12], Eiffel [31], Full Maude [8], POOL [2], and Self [7]. Although multiple inheritance provides a flexible way to describe class hierarchies, it is avoided or only allowed in a restricted version (such as interfaces, abstract classes, or traits) in many languages, e.g., Java and C#. Apart from semantic issues, two important arguments against multiple inheritance are:

(1) the run-time system of languages with multiple inheritance is more complex and less efficient, and (2) inheriting from many classes increases the possibility of programmer mistakes. However, efficient run-time systems for languages with multiple inheritance have been developed [25, 38]. In order to address argument (2), formal methods may contribute to a better understanding of existing multiple inheritance mechanisms and hopefully contribute to mechanisms with better support for reasoning. The multiple inheritance relation is transitive and defines a class hierarchy structured as a directed acyclic graph. A class in the class hierarchy extends the features declared in its inherited classes (or superclasses), possibly overloading or redefining some of these features. We shall say that a feature is defined *above* a class C if it is defined in C or in at least one of the classes inherited by C.

2.2 Naming Policies for Conflict Resolution

From a reasoning perspective, the difficulties regarding multiple inheritance occur when the name spaces of several inherited classes conflict, both with regard to program variables and methods [24]. A name conflict is *vertical* if a name occurs in a class and in one of its ancestors, corresponding to overridden method declarations. A name conflict is *horizontal* if the name occurs in distinct branches of the graph. While vertical name conflicts are fairly well understood, different solutions have been proposed to deal with horizontal name conflicts. One approach is to remove ambiguities by *explicit resolution*. This is achieved if a name which is inherited from several superclasses is redefined in the subclass or directed to a superclass definition through qualification [38]. If a class is multiply inherited, qualification by class path leads to a duplication of attributes while qualification by class name leads to unification (virtual classes in C++). This choice leads to one or two copies of the attributes of class A in Fig. 1b. Path qualification is not always sufficient to distinguish two inherited instances of a class and may therefore fail, as illustrated by Fig. 1c. Explicit resolution can also be achieved by renaming attributes and methods [2, 31], thus eliminating name conflicts. When there are no name conflicts, the inheritance graph may be *linearized* and the need for explicit support for multiple inheritance in the run-time system is avoided. This is also the case with mixin-based inheritance [4], and with traits [35]. Mixins and traits are integrated in the linear inheritance graph to extend and modify the resulting behavior of the superclass. However linearization has been criticized for changing the parent-child relationship between classes in the inheritance hierarchy [36].

Ambiguities may also be seen as a natural feature of multiple inheritance, occurring when related methods in different superclass hierarchies are given the same name. From this point of view it seems less desirable to apply a naming discipline which forces the programmer to modify names a posteriori, making the class definitions more difficult to understand. Taking this approach, ambiguities are addressed by *implicit resolution*. Three approaches can be used to explain implicit resolution of ambiguous method names. First, methods with the same name may be seen as equally appropriate. In this case the method definitions may nondeterministically compete for selection, as in Full Maude [8] and the Join-calculus [15]. (Redefinition is not supported by Full Maude, whereas renaming is required in the Join-calculus.) Second, methods of the same name may be jointly selected, extending the binding strategy of Beta to multiple inheritance.

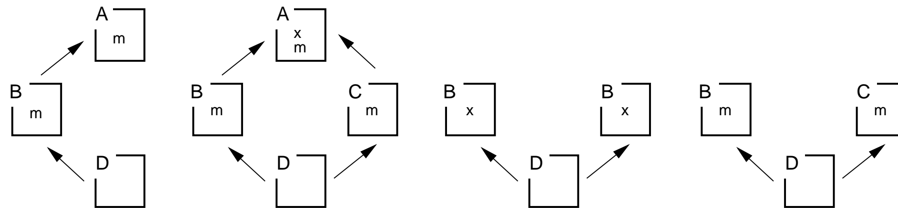

Figure 1. Examples of class inheritance: (a) single inheritance, (b) a common ancestor in the inheritance graph, (c) duplicate inheritance, and (d) multiple inheritance.

Third, ambiguities may be solved by fixing the *order* of the inherited classes; this way the strategy for selecting method definitions will be unambiguous [7,12,22]. This seems desirable as it leaves the programmer in control.

2.3 Virtual Binding

Virtual binding (or dynamic dispatch) is a powerful mechanism of object orientation, originally introduced in [11] for single inheritance in Simula. A method is virtually bound if the body corresponding to a method invocation is selected at run-time. Virtual binding is applied when the actual class of an object is not statically known. Traditionally, this happens when a method invoked from a class is overridden above the actual class of the object. When objects are typed by interfaces, many classes may implement the same interface. Consequently all external method calls are virtually bound.

Combined with class inheritance, virtual binding allows programming with the so-called template method pattern [16]: a base class provides architecture and subclasses provide the specialized (auxiliary) methods, while code reuse is supported for the architecture. The mechanism can be illustrated by an object of class *D* which executes a method defined in its superclass *A* (cf. Fig. 1a) and this method makes a call to a method *m*. With virtual binding, the code selected for execution will be associated to the first matching signature for *m above D*; i.e., the method in *B* is selected. However it is unsettled how to virtually bind method invocations in a class hierarchy with multiple inheritance, if methods are defined in different classes in the hierarchy. In the example of Fig. 1b, a strategy is needed to clearly express which method definition to select among the candidates for *m*.

Formal models of possible solutions to multiple inheritance may contribute to better understanding and use of multiple inheritance, and facilitate reasoning about code inheritance. A denotational account of multiple inheritance has been given [6], but virtual binding is not considered as name conflicts are assumed not to occur.

An operational semantics in rewriting logic allows executable experimentation with different strategies for virtual binding. For this purpose, we consider multiple inheritance in the setting of the Creol language, which has a complete formalization in rewriting logic. In previous work [22], an ordered solution was proposed in which the binding strategy did not distinguish a virtual call from a superclass (*A* in Fig. 1a) and a standard call from the subclass (*C* in Fig. 1a). In this paper a novel version of the

ordered approach is considered in which the order may vary between calls, as the ordering is dynamically decided by the context of each call. This new approach, called *pruned binding*, avoids renaming while providing better support for the encapsulation principle. Calls are always bound to specializations of the definition found by static analysis, allowing reasoning reuse for virtual calls in the setting of multiple inheritance.

Consider the case where *D* inherits two unrelated classes *B* and *C* (Fig. 1d), both with a method *m*. Assume that a *D* object calls a method in *C* which in turn calls *m* locally. With the ordered approach this call will bind to the *m* of *B* rather than that of *C*, assuming no redefinition of *m* in *D*. This binding is clearly undesirable since the *m* of *B* is not a redefinition of that of *C*. The two *m* methods have no relationship since they are from unrelated class hierarchies. The example in Sect. 5 demonstrates resulting problems. These problems are avoided with the pruned binding strategy. Furthermore, the strategy ensures the principle that when the actual class of an object is smaller, each local call will be bound to a smaller class. This principle is intuitive and is also useful for reasoning control.

3 A Language for Asynchronously Communicating Objects

This section provides a basis for the technical discussion which follows. We consider a small object-oriented language which is a subset of Creol [20,22], a high-level language for distributed concurrent objects. We distinguish data, typed by data types, and objects, typed by interfaces. The language allows both blocking and nonblocking method calls, based on a uniform semantics. Attributes (instance variables) and method declarations are organized in classes, which may have data and object parameters. Objects are concurrent and have their own processor which evaluates local processes. A process consists of program code with *processor release points* together with a local state, representing remaining parts of method activations. Processes may be *active*, reflecting autonomous behavior initiated at creation time by the *run* method, or *reactive*; i.e., in response to method invocations. Due to processor release points, the evaluation of processes may be interleaved. The values of an object's program variables may depend on the nondeterministic interleaving of processes. However, a method activation may have local variables supplementing the object variables, in particular the values of formal parameters are stored locally. An object may contain several (pending) activations of the same method, possibly with different values for local variables.

Guards *b* in statements **await** *b* explicitly declare potential processor release points. When a guard which evaluates to false is encountered during process evaluation, the process is *suspended* and the processor released. After processor release, any enabled pending process may be selected for evaluation. For the examples of this paper, it suffices to consider guards as boolean expressions over program variables, but we introduce reply guards in the operational semantics (cf. Sect. 6.5).

Statements can be composed sequentially or by conditional branching. Let S_1 and S_2 denote statement lists. Sequential composition may introduce inner guards: **await** *b* is a potential release point in S_1;**await** $b;\mathrm{S}_2$. Assignment to local and object variables is expressed as $\mathrm{V} := \mathrm{E}$ for a disjoint list of program variables V and an expression list E,

Syntactic categories.		*Definitions.*
s in Stm	v in Var	$p ::= m \mid x.m \mid m@\mathit{classname} \mid m < \mathit{classname}$
t in Label	e in Expr	$\mathrm{S} ::= s \mid s;\mathrm{S}$
m in Mtd	x in ObjExpr	$s ::= \textbf{skip} \mid (\mathrm{S}) \mid \mathrm{V} := \mathrm{E} \mid v := \textbf{new}\ \mathit{classname}(\mathrm{E})$
p in MtdCall	b in BoolExpr	$\mid p(\mathrm{E};\mathrm{V}) \mid \textbf{await}\ p(\mathrm{E};\mathrm{V}) \mid \textbf{await}\ b \mid \textbf{if}\ b\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2\ \textbf{fi}$

Figure 2. A subset of the Creol language for method definitions, with typical terms for each category. Capitalized terms such as E denote lists, sets, or multisets of the given syntactic categories.

of matching types. The reserved word *self* is used for self reference. In-parameters as well as the *self* and *caller* pseudo-variables are read-only variables.

All object interaction happens through method calls. We consider here blocking calls and nonblocking calls. (The full language provides more expressiveness [20].) A *nonblocking method call* is written **await** $x.m(\mathrm{E};\mathrm{V})$. The calling process emits the call to an object x and suspends itself while waiting for a reply. When the reply arrives, return values are assigned to V and evaluation continues.

A *blocking method call*, immediately blocking the processor while waiting for a reply, is written $x.m(\mathrm{E};\mathrm{V})$. When x evaluates to *self*, the call is said to be local. The language does not support monitor reentrance (except for calls to self), mutual blocking calls may therefore lead to deadlock. In order to evaluate local blocking calls, the evaluation of the call will precede the continuation of the active process, thereby unblocking the processor (self-reentrance).

Internal calls are not prefixed by an object identifier and are identified syntactically, otherwise the call is external. All calls are virtually bound, except when the method name is explicitly qualified by a class name, $m@C$. In our setting method calls can always be emitted, as a receiving object cannot block communication. *Method overtaking* is allowed: if methods offered by an object are invoked in one order, the object may start execution of the method activations in another order.

With nonblocking method calls, the object will not block while waiting for replies. This approach allows flexibility in the distributed setting: suspended processes or new method calls may be evaluated while waiting. If the called object never replies, deadlock is avoided as other activity in the object is possible. However, when the reply arrives, the *continuation* of the process must compete with other pending and enabled processes.

4 Multiple Inheritance

A mechanism for multiple inheritance is now considered, where all attributes and methods of a superclass are inherited by the subclass, and where superclass methods may be redefined. In the syntax, the keyword **inherits** is introduced followed by an *inheritance list*; i.e., a list of class names $C(\mathrm{E})$ where E provides the actual class parameters.

Let a class hierarchy be a directed acyclic graph of parameterized classes. Each class consists of a list of inherited classes, a set of attributes (program variables including class parameters), and method definitions. The encapsulation provided by interfaces

suggests that external calls to an object of class C are virtually bound to the closest method definition above C. However, the object may internally invoke methods of its superclasses. In the setting of multiple inheritance and overloading, methods defined in a superclass may be accessed from the subclass by qualified references. Vertical name conflicts for method names are resolved in a standard way: the first matching definition with respect to the types of the actual parameters is chosen while ascending a branch of the inheritance tree. Horizontal name conflicts will be resolved dynamically depending on the class of the object and the context of the call.

4.1 Qualified Names

Qualified names may be used to uniquely refer to an attribute or method in a class. For this purpose, we adapt the **qua** construct of Simula to the setting of multiple inheritance. For an attribute x or a method m declared in a class C, we denote by $x@C$ and $m@C$ the qualified names which provide static references to x and m. By extension, if x or m is *not* declared in C, but inherited from the superclasses of C, the qualified reference $m@C$ binds as an unqualified reference m from C.

Attribute names are not visible through an object's external interfaces. Consequently attribute names should not be merged if inheritance leads to name conflicts, and attributes of the same name should be allowed in different classes of the inheritance hierarchy [36]. In order to allow the reuse of attribute names, these will always be expanded into qualified names. This is desirable in order to avoid run-time errors that may occur if methods of superclasses assign to overloaded attributes. This qualification convention has the following consequence: unlike C++, there is no duplication of attributes when branches in the inheritance graph have a common superclass. Consequently if multiple copies of the superclass attributes are needed, one has to rely on delegation techniques.

Instantiation of attributes. At object creation time, attributes are collected from the object's class and superclasses. An attribute in a class C is declared by **var** $x : T = e$, where x is the name of the attribute, T its type, and e its initial value. The expression e may refer to the values of the class parameter variables $\mathtt{V}$, as well as to the values of inherited attributes by means of qualified references. The initial state values of an object of class C then depend on the actual parameter values bound to $\mathtt{V}$. These may be passed as parameter values to inherited classes in order to derive values for the inherited attributes, which in turn may be used to instantiate the locally declared attributes.

Accessing inherited attributes and methods. If C is a superclass of C', we introduce the syntax **await** $m@C(\mathtt{E};\mathtt{V})$ for nonblocking, and $m@C(\mathtt{E};\mathtt{V})$ for blocking, internal calls to a method above C in the inheritance graph. These calls may be bound without knowing the exact class of the *self* object, so they are called *static*, in contrast to calls without @, called *virtual*. We assume that attributes have unique names in the inheritance graph; this may be enforced at compile time by extending each attribute name x with the name of the class in which it is declared, which implies that attributes are bound statically. Consequently, a method declared in a class C may only access attributes declared above C. In a subclass, an attribute x of a superclass C is accessed by the qualified reference $x@C$. This means that multiply inherited superclasses are shared, rather than duplicated.

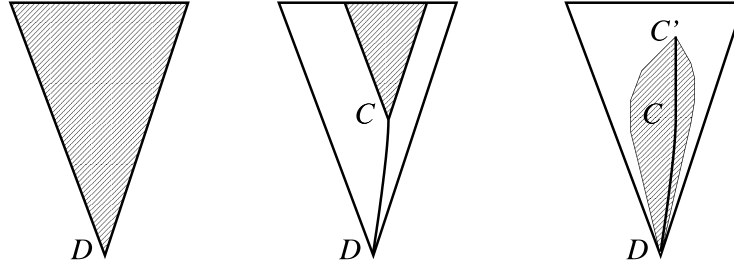

Figure 3. Binding calls to m, $m@C$, and $m < C'$ from class D

Duplication may be achieved by class renaming in an inheritance list. The language syntax is summarized in Fig. 2.

4.2 Virtual Binding

Let the nominal subtype relation $\prec$ be a reflexive partial ordering on types, including interfaces. A data type may only be a subtype of a data type and an interface only of an interface. If $T \prec T'$ then any value of T may masquerade as a value of T'. For product types R and R', $R \prec R'$ is the point-wise extension of the subtype relation; i.e., R and R' have the same length l and $T_i \prec T_i'$ for every i ($0 \leq i \leq l$) and types T_i and T_i' in position i in R and R' respectively. To explain the typing and binding of methods, $\prec$ is extended to function spaces $A \rightarrow B$, where A and B are (possibly empty) product types:

$$A \rightarrow B \prec A' \rightarrow B' = A \prec A' \wedge B' \prec B$$

expressing the relationship between actual and formal parameters, but not subtyping over function spaces, which are not part of the functional language. The static analysis of an internal call $m(\mathtt{E};\mathtt{V})$ will assign unique types to the in and out parameter depending on the textual context, say that the parameters are textually declared as $\mathtt{E} : T_{\mathrm{E}}$ and $\mathtt{V} : T_{\mathrm{V}}$. The call is *type correct* if there is a method declaration $m : A \rightarrow B$ in the class C, possibly inherited, such that $T_{\mathrm{E}} \rightarrow T_{\mathrm{V}} \prec A \rightarrow B$. The binding of an internal nonblocking call **await** $m(\mathtt{E};\mathtt{V})$ is handled as the corresponding blocking call $m(\mathtt{E};\mathtt{V})$. An external call to an object of interface I is type correct if it can be bound to a method declaration in I in a similar way. The static analysis of a class will verify that it implements the methods declared in its interfaces.

Let a class C be *below* a class C' if C is C', or is a direct or indirect subclass of C'. Similarly, a method declaration inside a class C is *below* a class C' if C is below C'. We introduce the syntax $m < C'$ for *constrained method calls*, restricting the virtual binding of m to methods below C'. (Static typing requires the class enclosing the call to be below C'.) The pruned virtual binding of method calls is now explained. (The formalization is given in Sec. 6.4.) At run-time, a call to a method of an object o will always be bound above the class of o. Let m be a method declared in an interface I and let o be an instance of a class C implementing I. There are two cases:

1. m is called externally, in which case C is not statically known. In this case, C is dynamically identified as the class of o.
2. m is called internally from C', a superclass of the actual class C of o. In this case static analysis will identify the call with a declaration of m above C', say in C''. Consequently, we let the call be constrained by C'', and compilation replaces the reference to m with a reference to $m < C''$.

The dynamically decided context of a call may eliminate parts of the inheritance graph above the actual class of the callee with respect to the binding of a specific call. If a method name is ambiguous within the dynamic constraint, we assume that any solution is acceptable. For a natural and simple model of priority, the call will be bound to the first matching method definition above C, in a left-first depth-first order. (An arbitrary order may be obtained by replacing the inheritance list by a multiset.)

It is easy to see that run-time binding always succeeds in any well-typed program. When a method $m : T_E \rightarrow T_V$ in an object o of interface I is externally called at run-time, the actual class C of o is dynamically decided and the virtual binding mechanism will bind to a declaration $m : A \rightarrow B$ such that $T_E \rightarrow T_V \prec A \rightarrow B$, taking the first such m when traversing the inheritance graph above C. Static analysis guarantees that C implements I and consequently that at least one method declaration of m above C may be bound to the call. An internal call $m : T_E \rightarrow T_V$ is made by an object of a subclass D of C (from the static analysis) and the virtual binding mechanism will bind to a declaration of $m : A' \rightarrow B'$ such that $T_E \rightarrow T_V \prec A' \rightarrow B'$, following the binding strategy constrained by D. Because C is inherited by D, the virtual binding is guaranteed to succeed. However, it is not guaranteed that the declaration above C which was found by static analysis will be selected. In order to ensure that a call to m in D will choose the declaration above C, the method may be qualified as $m@C$ in D. For virtual calls from a superclass C' of C, such qualification cannot be used. In order to ensure that a virtually bound call from a superclass will select a specialization of the statically found declaration, the binding will be constrained by C'. Even if no specialization is found, the binding will succeed as the constraint does not remove the declaration found by static analysis.

5 Example: Combining Authorization Policies

In a database containing sensitive information and different authorization policies, the information returned for a request will depend on the clearance level of the agent making the request. Let Any denote the interface of arbitrary objects, Agent the interface of agents, and Auth an authorization interface with methods *grant*(x), *revoke*(x), *auth*(x), and *delay* for agents x. The two classes *SAuth* and *MAuth*, both implementing Auth, implement single and multiple authorization policies, respectively. Since the attribute *gr* in *SAuth* is implemented as an object identifier, *SAuth* only authorizes one agent at a time whereas *MAuth* authorizes multiple agents. The method *grant*(x) returns when x becomes authorized, and authorization is removed by *revoke*(x). The method *auth*(x) suspends until x is authorized, and *delay* returns once no agent is authorized.

```
class SAuth implements Auth
begin with Any
  var gr: Agent= null
  op grant(in x:Agent) == delay; gr := x
  op revoke(in x:Agent) ==
      if gr = x then gr := null fi
  op auth(in x:Agent) == await (gr = x)
  op delay == await (gr = null)
end
```

```
class MAuth implements Auth
begin with Any
  var gr: Set[Agent] = ∅
  op grant(in x:Agent) == gr := gr ∪ {x}
  op revoke(in x:Agent) == gr := gr \ {x}
  op auth(in x:Agent) == await (x ∈ gr)
  op delay == await (gr = ∅)
end
```

Authorization levels. *Low clearance* agents may share access to unclassified data while *high clearance* agents have unique access to (classified) data. Proper usage is defined by two interfaces, defining open and close operations at both access levels:

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

```
interface Low
begin with Agent
  op openL
  op access(in k:Key; out y:Data)
  op closeL
end
```

When the *openH* method returns, the calling agent would not know whether high access was granted, unless a boolean out parameter is present.

Let a class *DB* provide the actual operations on the database. We assume given the operations *access*(**in** k:Key, high:Bool; **out** y:Data), where *high* defines the access level, and *clear*(**in** x : Agent; **out** b : Bool) to give clearance to sensitive data for agent x. Any agent may get low access rights, while only agents cleared by the database may be granted exclusive high access. The class *MAuth* will authorize low clearance, and *SAuth* will authorize high clearance. *SAuth* authorizes only one agent at a time.

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

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

The code given here uses nonblocking calls whenever there is a possibility of local deadlock. Thus, objects of the four classes above will be able to respond to new requests even when used improperly, for instance when agent access is not initiated by open. Notice that the *caller* pseudo-variable is used to pass on agent identity in local calls. The **with** Agent clauses imply that Agent is the type of *caller*, ensuring strong typing.

The database itself has no interface containing *access*, therefore all database access is through the High and Low interfaces. Notice also that objects of the *LAuth* and *HAuth* classes may not be used through the Auth interface. This would have been harmful for the authorization provided in the example. For instance, a call to *grant* to a *HAuth* object could then result in high *access* without clearance of the calling agent! This supports the approach not to inherit implementation clauses.

Combining authorization levels. High and low authorization policies may be combined in a subclass *HLAuth* which implements both interfaces, inheriting *LAuth* and *HAuth*.

class *HLAuth* **implements** High, Low
  inherits *LAuth*, *HAuth*
begin with Agent
  op access(**in** k:Key; **out** y:Data) == **if** caller=gr@*SAuth*
    then access@*HAuth*(k; y) **else** access@*LAuth*(k; y) **fi**
end

Notice that the same database is used for both High and Low interaction. Although the *DB* class is inherited twice, *HLAuth* gets only one copy (cf. Sect. 4.1).

The example demonstrates natural usage of classes and multiple inheritance. Nevertheless, it reveals problems with the combination of inheritance and *statically ordered* virtual binding: Objects of the classes *LAuth* and *HAuth* work well, in the sense that agents opening access through the Low and High interfaces get the appropriate access. However the addition of the common subclass *HLAuth* is detrimental, assuming a fixed inheritance ordering: When used through the High interface, this class would allow *multiple high access* to data! Calls to the High operations of *HLAuth* will trigger calls to the *HAuth* methods. From these methods the virtual internal calls to *grant*, *revoke*, and *auth* will now bind to those of the *MAuth* class, if selected in a left-most depth-first traversal of the inheritance tree of the actual class *HLAuth*. Note that if the inheritance ordering in *HLAuth* were reversed, similar problems occur with the binding of Low interaction.

The *pruned* binding strategy proposed in this paper ensures that the virtual internal calls inside classes *HAuth* and *LAuth* will be bound in classes *SAuth* and *MAuth*, respectively, regardless of the actual class of the caller (*HAuth*, *LAuth*, or *HLAuth*) and of the inheritance ordering in *HLAuth*. In particular the *grant* call inside *HAuth* will be understood as $grant < SAuth$, which may not bind to *grant* of *MAuth*.

6 An Operational Semantics of Inheritance and Virtual Binding

The operational semantics is defined using rewriting logic [29]. A rewrite theory is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$, where the signature Σ defines the function symbols of the language, E defines equations between terms, L is a set of labels, and R is a set of labeled rewrite rules. From a computational viewpoint, a rewrite rule $t \longrightarrow t'$ may be interpreted as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t'. Each rewrite rule describes how a part of a configuration can evolve in one transition step. If rewrite rules may be applied to

non-overlapping subconfigurations, the transitions may be performed in parallel. Consequently, concurrency is implicit in rewriting logic (RL). A number of concurrency models have been successfully represented in RL [8, 29], including Petri nets, CCS, Actors, and Unity, as well as the ODP computational model [33]. RL also offers its own model of object orientation [8].

Informally, a state configuration in RL is a multiset of terms of given types. Types are specified in (membership) equational logic (Σ, E), the functional sublanguage of RL which supports algebraic specification in the OBJ [17] style. When modeling computational systems, configurations may include the local system states. Different parts of the system are modeled by terms of the different types defined in the equational logic.

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations which define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the defining equations of E. Conditional rewrite rules are allowed, where the condition is formulated as a conjunction of rewrites and equations which must hold for the main rule to apply:

$$\textit{subconfiguration} \longrightarrow \textit{subconfiguration}\ \textbf{\textit{if}}\ \textit{condition}.$$

Rules in RL may be formulated at a high level of abstraction, closely resembling a compositional operational semantics. In fact, structural operational semantics can be uniformly mapped into RL specifications [30].

6.1 System Configurations

A method call will be reflected by a pair of messages, and object activity will be organized around a *message queue* which contains incoming messages and a *process queue* which contains suspended processes, i.e. remaining parts of method activations. Messages have the general form *message* **to** *dest* where *dest* is a single object or class, or a list of classes. A state configuration is a multiset combining Creol objects, classes, and messages. (In order to increase the parallelism in the model, message queues could be external to object bodies as shown in [20, 22].) As usual in RL, the associative constructor for lists, as well as the associative and commutative constructor for multisets, are represented by whitespace.

In RL, objects are commonly represented by terms of the type $\langle o : C \,|\, a_1 : v_1, \ldots, a_n : v_n \rangle$ where o is the object's identifier, C is its class, the a_i's are the names of the object's attributes, and the v_i's are the corresponding values [8]. We adopt this form of presentation and define Creol objects and classes as RL objects. Omitting RL types, a Creol object is represented by an RL object $\langle Ob \mid Cl, Pr, PrQ, Lvar, Att, Lab, EvQ \rangle$, where Ob is the object identifier, Cl the class name, Pr the active process code, PrQ a multiset of suspended processes with unspecified queue ordering, EvQ a multiset of unprocessed messages, and $Lvar$ and Att the local and object state, respectively. Let τ be a type partially ordered by $<$, with least element 1, and let $next : \tau \rightarrow \tau$ be such that $\forall x . x < next(x)$. Lab is the method call identifier corresponding to labels in the language, of type τ. Thus, the object identifier Ob and the generated local label value provide a globally unique identifier for each method call.

The classes of Creol are represented by RL objects $\langle Cl \,|\, Inh, Att, Mtds, \ Tok\rangle$, where *Cl* is the class name, *Inh* is the inheritance list, *Att* a list of attributes, *Mtds* a multiset of methods, and *Tok* is an arbitrary term of type τ. When an object needs a method, it is bound to a definition in the *Mtds* multiset of its class or of a superclass.

In RL's object model [8], classes are not represented explicitly in the system configuration. This leads to ad hoc mechanisms to handle object creation, which we avoid by explicit class representation. The Creol construct *new* $C(\mathrm{E})$ creates a new object with a unique object identifier, attributes as listed in the class parameter list and in *Att*, and places the code from the *run* method in *Pr*.

6.2 Concurrent Transitions

Concurrent change is achieved in the operational semantics by applying concurrent rewrite steps to state configurations. There are four different kinds of rewrite rules:

– *Rules that execute code from the active process:* For every program statement there is at least one rule. For example, the assignment rule for the program $\mathrm{V} := \mathrm{E}$ binds the values of the expression list E to the list V of local and object variables.
– *Rules for suspension of the active process:* When an active process guard evaluates to false, the process and its local variables are suspended, leaving *Pr* empty.
– *Rules that activate suspended processes:* When *Pr* is empty, suspended processes may be activated. When this happens, the local state is replaced.
– *Transport rules:* These rules move messages into the message queues, representing network flow.

When auxiliary functions are needed in the semantics, these are defined in equational logic, and are evaluated in between the state transitions [29]. The rules related to method calls, virtual binding, and object creation are now considered in detail. In the presentation irrelevant attributes are ignored in the style of Full Maude [8].

6.3 Method Calls

Blocking and nonblocking calls are given a uniform semantics. In the operational semantics, objects communicate by sending messages. Two messages encode a method call. We here assume that the types of the actual in- and out-parameters of the call have been added to the method invocation as an additional argument *Sig* at compile time. If an object o_1 calls a method m of an object o_2, with actual type *Sig* and actual parameters *In*, and the execution of $m(Sig, In)$ results in the return values *Out*, the call is reflected by two messages $invoc(m, Sig, (n\ o_1\ In))$ **to** o_2 and $comp(n, Out)$ **to** o_1, which represent the invocation and completion of the call, respectively. In the asynchronous setting, invocation messages must include the caller's identity, so completions can be transmitted to the correct destination. Objects may have several pending calls to another object, so the completion message includes a locally unique label value n, generated by the caller.

A blocking call $p(Sig, In; \mathrm{V})$, where V is a list of variables and p one of the forms $x.m$, $m@C$, or $m < C$, is translated into an *asynchronous call*, $!p(Sig, In)$, immediately followed by a blocking *reply statement*, $n?(\mathrm{V})$, where n is the label value uniquely identifying the call:

$$\langle o : Ob \mid Pr : p(Sig, In; \textsc{v}); \textsc{s}, Lab : n\rangle = \langle o : Ob \mid Pr : !p(Sig, In); n?(\textsc{v}); \textsc{s}, Lab : n\rangle$$

A nonblocking call is understood as an asynchronous call followed by a *reply guard*:

$$\langle o : Ob \mid Pr : \textbf{await}\ p(Sig, In; \textsc{v}); \textsc{s}, Lab : n\rangle$$
$$= \langle o : Ob \mid Pr : !p(Sig, In); \textbf{await}\ n?; n?(\textsc{v}); \textsc{s}, Lab : n\rangle$$

A reply guard **await** $n?$ evaluates to true when a *comp* message with the label value n has arrived, in which case the reply statement $n?(\textsc{v})$ will assign the return values to $\textsc{v}$, otherwise the active process is suspended (see below). Consequently, it suffices to consider asynchronous invocations, and blocking and guarded replies to capture both blocking and nonblocking method calls.

When an object calls an external method, a message is placed in the configuration:

$$\langle o : Ob \mid Pr : !x.m(Sig, In); \textsc{s}, Lvar : \textsc{l}, Att : \textsc{a}, Lab : n\rangle$$
$$\longrightarrow$$
$$\langle o : Ob \mid Pr : \textsc{s}, Lvar : \textsc{l}, Att : \textsc{a}, Lab : next(n)\rangle$$
$$invoc(m, Sig, (n\ o\ eval(In, (\textsc{a}; \textsc{l})))) \ \textbf{to}\ eval(x, (\textsc{a}; \textsc{l}))$$

where x is an object expression, m a method name, and *eval* is a function which evaluates an expression (list) in the context of a state. When x evaluates to o, the object creates an *invoc* message to itself. Similarly, an internal call gives rise to the same invocation message:

$$\langle o : Ob \mid Pr : !p(Sig, In); \textsc{s}, Lvar : \textsc{l}, Att : \textsc{a}, Lab : n\rangle$$
$$\longrightarrow$$
$$\langle o : Ob \mid Pr : \textsc{s}, Lvar : \textsc{l}, Att : \textsc{a}, Lab : next(n)\rangle$$
$$invoc(p, Sig, (n\ o\ eval(In, (\textsc{a}; \textsc{l})))) \ \textbf{to}\ o$$

where p is of the form $m@C$ or $m < C$. The constraint C will be used in the virtual binding as described below.

Transport rules take charge of messages, which eventually arrive at the destination's message queue:

$$(invoc(\textsc{e})\ \textbf{to}\ o)\ \langle o : Ob \mid EvQ : \textsc{q}\rangle \longrightarrow \langle o : Ob \mid EvQ : \textsc{q}\ invoc(\textsc{e})\rangle$$
$$(comp(\textsc{e})\ \textbf{to}\ o)\ \langle o : Ob \mid EvQ : \textsc{q}\rangle \longrightarrow \langle o : Ob \mid EvQ : \textsc{q}\ comp(\textsc{e})\rangle$$

These rules model loose distribution of objects. Message overtaking is captured by the nondeterminism inherent in RL: messages sent by an object to another object in one order may arrive in any order.

The caller may wait for a completion in a reply statement to synchronize on the completion of the call, or in a reply guard. The reply statement $n?(\textsc{v})$ blocks until the appropriate reply message has arrived in the message queue. This blocking is captured by a rule requiring matching label values in the active statement and the event queue:

$$\langle o : Ob \mid Pr : (n?(\textsc{v}); \textsc{s}), EvQ : \textsc{q}\ comp(n, Out)\rangle$$
$$\longrightarrow\ \langle o : Ob \mid Pr : (\textsc{v} := Out; \textsc{s}), EvQ : \textsc{q}\rangle$$

In the model, *EvQ* is a multiset; thus the rule will match any occurrence of $comp(n, Out)$ in the queue. Remark that blocking reply statements associated with calls to self require special treatment in order to avoid deadlock [20].

6.4 Virtual and Static Binding of Method Calls

In order to allow concurrent and dynamic execution, the full inheritance graph will not be statically given. Rather, the binding mechanism dynamically inspects the current class hierarchy as present in the configuration. Our approach to virtual binding is to use a *bind* message to be sent from a class to its superclasses, resulting in a *bound* message returned to the object generating the *bind* message. This way, the inheritance graph is explored dynamically and as far as necessary when needed. When the invocation of a method m is found in the message queue of an object o, a message $bind(o,m,In)$ **to** C can be generated by dynamically retrieving the class C of the object. Here Sig is the method signature as provided by the caller and In is the list of actual in-parameters:

$$\begin{array}{l}\langle o : Ob \mid Cl : C, EvQ : invoc(m, Sig, In)\ \textsc{q}\rangle \\ \quad\longrightarrow \langle o : Ob \mid Cl : C, EvQ : \textsc{q}\rangle\ (bind(m, Sig, In, o)\ \textbf{to}\ C)\end{array}$$

The same applies to internal static calls $m@C$. Static method calls are generated without inspecting the *actual* class of the callee, thus surpassing local definitions:

$$\langle o : Ob \mid EvQ : invoc(m@C, Sig, In)\ \textsc{q}\rangle \longrightarrow \langle o : Ob \mid EvQ : \textsc{q}\rangle\ (bind(m, Sig, In, o)\ \textbf{to}\ C)$$

If a suitable m is defined locally in C, a process with the method code and local state is returned in a *bound* message. Otherwise, the *bind* message is retransmitted to the superclasses of C in a left-first, depth-first order. In order to easily traverse the inheritance graph, an inheritance list is used as the destination of the *bind* message:

$$\begin{array}{l}(bind(m, Sig, In, o)\ \textbf{to}\ C\ \textsc{i})\ \langle C : Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle \\ \quad\longrightarrow\ \textbf{if}\ match(m, Sig, \textsc{m})\ \textbf{then}\ bound(get(m, \textsc{m}, In))\ \textbf{to}\ o \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ bind(m, Sig, In, o)\ \textbf{to}\ (\textsc{i}'\ \textsc{i})\ \textbf{fi}\ \langle C : Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle\end{array}$$

The auxiliary predicate $match(m, Sig, \textsc{m})$ evaluates to true if m is declared in $\textsc{m}$ with a signature Sig' such that $Sig \prec Sig'$, and the function *get* returns a process with the method's code and local state from the method multiset $\textsc{m}$ of the class. (Static checking ensures that virtual binding will succeed.) Values of the actual in-parameters In, the caller o', and the label value n are stored locally. The process w resulting from the binding is loaded into the internal process queue:

$$(bound(w)\ \textbf{to}\ o)\ \langle o : Ob \mid PrQ : \textsc{w}\rangle \longrightarrow \langle o : Ob \mid PrQ : \textsc{w}\ w\rangle$$

Note that the use of rewrite rules rather than equations mimics distributed and concurrent processing of method lookup.

Internal virtual binding. The binding of an internal virtual call $m < C'$ constrained by C' is slightly more complex. When a match in a class C is found, the inheritance graph of C is checked to ensure that C is below C', otherwise the binding must resume:

$$\begin{array}{l}(bind(m < C', Sig, In, o)\ \textbf{to}\ C\ \textsc{i})\ \langle C : Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle \\ \quad\longrightarrow\ \textbf{if}\ match(m, Sig, \textsc{m})\ \textbf{then}\ (find(C', C)\ \textbf{to}\ C)\ (stopbind(m < C', Sig, In, o)\ \textbf{to}\ C\ \textsc{i}) \\ \qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ bind(m, Sig, In, o)\ \textbf{to}\ (\textsc{i}'\ \textsc{i})\ \textbf{fi}\ \langle C : Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle\end{array}$$

$$\begin{array}{l}(found(b, C')\ \textbf{to}\ C)\ (stopbind(m, Sig, In, o)\ \textbf{to}\ C\ \textsc{i})\ \langle C : Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle \\ \quad\longrightarrow\ \textbf{if}\ b\ \textbf{then}\ bound(get(m, \textsc{m}, In))\ \textbf{to}\ o\ \textbf{else}\ bind(m, Sig, In, o)\ \textbf{to}\ \textsc{i}\ \textbf{fi} \\ \qquad\quad \langle C : Cl \mid Inh : \textsc{i}', Mtds : \textsc{m}\rangle\end{array}$$

where *stopbind* is an additional message used to suspend binding while checking that C is below C'. This is done by two auxiliary messages: The message $\mathit{find}(C,o)$ **to** I represents that o is asking I if C is found in I or further up in the hierarchy, whereas $\mathit{found}(b,C)$ **to** o gives the answer to o, where the boolean b is true if the request was successful. This can be formalized by the rewrite rules (ignoring class parameter lists)

$\mathit{find}(C,o)$ **to** $\varepsilon \longrightarrow \mathit{found}(\mathit{false},C)$ **to** o

$\mathit{find}(C,o)$ **to** $\mathrm{I}\ C\ \mathrm{I}' \longrightarrow \mathit{found}(\mathit{true},C)$ **to** o

$(\mathit{find}(C,o)$ **to** $C'\ \mathrm{I})\ \langle C' : Cl \,|\, Inh : \mathrm{I}'\rangle \longrightarrow (\mathit{find}(C,o)$ **to** $\mathrm{I}\ \mathrm{I}')\ \langle C' : Cl \,|\, Inh : \mathrm{I}'\rangle$ **if** $(C \neq C')$

This search corresponds to breadth-first, left-first traversal of the inheritance graph.

6.5 Guarded Statements

Guards represent potential processor release points. Guards may be boolean or reply guards. When a guard is encountered, the execution continues if the guard is enabled:

$\langle o : Ob \,|\, Pr : \mathit{await}\ g; \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q}\rangle$
$\longrightarrow$
$\langle o : Ob \,|\, Pr : \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q}\rangle$ **if** $\mathit{enabled}(g,(\mathrm{A},\mathrm{L}),\mathrm{Q})$

Enabledness is defined by induction over the construction of guards by the predicate

$$\mathit{enabled}(n?,\mathrm{D},\mathrm{Q}) = n\ \mathit{in}\ \mathrm{Q} \qquad \mathit{enabled}(b,\mathrm{D},\mathrm{Q}) = \mathit{eval}(b,\mathrm{D})$$

where D denotes a state, and the function *in* checks whether a completion message corresponding to the given label value is in the message queue Q. Enabledness is extended to statement lists, considering the head statement, and considering unguarded statements as enabled. When a non-enabled guard is encountered, the active process is *suspended* on the process queue:

$\langle o : Ob \,|\, Pr : \mathrm{S}, PrQ : \mathrm{W}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q}\rangle$
$\longrightarrow$
$\langle o : Ob \,|\, Pr : \varepsilon, PrQ : (\mathrm{W}\ \langle \mathrm{S},\mathrm{L}\rangle), Lvar : \varepsilon, Att : \mathrm{A}, EvQ : \mathrm{Q}\rangle$ **if** $\mathit{not}\ \mathit{enabled}(\mathrm{S},(\mathrm{A};\mathrm{L}),\mathrm{Q})$

where $\langle \mathrm{S},\mathrm{L}\rangle$ denotes the process with statements S and local state L. If there is no active process, a suspended process can be *reactivated* if it is enabled:

$\langle o : Ob \,|\, Pr : \varepsilon, PrQ : \langle \mathrm{S},\mathrm{L}\rangle\ \mathrm{W}, Lvar : \mathrm{L}', Att : \mathrm{A}, EvQ : \mathrm{Q}\rangle$
$\longrightarrow$
$\langle o : Ob \,|\, Pr : \mathrm{S}, PrQ : \mathrm{W}, Lvar : \mathrm{L}, Att : \mathrm{A}, EvQ : \mathrm{Q}\rangle$ **if** $\mathit{enabled}(\mathrm{S},(\mathrm{A},\mathrm{L}),\mathrm{Q})$

This rule allows any enabled process to continue because *PrQ* is a multiset.

6.6 Object Creation and Attribute Instantiation

Object creation results in a new object with a unique identifier. The new object makes an initial blocking call to its *run* method (if present in the class), thereby initiating active object behavior and leaving the programmer in control of defining the initial release

point. New object identifiers are created by concatenating tokens n from the unbounded set Tok to the class name. The identifier is returned to the object which initiated the object creation.

$$\langle o : Ob \mid Pr : v := new\ C(In); \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}\rangle\ \langle C : Cl \mid Att : \mathrm{A}', Tok : n\rangle$$
$$\longrightarrow$$
$$\langle o : Ob \mid Pr : v := newid; \mathrm{S}, Lvar : \mathrm{L}, Att : \mathrm{A}\rangle\ \langle C : Cl \mid Att : \mathrm{A}', Tok : next(n)\rangle$$
$$\langle newid : Ob \mid Cl : C, Pr : run, PrQ : \varepsilon, Lvar : \varepsilon, Att : \varepsilon, Lab : 1, EvQ : \varepsilon\rangle$$
$$inherit(newid, \varepsilon)\ \mathbf{to}\ C(eval(In, (\mathrm{A}, \mathrm{L})))$$

Here, $newid$ denotes the new identifier. Before the new object can be activated, its initial state must be created. This is done by collecting attribute lists, which consist of program variables bound to initial expressions, from the classes inherited by C. The initial expressions must be reduced to values and bound to the program variables in the state. Class parameters and inherited attributes provide a mechanism to pass values to the initial expressions of the inheritance list in a class. The variables bound by the class parameters are stored first in the attribute list of a class in the textual order.

An $inherit$ message, which sends an object identifier and a substitution to a class inheritance list, causes the inheritance tree to be traversed in a right-first depth-first order, while dynamically accumulating all inherited attributes and their initializing expressions, passing on appropriate class parameters as stated in the inheritance lists. The traversal results in a list of attributes with initializing expressions, which are evaluated by $evalS$ from left to right and delivered to the new object. The attribute list is ordered such that the attributes of a superclass precede those of a subclass, for all classes above the class of the object. Consequently, the type system can guarantee that all variables occurring in an initial expression of a program variable v have been instantiated before v is instantiated.

$$inherit(o, \mathrm{IA})\ \mathbf{to}\ nil = inherited(evalS((self \mapsto o)\ \mathrm{IA}), \varepsilon)\ \mathbf{to}\ o$$

$$inherit(o, \mathrm{IA})\ \mathbf{to}\ (\mathrm{I}\ C(In))\ \ \langle C : Cl \mid Inh : \mathrm{I}', Att : \mathrm{IA}'\rangle$$
$$= inherit(o, (pass(\mathrm{IA}', In)\ \mathrm{IA}))\ \mathbf{to}\ (\mathrm{I}\ \mathrm{I}')\ \ \langle C : Cl \mid Inh : \mathrm{I}', Att : \mathrm{IA}'\rangle$$

The auxiliary function $pass$ passes class parameters, given as expressions, to an attribute list and $evalS(\mathrm{IA}, \mathrm{A})$ evaluates attributes in IA from left to right, given a state A.

$$pass(\mathrm{IA}, \varepsilon) = \mathrm{IA}$$
$$pass(((v \mapsto e)\ \mathrm{IA}), e'\ \mathrm{E}') = (v \mapsto e')\ pass(\mathrm{IA}, \mathrm{E}')$$

$$evalS(\varepsilon, \mathrm{A}) = \varepsilon$$
$$evalS((v \mapsto e)\ \mathrm{IA}, \mathrm{A}) = (v \mapsto eval(e, \mathrm{A}))\ evalS(\mathrm{IA}, (v \mapsto eval(e, \mathrm{A}))\ \mathrm{A})$$

The resulting state is consumed by the new object by the equation

$$(inherited(\mathrm{A})\ \mathbf{to}\ o)\ \ \langle o : Ob \mid Att : \varepsilon\rangle = \langle o : Ob \mid Att : \mathrm{A}\rangle$$

Notice again that the use of equations enables a new object to be created and initialized in a single rewriting step.

In the presence of multiple inheritance, a class C may inherit a superclass several times. The equation

$$\mathrm{A}\ (v \mapsto e)\ \mathrm{A}'\ (v \mapsto e') = \mathrm{A}\ (v \mapsto e)\ \mathrm{A}'$$

on attribute lists ensures that an attribute is only stored once. Thus multi-inheritance of the same class is the same as inheriting the class once, keeping the leftmost instantiation. Duplicate classes may be achieved by class renaming in inheritance lists.

7 Related Work

Formal models clarify the intricacies of object orientation and may thus contribute to better programming languages in the future, making programs easier to understand, maintain, and reason about. Work on object calculi such as the ς-calculus [1] capture object-oriented features such as self-reference, encapsulation, and method calls. Concurrent object calculi [13,18] extend these mechanisms to multithreaded and distributed systems, but the complexities of class inheritance are not addresses in [1,13,18]. A concurrent object calculus with single inheritance is presented by Laneve [26]. Methods of superclasses are accessible and virtual binding is addressed due to a careful renaming discipline. A denotational semantics for single inheritance with similar features is studied by Cook and Palsberg [9]. Multiple inheritance is not addressed in these works.

Formalizations of multiple inheritance in the literature are usually based on the *objects-as-records* paradigm. This approach focuses on subtyping issues related to subclassing, but issues related to method binding are not easily captured. Even access to methods of superclasses is not addressed in Cardelli's denotational semantics of multiple inheritance [6]. Rossi, Friedman, and Wand [34] propose a formal definition of multiple inheritance based on *subobjects*, a run-time data structure used for virtual pointer tables [25, 38]. This formalism focuses on compile time issues and does not clarify multiple inheritance at the abstraction level of the programming language.

Multiple inheritance is supported in languages such as C++ [38], CLOS [12], Eiffel [31], POOL [2], and Self [7]. As discussed in Sect. 2.1, horizontal name conflicts in C++, POOL, and Eiffel are removed by explicit resolution, after which the inheritance graph may be linearized. Multiple dispatch, or multi-methods [12], gives a more powerful binding mechanism, but does not handle the problems considered here, since they appear even for methods without any parameters. Also reasoning about multi-methods is difficult in case of redefinition.

A natural semantics for virtual binding in Eiffel is proposed in [3]. This work is similar in spirit to ours and models the binding mechanism at the abstraction level of the program, capturing Eiffel's renaming mechanism. Mixin-based inheritance [4] and traits [35] also depend upon linearization to be merged correctly into the single inheritance chain. Linearization changes the parent-child relationship between classes in the inheritance hierarchy [36]. Consequently understanding, e.g., method binding quickly becomes difficult.

Full Maude [8] and the Join-calculus [15] model multiple inheritance by disjoint union of methods. Name ambiguity lets method definitions compete for selection. The definition selected when an ambiguously named method is called, may be nondeterministically chosen. Alternatively, programmer control may be improved if inherited classes are ordered [7, 12], resulting in a deterministic binding strategy. However, the ordering

of superclasses may result in surprising but "correct" behavior. The example of Sect. 5 displays such surprising behavior regardless of how the inherited classes are ordered.

The dynamically typed prototype-based language Self [7] proposes an elegant *prioritized binding strategy* to solve this problem, although a formal semantics is not given. The strategy is based on combining ordered and unordered multiple inheritance. Each superclass is annotated with a priority, and many superclasses may have the same priority. A name is only ambiguous if it occurs in two superclasses with the same priority, in which case a class related to the caller class is preferred. However, explicit class priorities may have surprising effects in large class hierarchies: names may become ambiguous through inheritance. If neither class is related to the caller the binding does not succeed, resulting in a method-not-understood error.

The *pruned binding strategy* proposed in this paper solves these issues without the need for manually declaring (equal) class priorities and without the possibility of method-not-understood errors: Calls are only bound to intended method redefinitions. The new binding strategy seems particularly useful during system maintenance to avoid introducing unintentional errors in evolving class hierarchies, as targeted by the Creol language [23]. In particular, we have given an operational semantics based on dynamic and distributed traversal of the available classes, rather than through virtual pointer tables. Our approach may therefore be combined with dynamic constructs for changing the class inheritance structure, such as adding a class C and enriching an existing class with C as a new superclass, which could be useful in open reconfigurable systems.

8 Conclusion

The treatment of ambiguous naming in object-oriented languages with multiple inheritance is unsettled. Disallowing naming ambiguities when inheriting multiple superclasses imposes undesirable restrictions with regard to, e.g., programming flexibility and code maintenance. Ordering inherited classes solves ambiguities by fixing the binding strategy above a given class. However, virtual binding combined with a fixed order may lead to surprising but "correct" effects. This paper has proposed the *pruned binding strategy* to ensure that overriding is intended. This strategy dynamically restricts the ordered inheritance graph depending on the context of the call, using the concept of constrained method call ($m < C$). This construct is also useful for fine grained programmer control of virtual binding in the case of multiple inheritance. The pruned binding strategy and constrained method calls remove unintended effects of ordered inheritance while ensuring that binding will always succeed. The binding strategy is combined with intentional redirection through qualified references and with redefinition in the subclass. In this paper, an operational semantics for the proposed binding strategy has been given in rewriting logic. Although the formalization is given in the setting of Creol, the mechanisms presented here could easily be lifted to another setting.

Acknowledgment. The authors would like to thank Stein Krogdahl for interesting discussions on multiple inheritance and virtual binding. The comments of the FMCO anonymous referees have improved the presentation.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, New York, NY, 1996.
2. P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 161–168. ACM Press, Oct. 1990.
3. I. Attali, D. Caromel, and S. O. Ehmety. A natural semantics for Eiffel dynamic binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, 1996.
4. G. Bracha and W. Cook. Mixin-based inheritance. In N. Meyrowitz, editor, *Proc. of the Conf. on Object-Oriented Programming: Systems, Languages, and Applications / Eur. Conf. on Object-Oriented Programming*, pages 303–311. ACM Press 1990.
5. K. B. Bruce, A. Schuett, R. van Gent, and A. Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Transactions on Programming Languages and Systems*, 25(2):225–290, 2003.
6. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2-3):138–164, 1988.
7. C. Chambers, D. Ungar, B.-W. Chang, and U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symb. Computation*, 4(3):207–222, 1991.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
9. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, Nov. 1994.
10. W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *17th Symp. on Principles of Programming Languages (POPL'90)*, pages 125–135. ACM Press, Jan. 1990.
11. O.-J. Dahl and K. Nygaard. Class and subclass declarations. In J. Buxton, editor, *Simulation Programming Languages*, pages 158–174. North-Holland, 1968. Reprinted in M. Broy and E. Denert, eds., *Software Pioneers — Contributions to Software Engineering*, Springer, 2002.
12. L. G. DeMichiel and R. P. Gabriel. The common lisp object system: An overview. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Eur. Conf. on Object-Oriented Programming (ECOOP'87)*, LNCS 276, pages 151–170. Springer, 1987.
13. P. Di Blasio and K. Fischer. A calculus for concurrent objects. In U. Montanari and V. Sassone, editors, *7th Intl. Conf. on Concurrency Theory (CONCUR'96)*, LNCS 1119, pages 655–670. Springer, Aug. 1996.
14. C. Fischer and H. Wehrheim. Behavioural subtyping relations for object-oriented formalisms. In T. Rus, editor, *8th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST 2000)*, LNCS 1816, pages 469–483. Springer, 2000.
15. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1-2):23–69, 2003.
16. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
17. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, pages 3–167. Kluwer, 2000.
18. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *High-Level Concurrent Languages (HLCL'98)*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.
19. E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proc. 5th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'02)*, pages 45–60. Kluwer, Mar. 2002.

20. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Press, Sept. 2004.
21. E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, LNCS 2635, pages 137–164. Springer, 2004.
22. E. B. Johnsen and O. Owe. Inheritance in the presence of asynchronous method calls. In *Proc. 38th Hawaii Intl. Conf. on System Sciences (HICSS'05)*. IEEE Press, Jan. 2005.
23. E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A dynamic class construct for asynchronous concurrent objects. In *Proc. 7th Intl. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, LNCS 3535, pages 15–30. Springer, June 2005.
24. J. L. Knudsen. Name collision in multiple classification hierarchies. In S. Gjessing and K. Nygaard, editors, *Eur. Conf. on Object-Oriented Programming (ECOOP'88)*, LNCS 322, pages 93–109. Springer, 1988.
25. S. Krogdahl. Multiple inheritance in Simula-like languages. *BIT*, 25(2):318–326, 1985.
26. C. Laneve. Inheritance in concurrent objects. In H. Bowman and J. Derrick, editors, *Formal methods for distributed processing – a survey of object-oriented approaches*, pages 326–353. Cambridge University Press, 2001.
27. B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.
28. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. The MIT Press, Cambridge, Mass., 1993.
29. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
30. J. Meseguer and G. Rosu. Rewriting logic semantics: From language specifications to formal analysis tools. In D. A. Basin and M. Rusinowitch, editors, *Proc. of the 2nd Intl. Joint Conf. on Automated Reasoning (IJCAR 2004)*, LNCS 3097, pages 1–44. Springer, 2004.
31. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, NJ., 1997.
32. G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proc. of the Symp. on Applied Computing*, pages 1267–1274. ACM Press, 2004.
33. E. Najm and J.-B. Stefani. A formal semantics for the ODP computational model. *Computer Networks and ISDN Systems*, 27:1305–1329, 1995.
34. J. G. Rossie Jr., D. P. Friedman, and M. Wand. Modeling subobject-based inheritance. In P. Cointe, editor, *10th Eur. Conf. on Object-Oriented Programming (ECOOP'96)*, LNCS 1098, pages 248–274. Springer, July 1996.
35. N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In L. Cardelli, editor, *Proc. 17th Eur. Conf. on Object-Oriented Programming (ECOOP 2003)*, LNCS 2743, pages 248–274. Springer, 2003.
36. A. Snyder. Inheritance and the development of encapsulated software systems. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, 1987.
37. N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth Intl. Conf. on Software Reuse (ICSR5)*, pages 206–215. IEEE Press, 1998.
38. B. Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–395, Dec. 1989.
39. E. Tempero and R. Biddle. Simulating multiple inheritance in Java. *The Journal of Systems and Software*, 55(1):87–100, Nov. 2000.