Dynamic Reflection for a Statically Typed Language

Søren Brandt and René W. Schmidt

Department of Computer Science
University of Aarhus
DK-8000 Aarhus C, Denmark

Abstract

We present a runtime metalevel interface for BETA. BETA is a compiled and statically typed object-oriented programming language. The metalevel interface preserves the type safe properties of the language and supports static type checking. This is achieved through a novel language construct, the *attribute reference*, on top of which the metalevel interface is built. The metalevel interface is based on a simple conceptual model that reifies a few basic language primitives. For the implementation, a metalevel architecture based on a virtual machine view of the runtime system is introduced. In this model, an open implementation of compiled language is achieved by providing the runtime virtual machine with a metalevel interface supporting runtime reflection.

1 Introduction

Reusability is a main ingredient in most scientific and technical disciplines. Scientific theories are building blocks for a deeper understanding of various systems, well-engineered technical devices serve as building blocks for larger systems, and software components should in general be able to form part of multiple end-user applications. The keywords are modularity, extensibility, and tailorability.

Structured programming turned modularity into the single most important program design principle. Extensibility and tailorability are major reasons for the recent success of object-oriented programming, which allows the programmer to treat new classes as built-ins, and tailor existing classes through subclassing and similar mechanisms.

However, while object-oriented programming languages provide excellent facilities for expressing abstractions in many application domains, most languages fall short in the disciplines of extensibility and tailorability of their own domain.

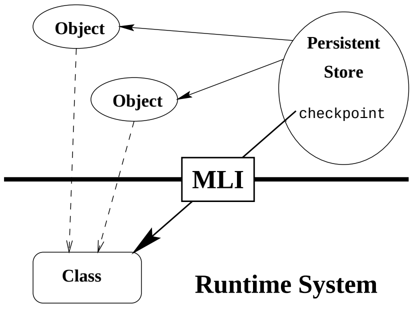

Figure 1: A Persistent Store using runtime metalevel information.

Extension of a programming language with new type-orthogonal functionality such as persistent storage, generic object copying, or distribution is typically not possible. Common for these functionalities are that they all depend on metainformation and possibly implementation details of the language. To circumvent language limitations, the programmer is often forced to use source code preprocessing, special compilers, or new runtime systems.

This paper presents MetaBETA, an open implementation [Kiczales 92] of the BETA language [Madsen et al. 93]. By providing a metalevel interface, MetaBETA extends the BETA language with support for writing abstractions *for* the language directly *in* the language.

The metalevel interface (MLI) is an abstract interface to the implementation of a running program. Through the *introspective* part of the MLI, programs can get an abstract description of the layout of any object. Through the *intercessory* part, basic language primitives such as object invocation, creation, and destruction can be changed. Figure 1 depicts a persistent store utilizing the introspective interface to access type information, thereby being able to serialize arbitrary objects to stable storage. In MetaBETA, the metalevel interface is runtime based, so the persistent store can serialize any object, independent of its class and creator. Access to source code is not necessary.

BETA is a statically typed[1] and compiled language. It is designed for programming-in-the-large, with explicit type annotations to make source code easier to read, maintain, and debug. Thus, adding a MLI raises two major concerns: i) seamless integration of the MLI with the existing type system, and ii) performance retention.

Our approach to handling these concerns is outlined in the next section, where we present the MetaBETA conceptual model and its motivation. Section 3 presents the metalevel interface. Section 4 introduces the metalevel architecture, which is a runtime organization of a compiled program that supports the met-

[1]With statically typed we mean that most type checks are done at compile time. As is the case with, e.g., Eiffel [Meyer 92] and Java [Arnold & Gosling 96], only partial compile-time checking is possible, because subclasses may strengthen the type of inherited attributes. I.e., BETA allows covariant class hierarchies.

alevel interface. Section 5 describes the current status of the project along with preliminary performance results. Section 6 summarizes related work. Finally, Section 7 presents our conclusions.

2 Design and Motivation

Th lack of extensibility and tailorability of the current BETA implementation was recognized during the design and implementation of a number of substrate systems for the Mjølner BETA system, including: type-orthogonal persistence [Brandt 94, Grønbæk et al. 94], distributed object system [Brandt & Madsen 94], an embedded interpreter [Malhotra 93], a source-level debugger, and object- and class-browsers.

These tools manipulate objects in a type-independent manner, and currently depend on the memory layout of objects, as well as other implementation details. Such dependencies severely compromises portability and type-safety. A common trait of these tools is that they all work at the metalevel, using parts of a *running* BETA program as data. Our work on MetaBETA is aimed at identifying the core metalevel functionality needed by these systems, and to support those needs through a type-safe metalevel interface.

Example 1 A type-orthogonal library such as object distribution must be applicable to any class, even if that class was written without distribution in mind, and even if the source code is not available. Hence, the library must be able to handle object in a type generic fashion. □

The design criterias for the metalevel interface was that it should be simple, dynamic, and efficient. By *simple*, we mean an interface that is easy to use and understand, i.e., well integrated into the existing language. Albeit the metalevel is to be used only for special purposes, it should be a tool for all programmers, and not limited to "kernel-hackers". By *dynamic*, we mean that metalevel computation is a property that can be added to individual objects or sets of objects at runtime. By *efficient*, we mean that the performance impact of using the MLI must not be prohibitive, and the overhead of reflectional capabilities should only be paid on use.

Our MLI builds on two concepts: first-class type information and runtime events.

The introspective part of the MLI provides a way to obtain runtime access to class-definitions and type-annotations as they existed in the source code. Type-information is reified by *pattern references* and *attribute references*. In BETA, patterns[2] are already first-class values, while attribute references are introduced in MetaBETA to turn attributes[3] into first-class values as well. Attribute references

[2]Similar to a Class in this context. Patterns are described in Section 3.
[3]Similar to slots in CLOS terminology.

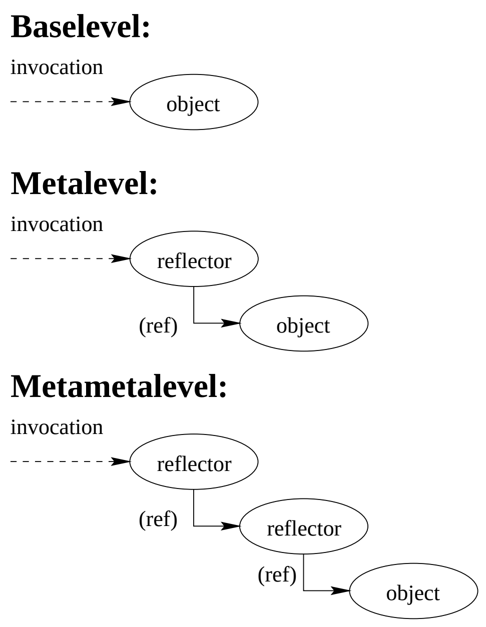

Figure 2: Reflectors and Metalevels.

are the only syntactical difference between BETA and MetaBETA, and integrate seamlessly into the existing type system.

To expose implementation primitives, the intercessory part of the MLI uses *reflectors*. A reflector is an ordinary programmer-defined BETA object that can be hooked into the runtime system to be automatically invoked whenever a given *event* occurs. MetaBETA defines three different events, associated with the language primitives object creation, object invocation, and object reclamation. Instead of having a metaobject that controls the execution of each object, there exist a metaobject for each reified language primitive. Thus, reflection can be performed on just parts of an object, making it easier to structure metacode and to control the performance overhead.

Figure 2 despicts the use of reflectors to control object invocation: Normally (baselevel), an object is invoked directly. If a reflector is associated with the invocation-event for a given object, the reflector is invoked instead of the object itself. The reflector is responsible for invoking the object, but is not required to do so. Because a reflector is an ordinary BETA object, it is possible to associate a reflector with a reflector-invocation-event, allowing, in effect, metametalevel computation and so on.

3 The Metalevel Interface

This section describes the set of primitives reified by the MetaBETA MLI. MetaBETA programs access the MLI through the globally visible `MLI` object.

Attribute references and pattern references are used as the foundation of the metalevel interface, and are described in Sections 3.1 and 3.2. The MLI itself is presented in Sections 3.3 and 3.4.

3.1 Patterns and Pattern References

BETA source code is a collection of *patterns*. The pattern is a unifying abstraction for concepts such as class, method, virtual method, exception, coroutine, and generics. A pattern `P` is declared as follows:

```
P: Super (#
   (* attribute list *)
   enter In
   do (* do-part *)
   exit Out
   #)
```

where `Super` defines an optional superpattern for `P`, `In` is an optional enter-list (formal parameters), and `Out` is an optional exit-list (return values). The do-part of a pattern consists of imperative code executed by instances of `P`.

Example 2 Figure 3 shows how patterns are used to model both classes and methods. The `Calc` pattern is used as a class. It has a single nested pattern `add` which is used as a method. There is no syntactical difference between class patterns and method patterns. The only difference is their usage. Figure 3 also declares an instance, `aCalc`, of the `Calc` pattern, and calls its `add` method. The return value from `add` is assigned to the integer variable `res`. Note, that the BETA assignment operator, `->`, assigns from left to right. □

```
    Calc:  (* Calc is a class pattern    *)
      (# add: (* add is a method pattern *)
           (# a,b,c: @Integer;
           enter (a,b)
           do a+b->c;
           exit c
           #);
      #);
    aCalc: @Calc;  (* aCalc is an object *)
    res: @Integer;
do (5,4)->aCalc.add->res;
```

Figure 3: The `Calc` pattern.

Dynamic pattern references turn patterns into first-class values. Given a dynamic pattern reference, it is possible to create instances of the referred pattern, and to check subpattern relationships by comparing it with another pattern reference. The code below declares a dynamic pattern reference, `calcP`, assigns the `Calc` pattern to `calcP`, and finally creates an instance of `Calc`:

```
      calcP: ##Calc; (* Pattern reference *)
      aCalc: ^Calc;  (* Object reference  *)
   do Calc##->calcP##;
      &calcP[]->aCalc[];
```

A pattern reference implicitly denotes the set of objects that are instances of the referred pattern (the *extension* set), and may therefore be used to denote the set of objects a given metalevel operation operates on[4].

3.2 Attribute References

Dynamic pattern references allow programs to explicitly talk about patterns. The MetaBETA MLI also needs to explicitly talk about *attributes* of patterns. For this purpose we add a new language concept, the *attribute reference*, to the BETA language.

Attribute reference declarations are *typed* to allow the compiler and programmer to make assumptions concerning the kind of attribute referred by a particular attribute reference. Metalevel code thereby becomes more readable, and the compiler will be able to generate efficient, type-safe code. An attribute reference is declared as:

```
attr_ref: .loc_qua*attr_qua
```

The syntax includes three qualifications:

1. `loc_qua`, the (optional) *location qualification*. The attribute referred by `attr_ref` must be an attribute of the pattern `loc_qua`, or of some superpattern of `loc_qua`.

2. The *kind qualification*, shown as `*` in the declaration of `attr_ref`, constrains the attribute reference to refer specific kinds of attributes. Examples of attribute kinds are dynamic (`^`) and static (`@`) object references.

3. `attr_qua`, the (optional) *attribute qualification*, constrains the qualification of the attribute referred by `attr_ref`. For example, an attribute reference

[4]To allow pattern references that denote more general extension sets than those corresponding to pattern extensions, the MetaBETA pattern reference mechanism has been generalized into a *type reference* mechanism, as described in [Brandt & Knudsen 96]. The generalized mechanism allows pattern references with single-element extension, and pattern references corresponding to sets of patterns.

```
Person: (# ... #);
Family:
  (# mother: ^Person;
     father: ^Person;
     child: ^Person;
  #);
aFamilyPerson: .Family^Person;
```

Figure 4: Example of an Attribute Reference.

may be qualified to refer only attributes referring text objects.

Example 3 The MetaBETA fragment in Figure 4 declares the patterns `Person` and `Family`, and the attribute reference `aFamilyPerson`. Families have attributes `mother`, `father`, and `child`, each of which are dynamic references to person objects. The attribute reference `aFamilyPerson` has location qualification `Family`, kind qualification "dynamic object reference", and attribute qualification `Person`, which qualifies it to refer any of the family attributes. The attribute reference can be used as:

```
     aFamily: @Family;
     aPerson: ^Person;
do ...
     Family.mother:->aFamilyPerson;
     aFamily.(aFamilyPerson)[]->aPerson[];
```

First, the `Family.mother` attribute is assigned to the `aFamilyPerson` attribute reference. Then, `aFamilyPerson` is applied to the `aFamily` object, thus returning a reference to the mother in `aFamily`. In this example, all necessary type-checking can be done at compile-time due to attribute reference qualifications. □

Using the syntax:

```
uc_attr_ref: .?
```

we may declare an unconstrained attribute reference that is allowed to refer any kind of attribute, including attribute reference attributes themselves.

3.3 Introspective Capabilities

An executing program can access metainformation about itself by using the introspective capabilities. Metainformation is represented as pattern and attribute references. The MLI allows extraction of complete object descriptions modulo the do-parts.

Introspecting Attribute References Given an attribute reference, the BETA MLI allows extraction of the name of the attribute, the qualification of the attribute, and the attribute kind. Qualifications are returned as dynamic pattern references that, e.g., can be compared against known patterns to locate attributes of specific types in an object.

Introspecting Pattern References Given a pattern reference, `p`, it is possible to scan the attributes that exist in instances of `p`. For each attribute of `p`, an attribute reference, `ar`, is returned. Persistent stores, object browsers, and schema evolution tools, to mention a few, can use this to handle objects in a type independent manner.

The `scanDyn` method of the MLI provides a way to scan all object reference attributes of a pattern:

```
scanDyn:
  (# current: .^;
  enter (p:##Object)
  do ... INNER ...
  #);
```

The `scanDyn` iterator pattern calls `INNER` for each attribute of the pattern `p`[5].

Example 4 To print the names of all dynamic object reference attributes of the `Family` pattern, we can write:

```
Family##->MLI.scanDyn (#
do current->MLI.AttrName->screen.putline;
#);
```

This will print out the text strings `mother`, `father`, and `child`. The real MLI allows subpatterns of `scanDyn` to strengthen the qualification of `current`, thus supporting static type-checking of code inside subpatterns of `scanDyn` that use `current` to access object attributes. □

3.4 Intercessory Capabilities

The intercessory capabilities opens up the runtime system, providing access to the implementation of language primitives. The MLI provides an abstract interface to the implementation of object creation, object execution, and object destruction.

The MLI presents each reified primitive by: i) a controller object in the runtime system; ii) a reflector pattern, which is used as superpattern for user-defined

[5]Execution of a method m always starts at the least-specific specialization of m, and continues down the specialization-chain at each `INNER` imperative. An `INNER` imperative for which no specialization is specified corresponds to the empty imperative.

reflectors; iii) a method that directly executes the reified primitive; and iv) a method that invokes the default implementation of the primitive, shortcutting the reflectional mechanisms. Table 1 summarizes the intercessory mechanisms supported by the MetaBETA MLI. These mechanisms are described in detail in the following sections.

Primitive	Controller	Reflector	Callable Int.	Default Impl. Int.
Obj. Creation	AllocCntr	AllocReflector	`&`	defaultAlloc
Obj. Execution	ExecCntr	ExecReflector	exec	defaultExec
Obj. Deletion	GCCntr	GCReflector	N/A	N/A

Table 1: MetaBETA Intercession Primitives.

3.4.1 Object Creation

A new BETA object is created with the `&` operator. In MetaBETA, `&` is reified, allowing user-defined code to be executed in response to `&` invocations. The user-defined code is presented to the metalevel interface in the form of instances of the `AllocReflector` pattern:

```
AllocReflector:
  (# type:< Object;
     onAlloc:< (#
       enter (new:^type)
       do INNER;
       #);
  #);
```

Invocation of `AllocReflector` objects is controlled by the `AllocCntr` metaobject whose interface is shown in Figure 5. An `AllocReflector` is inserted into the runtime system by the `AllocCntr.register` method. As a parameter, `AllocCntr.register` takes a pattern reference `q`, and an `AllocReflector` reference `ar`. The `q` parameter determines the object creations that should trigger the `ar.onAlloc` virtual[6]: When objects of type `q` is created, `ar.onAlloc` is called with the new object as parameter.

In case several reflectors apply to an object creation, those registered with the most general `q` are executed first. If several `AllocReflectors` are registered with the same `q`, they are called in last-registered-first-called order.

Example 5 In Figure 6, the reflectors `VehicleAR` and `CarAR` are registered to reflect on, respectivly, the allocation of `Vehicle` and `Car` objects. Then, a `Vehicle`

[6]`onAlloc` is a virtual pattern, as determined by `:<`, and can be specialized in subpatterns of `AllocReflector`.

```
AllocCntr: @
  (# register: (#
       enter (q:##Object,
              ar:^AllocReflector)
       exit (reflId: @Integer)
       #);
     remove: (#
       enter (reflId: @Integer)
       #);
  #);
```

Figure 5: `AllocCntr` interface.

and a `Car` is created. The `Vehicle` creation only triggers `VehicleAR.onAlloc`, whereas the `Car` creation triggers `VehicleAR.onAlloc` and then `CarAR.onAlloc`, since `Vehicle` is a superpattern of `Car`. As a result of executing Figure 6,

```
A: Vehicle
B: Vehicle Car
```

is therefore printed to the screen. □

The `AllocCntr` allocates the new object before any `AllocReflector` is called. Hence, reflectors cannot change the type or size of a newly created object, a limitation which avoids severe performance penalties. In particular, the size and type of new objects must be known at compile-time, to allow objects to be statically inlined in other objects and to support type-inference.

Implementation: `AllocReflector`s can be efficiently implemented by adding an extra word to the runtime class representation. Figure 7 illustrates the runtime data structures built in response to the registration of the two `AllocReflector`s from Figure 6.

The registration of `VehicleAR` installs a reference from the `Vehicle` class object to `VehicleAR`, and a reference from the `Car` class object to `VehicleAR`, since `Car` is a subclass of `Vehicle`, and since a more specific `AllocReflector` was not already installed for `Car`. The subsequent registration of `CarAR` overwrites the reference from the `Car` class object to `VehicleAR`, since the `VehicleAR` reference was "inherited" from the `Vehicle` class object. If multiple reflectors are registered with the same class, this will be represented as a linked list of reflectors.

Testing for an `AllocReflector` can now be implemented as a constant-time check for the presence of a reflector in the runtime class representation for the newly created object. However, this overhead will negatively affect non-reflective applications, and is therefore not acceptable. However, by exploit-

```
                VehicleAR: @MLI.AllocReflector
                  (# onAlloc:: (#
                       do ' Vehicle'->screen.puttext;
                       #);
                   #);
                CarAR: @MLI.AllocReflector
                  (# onAlloc:: (#
                       do ' Car'->screen.puttext;
                       #);
                  #);
                aVehicle: ^Vehicle;
                aCar: ^Car;
             do (Vehicle##,
                 VehicleAR[])->MLI.AllocCntr.register;
                (Car##,
                 CarAR[])->MLI.AllocCntr.register;
                'A: '->screen.puttext;
                &Vehicle[]->aVehicle[];
                '\nB:'->screen.puttext;
                &Car[]->aCar[];
```

Figure 6: `AllocReflector` Example

ing the copying collection scheme used in the youngest object generation of the Mjølner BETA System, an implementation with no overhead on non-use can be achieved. In BETA, object creation is efficiently performed by bumping the top-of-heap pointer with the instance size retrieved from the class object, followed by a check that no heap-overflow occurred. In MetaBETA, this implementation is unchanged, and thus no additional overhead is imposed. However, if an `AllocReflector` is installed in a class object, we arrange for the class object to return an instance-size large enough to force a heap-overflow. On heap-overflow, an extra check will determine whether it is actually time for a garbage-collection, or whether an `AllocReflector` is to be invoked.

3.4.2 Object Execution

MetaBETA allows programmer defined `ExecReflector` objects to be invoked on object execution events, so the default implementation can be specialized. An `ExecReflector` is defined as:

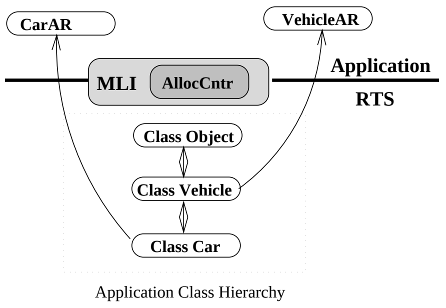

Figure 7: AllocCntr Runtime Representation.

```
ExecReflector:
  (# type:< Object;
     onExec:<
       (# callNextReflector: (# do ... #);
       enter (method:^type, dp:##Object)
       do INNER
       #);
  #);
```

The `ExecCntr` metaobject is responsible for invoking the `onExec` virtual when specific do-parts are executed. To understand the exact `ExecReflector` semantics, we first take a closer look at BETA method execution.

The method call `(5,4)->aCalc.add->res`[7] (from Figure 3) is executed in the following steps:

1. A new instance, `method`, of the `aCalc.add` pattern is created[8]. `method` is an ordinary BETA object playing the role of an activation record.

2. The actual input list, `(5,4)`, is evaluated and assigned to the formal enter list of `method`, thus assigning 5 to `method.a` and 4 to `method.b`.

3. The do-part of `method` is executed, assigning 9 to `method.c`.

4. The formal exit list, (`c`), of `method` is evaluated and assigned to the actual output list (`res`). Hence, the value of `method.c` is assigned to `res`.

Notification of `ExecReflector` objects embraces step 3 above. This allows the `ExecReflector` to query and change the value of attributes of the `method` object before and after its execution, and, if necessary, to actually execute the `method` object itself. Thus, the `ExecCntr` metaobject exposes three aspects to the

[7]Syntactic sugar for `(5,4)->&aCalc.add->res`, `&` being the object creation operator.

[8]If `add` had been a virtual pattern, the allocation step would include a table lookup to retrieve the exact binding of `add`.

scrutiny of `ExecReflectors`: The state of the method object before execution, the actual execution of the method, and the state of the method object after execution.

```
ExecCntr: @
  (# register:
       (# conflict:< Exception (#
            enter (cfl:^ExecReflector)
            do INNER
            #)
       enter (dp:##object,
              er:^ExecReflector,
              active:@Boolean)
       do ...
       exit (reflId:@Integer)
       #);
     remove: (#
       enter (reflId:@Integer)
       do ...
       #);
  #);
exec: (#
  enter (method:^Object, dp:##Object)
  do ...
  #);
defaultExec: (#
  enter (method:^Object, dp:##Object)
  do ...
  #);
```

Figure 8: `ExecCntr` interface.

The `ExecCntr` interface is shown in Figure 8. To register an `ExecReflector`, metalevel code calls `ExecCntr.register` with a do-part specification `dp`, an `ExecReflector` parameter `er`, and a boolean parameter `active`. Then, whenever an object that qualifies to `dp` is about to execute the `dp` do-part, `ExecCntr` calls the `er.onExec` virtual with the `method` object and do-part to be executed as parameters.

The `active` parameter to `ExecCntr.register` reflects whether `er` handles the execution of method objects itself, or whether it simply monitors object execution events.

If several `ExecReflectors` apply, passive reflectors are executed in the same order as described for `AllocReflector` objects. An `ExecReflector` passes on control to the next reflector in the chain by calling `callNextReflector`. When all passive reflectors have been executed, `callNextReflector` passes on control to a possible active reflector. It is called last since it is responsible for actually executing the object. If no active reflector applies, the `method` object is automatically executed when the last passive `ExecReflector` calls `callNextReflector`. Usually there should be at most one active reflector applicable. If more than one is registered, the `conflict` exception will be raised at registration time. However, by catching and ignoring or explicitly sorting out conflicts, it is possible to register several active reflectors applying to the same object execution.

`exec` is a procedural interface to the object execution primitive. It executes the do-part of an object starting at a specified level of the specialization chain. Using `exec` to execute an object will trigger possible `ExecReflector`'s installed for the object, whereas `defaultExec` will not. These methods can be used by active programmer-defined `onExec` methods to execute an object.

```
    addER: @MLI.ExecReflector
      (# type:: Calc.add;
         onExec:: (#
          do method.a->screen.putint;
             method.b->screen.putint;
             callnextReflector;
             method.c->screen.putint
          #);
      #);
    aCalc: @Calc;
    res: @Integer;
do (Calc.add##,
     addER[],FALSE)->MLI.ExecContr.register;
   (1,2)->aCalc.add->res;
```

Figure 9: Using an `ExecReflector`.

Example 6 Assume that we want to monitor calls to the `add` method of `Calc` objects from Figure 3. To do so, an `ExecReflector` is associated with a dynamic pattern reference to `Calc.add`, as shown in Figure 9. Execution of `Calc.add` instances is then redirected into execution of `addER.onExec`, whose `method` parameter will refer the `aCalc.add` instance about to be executed. `addER.onExec` prints the parameters to the method call, and then calls `callNextReflector` to execute the `method` object. Notice how `addER` specializes the `type` virtual to `Calc.add`, giving `onExec` static knowledge of the attributes of the `method` object.

The registration of `addER` shown in Figure 9 makes `addER` reflect on the execution of the `add` method of any `Calc` instance. Alternatively, if reflection is only wanted for a specific `Calc` instance, e.g., `aCalc`, then `addER` can been registered as:

```
do (aCalc.add##,addER[],FALSE)
     ->MLI.ExecContr.register;
```

The difference is that the latter uses the `aCalc.add##` qualification, whereas Figure 9 used the more general `Calc.add##` qualification. Reflection happens on the same do-part, but the extension of `aCalc.add` is smaller than the extension of `Calc.add`, thereby restricting reflection to take place on a smaller set of object executions. □

Example 7 The following code shows an `ExecReflector` that might be used to implement object distribution. Methods to be executed in a remote address space are packed and sent to the remote host for execution:

```
DistER: @MLI.ExecReflector
  (# onExec:: (#
       do method[]->packAndSendToRemoteHost
       #);
  #);
```

On the remote host, the `method` object would get unpacked, executed, (using `exec` or `defaultExec`), and finally sent back. `DistER` is an active `ExecReflector`, and the following code registers the `aCalc.add` method for remote execution:

```
do (aCalc.add##,DistER[],TRUE)
     ->MLI.ExecContr.register;
```

□

Implementation Execution reflection is implemented by runtime code modifications. Since patching does not occur until an `ExecReflector` is installed, and patches are undone when the `ExecReflector` is removed, there is no overhead for programs that do not use the mechanism.

Reconsider Example 6: It was demonstrated that we could either choose to reflect on the execution of all add methods, using the qualification `Calc.add##`, or we could choose to reflect only on the execution of the add method of a specific `Calc` object, using the qualification `aCalc.add##`. In any case, the code patch will be applied to the piece of machine-code that implements the `add` do-part. Thus, to reflect on `aCalc.add##` executions, patch code must verify that the current execution does not correspond to the execution of the `add` method of a `Calc` instance different from `aCalc`, before invoking the installed reflector. As a result, reflection on the `aCalc.add##` qualification will induce an overhead on all methods qualifying to `Calc.add##`. Preliminary measurements in Section 5 will illustrate this point.

3.4.3 Object Reclamation

The `GCCntr` metaobject shown in Figure 10 allows MetaBETA programs to monitor the reclamation of specific objects. BETA is garbage-collected, so objects are never explicitly deallocated.

```
GCReflector:
  (# onReclaim:< (#
       enter (reflId: @Integer)
       do INNER
       #);
  #);
GCCntr: @
  (# register: (#
       enter (obj: ^Object,
              gr: ^GCReflector)
       exit  (reflId: @Integer)
       #);
     query:  (#
       enter (reflId: @Integer)
       exit  (obj:^Object)
       #);
     remove: (#
       enter (reflId: @Integer)
       #);
  #);
```

Figure 10: `GCCntr` interface.

To register a `GCReflector`, metalevel code calls `GCCntr.register` with an object parameter `obj` and a `GCReflector` parameter `gr`. Then, when `obj` becomes garbage, `gr.onReclaim` is called, and passed the `reflId` to identify the dead object. `reflId` is a unique receipt for the registration of a reflector, and can be used to `remove` a reflector or to obtain a direct reference to the object corresponding to `reflId`, using `GCCntr.query`. In this way, `reflId` can be seen as a weak object reference.

`GCCntr` does not allow garbage-collection of an object to be prevented, since this would introduce problems comparable to those of incremental and concurrent garbage-collection. Likewise, GC-tracing of all instances of some pattern is not

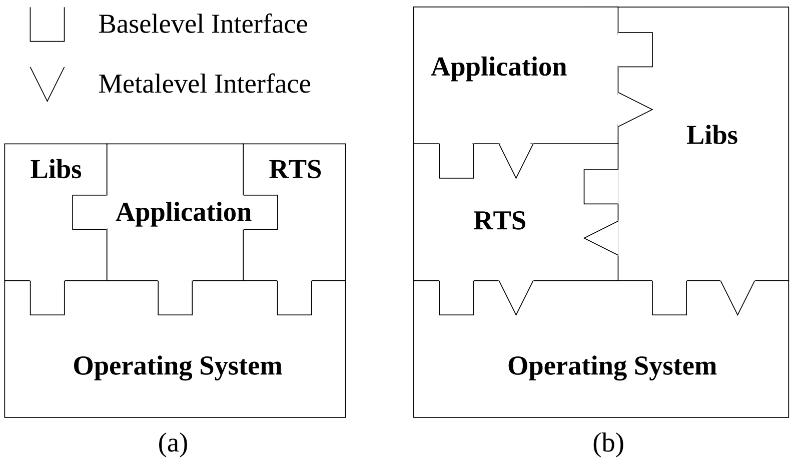

Figure 11: System Structures for: (a) a traditional compiled language, and (b) the metalevel architecture.

supported: In the case of a copying garbage-collector, that would require an expensive scan of all garbage after each collection.

Functionality corresponding to `GCCntr` already exists in the current BETA implementation, and is used by the source code debugger to track objects. However, the functionality is not available through a public interface.

4 The Metalevel Architecture

The implementation of a program with a runtime metalevel interface must maintain a causal connection to the metainformation at run-time. However, for most compiled languages, there is no specific component responsible for executing the code and maintaining the metainformation. The compiler generates machine-code to be executed directly by the processor.

The MetaBETA metalevel architecture is a runtime organization for compiled languages that supports a metalevel interface by making the runtime system responsible for code execution and metainformation maintenance. The organization is very similar to the implementation of most semi-interpreted and dynamically typed languages, such as CLOS [Kiczales et al. 91] and Smalltalk [Goldberg & Robson 89].

4.1 Runtime Organization

We think of the runtime system as a virtual machine extending the underlying operating system. In the same way as binary programs may be considered data to be executed by the operating system, we can think of compiled programs as data to be executed by the runtime system. Conceptually, the runtime system is responsible for executing the compiled code. In practice, it hands the machine-code part of the compiled program to the underlying CPU for direct execution.

The virtual machine view of the runtime system induces a different system structure than for traditional compiled languages. The difference is shown in Figure 11. For a traditional compiled language, the application runs directly on top of the operating system, supported by the runtime system (RTS) and libraries. The runtime system, operating system, and libraries are viewed as black box implementations, i.e., they can only be accessed through baselevel interfaces. In the metalevel architecture (Figure 11b) the runtime system is the central component. It is responsible for running the application and providing metainformation to the application and libraries through its metalevel interface. The runtime system and libraries can also use a metalevel interface provided by the operating system [Kiczales & Lamping 93].

The benefit of the virtual machine view is that it conceptually as well as in practice defines an entity to provide the metalevel interface for a compiled program.

4.2 The Runtime System

A central responsibility of the runtime system is to load and link compiled MetaBETA code. The compiler generates code to the MetaBETA runtime system, not to the operating system. A special code-format is used that contains the usual machine-code part, and also the metainformation needed by the metalevel interface.

Thus, the runtime system is the only executing program seen by the operating system. To execute a compiled MetaBETA program, the runtime system is started from the command-line with a parameter specifying which application it has to load and execute. This is in contrast to a traditional compiled language where the compiler generates executables for each application.

Since the runtime system is responsible for linking and loading code it can maintain and update the metainformation for a given program execution. Furthermore, it knows the code-layout in memory so runtime code-modification can be implemented efficiently.

5 Status

The implementation of MetaBETA is underway. Our strategy is to make prototype implementations of specific parts of the metalevel interface by extending the current Mjølner BETA implementation. This approach allows us to quickly implement and test parts of the MLI, and to verify the design on real metacode. Also, several implementation issues can be tested and verified independently. We are also working on a detailed design for the BETA virtual machine, i.e., the code-format, the loader, and the internal representation of metainformation.

Pattern	Normal	w/ Reflector	Overhead Total	Overhead Patch
Pattern Reflection				
A	1.04	3.25	212%	16%
B	1.29	3.47	167%	15%
Object Reflection				
objA	0.29	2.43	738%	19%
objA′	0.29	0.40	-	40%

Table 2: Reflector Overhead. Times are in μsecs and is measured on a 75MHz SparcStation 20.

A working prototype of the `ExecReflector` has been implemented with the main purpose of examining the overhead of inserting reflectors at runtime by patching binary code[9]. To determine the overhead, we have measured the overhead of reflecting on a specific pattern (Pattern Reflection), and on a specific instance of a pattern (Object Reflection). The results are shown in Table 2. The patterns `A`, `B`, and `Refl` are defined as follows:

```
A: (# do INNER #);
B: A(# do INNER #);
Refl: @MLI.ExecReflector (#
   onExec:: (# do callNextReflector #);
#);
```

For *pattern* reflection we both measure the time to execute a top-level pattern `A`, and its subpattern `B`, because the BETA compiler generates a different call sequence for code called through `INNER`.

Most of the *Total Overhead* shown in Table 2 is the inherent overhead of creating and invoking the `Refl.onExec` method, which takes 1.73 μsecs. Thus, the patch-code overhead is $T_{w/reflector} - (T_{normal} + T_{Refl})$. It is shown in percent in the last column.

Reflection on execution of a single object requires that the patch code checks whether it actually applies to the current object execution. This overhead is measured under *Object Reflection*: `objA` and `objA′` are both instances of the `A` pattern, but only execution of `objA` is reflected upon. The time for executing an existing object is much faster than object creation and execution, because it does not require a new object to be allocated. Hence, the total overhead is much worse

[9]The prototype does not as yet support dynamic link and load of code.

than for pattern reflection. Installing a reflector on a specific object introduces an invocation overhead of 0.11 μs on other objects of the same type.

We expect these preliminary numbers to be acceptable as compared to the overhead of, e.g., implementation of object distribution by other means. However, to verify this claim, future work includes the implementation of distributed BETA based on MetaBETA abstractions.

6 Related Work

The work presented in this paper is related to a number of previous research efforts in reflective language design and implementation. How they relate to MetaBETA is described in the following.

CLOS [Kiczales et al. 91] includes a comprehensive metalevel interface, (the CLOS MOP, or metaobject protocol), which allows the metalevel programmer to inspect and change several primitives of the programming language, including redefinition of slot access, multiple inheritance semantics, and replacement of metaclasses. The CLOS MOP greatly influenced the design of the introspective part of the MLI. However, the scope of the intercessory parts in BETA is on purpose much more limited, because we do not aim at changing fundamental language semantics such as inheritance order and scoping rules. Our focus is more on opening up the implementation of a few basic language primitives.

Also, BETA is statically typed, whereas CLOS is dynamically typed, leading to a considerably different style of metalevel interface. Most notably, the BETA MLI allows, but does not force, the metalevel programmer to express static typing constraints, thereby leading to more readable and possibly more efficient metalevel code. ABCL/R2 [Masuhara et al. 92], AL-1/D [Okamura et al. 93], and SELF [Hölzle & Ungar 94] are further examples of dynamically typed programming languages with reflectional capabilities. As with CLOS, the most notable difference between the metalevel interfaces of these languages and MetaBETA, is the issue of static versus dynamic typing.

Open C++'s [Chiba 93] (version 1) concept of *reflect methods* was the direct inspiration for the reflector mechanism used in MetaBETA to reify basic language primitives. A reflector in MetaBETA controls the implementation of a specific language primitive, which is similar to the CodA [McAffer 95] system for Smalltalk.

The newest version of Open C++ [Chiba 95] implements a compile-time metaobject protocol, with metalevel code controlling the compilation of baselevel code. A similar approach is used in the START system [Kirby et al. 94], implemented for the Napier88 language [Morrison et al. 93]. These systems offer compile-time reflection, and thus assumes that the metalevel is statically known. In comparison, MetaBETA is entirely dynamic, with reflective hooks inserted at run-time, i.e., MetaBETA offers later binding than systems based on compile-time reflection. ABCL/R3 [Masuhara et al. 95] may be seen as a combination of the two approaches: Partial evaluation attempts to compile away interpretation of the metalevel, whereas different code-versions support runtime change of metaobjects.

The C++ concept of *pointers to members* may be considered a restricted version of attribute references, which the MetaBETA MLI is based on. However, the absence of runtime type information, run-time type checking, and restriction to function members renders C++ pointers to members less powerful than attribute references.

7 Conclusion

This paper desribed an extension of the statically-typed and compiled language BETA with a metalevel interface. Our metalevel interface is based on first-class attributes and classes, which allows metalevel computations to be easily integrated into the existing language without circumventing or changing the existing type-system.

Our metalevel interface is a runtime entity giving a running program access to the representation of objects. The use of the metalevel interface does not require access to source code. This provides a large flexibility that allows metalevel code to be applied to classes written before the metalevel code itself.

Previous metalevel interfaces for statically typed languages have either circumvented the language type-system, or has been based on a compile-time MLI. With the introduction of attribute references, MetaBETA provides a both runtime based and type-safe metalevel interface for statically typed languages.

Acknowledgements

We would like to thank Ole Lehrmann Madsen and Jørgen Lindskov Knudsen who read earlier versions of this paper and provided many valuable comments.

References

[Arnold & Gosling 96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley Publishing Company, Reading, MA, March 1996.

[Brandt & Knudsen 96] S. Brandt and J. L. Knudsen. Patterns and Qualifications in BETA: A Case for Generalization. In *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP)*, July 1996.

[Brandt & Madsen 94] S. Brandt and O. L. Madsen. Object-Oriented Distributed Programming in BETA. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Lecture Notes in Computer Science 791*, pages 185 – 212. Springer-Verlag, January 1994.

[Brandt 94] S. Brandt. Implementing Shared and Persistent Objects in BETA — Progress Report. Technical report, Department of Computer Science, University of Aarhus, May 1994.

[Chiba 93] S. Chiba. Designing an Extensible Distributed Language with a Meta-Level Architechture. In O. Nierstrasz, editor, *Proceddings of the 7th European Conference on Object-Oriented Programming*, volume 707 of *LNCS*, pages 482 – 501. Springer-Verlag, July 1993.

[Chiba 95] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1995.

[Goldberg & Robson 89] A. Goldberg and D. Robson. *Smalltalk-80. The Language*. Addison-Wesley, Reading, MA, 1989.

[Grønbæk et al. 94] K. Grønbæk, J. Hem, O. Madsen, and L. Sloth. Hypermedia Systems: A Dexter-Based Architecture. *Communications of the ACM*, 37(2):64 – 74, February 1994.

[Hölzle & Ungar 94] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness with Performance. In *Proceedings of the 9th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 229 – 243, October 1994.

[Kiczales & Lamping 93] G. Kiczales and J. Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of International Workshop on Object-Orientation in Operating Systems (IWOOS)*, 1993.

[Kiczales 92] G. Kiczales. Towards a new model of Abstraction in Software Engineering. In A. Yonezawa and B. C. Smith, editors, *Proceedings of International Workshop on Reflection and Meta-level Architecture (IMSA)*, pages 1–11, November 1992.

[Kiczales et al. 91] G. Kiczales, J. Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kirby et al. 94] G. Kirby, R. Connor, and R. Morrison. START: A Linguistic Reflection Tool Using Hyper-Program Technology. In *Proceedings of the 6th International Workshop on Persistent Object Systems*, pages 346 – 365, September 1994.

[Madsen et al. 93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, Reading, MA, 1993.

[Malhotra 93] J. Malhotra. Dynamic Extensibility in a Statically-Compiled Object-Oriented Language. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*, Kanazawa, Japan, November 1993.

[Masuhara et al. 92] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa. Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently. In A. Paepcke, editor, *Proccedings of the 7th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 127 – 144, October 1992.

[Masuhara et al. 95] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 1995.

[McAffer 95] J. McAffer. Meta-Level Programming with CodA. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP)*, August 1995.

[Meyer 92] B. Meyer. *Eiffel, The Language*. Prentice Hall, 1992.

[Morrison et al. 93] R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, A. Dearly, G. N. C. Kirby, and D. S. Munro. The Napier88 Reference Manual (Release 2.0). Technical Report CS/93/15, University of St Andrews, 1993.

[Okamura et al. 93] H. Okamura, Y. Ishikawa, and M. Tokoro. Metalevel Decomposition in AL-1/D. In *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS)*, November 1993.