The Design of a Meta-Level Architecture for the BETA Language

Søren Brandt and René W. Schmidt

Department of Computer Science
University of Aarhus
DK-8000 Aarhus C, Denmark
email: {sbrandt,rws}@daimi.aau.dk

Abstract

The design of a meta-level architecture for a compiled and statically typed object-oriented programming language is presented. The architecture provides compiled languages with sufficient runtime information to implement a full-blown typed meta-level interface, including both introspective and introcessory capabilities. The central component of the meta-level architecture is the runtime system. It is viewed as a virtual machine implementing the functionality of the programming language, and therefore provides a natural entity to be extended with a meta-level interface.

The meta-level interface is based on a novel language construct, a meta-reference called *attribute reference.* Attribute references extend the expressive power of the language by allowing the programmer to not only have references to objects, but also to have *typed* references to instance variables. The meta-level interface provides access to type-information and reifies language primitives, such as object invocation and creation. Efficiency of the meta-level architecture is provided by careful design of the meta-level interface and by runtime code modifications. As an example, is it shown how an interactive programming environment can be constructed in the language itself, on top of the meta-level interface.

1 Introduction

In order for a programming language implementation to support addition of new uniform abstractions and tailoring of existing ones, it should provide a *meta-level interface*, providing runtime access to type-information and interception of language primitives.

Very few implementations of statically typed languages include a meta-level interface. That is, most current implementations of compiled and statically typed languages are limited to only allowing code to be written *in* the language and not *for* the language.

Debuggers, persistent storage, and distribution are examples of functionalities that rely on information that is not typically provided by the programming language itself, and therefore cannot be written directly in the language. For example, a persistent store needs a detailed knowledge of object representation and instantiation, in order to serialize objects to stable storage, and to re-instantiate objects from their serialized form. The programmer has to either rely on ad-hoc methods such as pre-processing of source-code, or rely on compiler-specific and possibly undocumented features to gain access to the needed information. This leads to implementations that are hard to maintain, difficult to port, and possibly with unnecessary limitations.

This paper presents the design of a meta-level interface and meta-level architecture for the object-oriented language BETA [Madsen et al. 93]. BETA is a statically typed[1] language, where programs are compiled to native machine code for efficient execution. Reasons for the lack of statically typed languages with meta-level interfaces include:

- Static type systems often make it hard to express

[1]With statically typed we mean that most type-checks are done at compile-time. BETA's type-system is covariant, so not all type-checks can be done at compile-time.

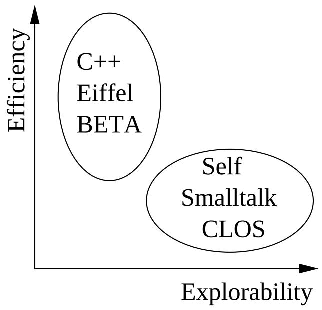

Figure 1: Efficiency versus explorability for different object-oriented languages

sufficiently flexible meta-level interfaces.

- A meta-level interface for a statically typed language must enforce static typing constraints.
- Only a limited amount of information about the source code is available at runtime.
- The runtime cost induced by the addition of a meta-level interface to a statically typed language may be prohibitive, due to their generally better runtime performance.

To address these problems, we propose a meta-level architecture for statically typed compiled languages. The central component in the architecture is the runtime system, which is viewed as a virtual machine whose base-level interface implements the functionality needed by the programming language. The meta-level interface of the runtime system provides functionality such as access to type-information, and tailoring of object invocation, creation, and destruction.

Viewing the runtime system as the central execution engine is similar to the view taken by dynamically typed[2] languages, such as Smalltalk [Goldberg & Robson 83], Self [Hölzle & Ungar 94], and CLOS [Kiczales et al. 91]. These languages are based on integrated programming environments that are responsible for the execution of programs, thereby supporting an exploratory programming style. This is in sharp contrast to most statically typed languages, where the development environment and language are separated. Statically typed languages, on the other hand, provide more readable programs through explicit type annotations, and they produce more efficient code. This difference is shown in Figure 1.

[2]Dynamically typed here means that most type-checks are done at runtime.

An interactive programming environment for statically typed object-oriented languages can be built using the meta-level architecture outlined in this paper. Instead of building specialized compilers and tools for such an environment, we are focussing on building a compiler and runtime system that allows the construction of such an environment on top. This architecture allows a clear separation between development environment and runtime environment, while narrowing the gab between explorability and efficiency.

The next section presents the motivation for the design of the meta-level architecture. Section 3 describes the central role the runtime system plays in the architecture. Section 4 presents the meta-level interface of the runtime system. Section 5 shows how the meta-level architecture facilitates the construction of an interactive programming environment. Section 6 summarizes related work. Finally, section 7 presents our conclusion.

2 Motivation

Abstractions for generic functionality can be structured in frameworks, so code can be reused in a wide variety of applications. However, several researchers [Kiczales 92, Rao 91] have argued that frameworks often do not get reused because of high performance overhead. A framework might provide the right abstraction, but its implementation is not optimized for the needs of the application. The solution to this problem is to make *open implementations*, where default implementations can be overridden by the application when needed. Thus, a framework should provide two interfaces, a *base-level interface* that defines its behaviour, and a *meta-level interface* that exposes certain parts of its implementation.

For example, an application implementing a fine-grained parallel algorithm and a program implementing a file transfer protocol both need the same basic abstraction: object distribution and mobility. However, the performance requirements for the two applications are very different. The first needs fine-grained object mobility in order to achieve high performance, while the other needs a protocol optimized for large objects. If the distribution library includes a meta-level interface that allows tailoring of the communication protocol, it would be possible for the same library to satisfy both of the above needs.

The need of an open language implementation has been recognized in the design and implementation of a number of substrate systems for the Mjølner BETA system, including:

- Type-orthogonal persistence [Brandt 94], including a simple persistent store, and a multiuser object-oriented database [Sloth & Hem 93, Grønbæk et al. 94].
- Distributed object system [Brandt & Madsen 94, Brandt 94].
- Source-level debugger.
- Embedded interpreter for the BETA language [Malhotra 94].
- Object- and class-browsers.

During work with these systems, a number of problems concerning portability and expressibility have been recognized. Most of these problems would have been non-existing, had the BETA language included a suitable meta-level interface. Our work on a meta-level architecture was initiated to solve these problems.

In general, a language meta-level interface turns the language implementation itself into an open implementation, but also provides support for implementing other open implementations. The language meta-level interface provides a standardized and efficient way of providing hooks and tailoring parts of an implementation, and is necessary to implement type-independent libraries, such as distribution and persistence, whose subject matter are primitives of the language itself. The language meta-level interface is not strictly necessary to implement a meta-level interface for libraries that define their own primitive objects, e.g., a GUI library. The meta-level architecture can, however, still be utilized by such libraries in order to simplify and generalize their implementation.

3 The Meta-Level Architecture

Informally, we can think of a runtime system as the part of a language implementation that is responsible for executing a program. For interpreted languages, the runtime system is the interpreter itself. The runtime system interprets the source code, maintains variable bindings, performs memory management, and checks for runtime errors.

For compiled languages, the distinctions between compiler, runtime system, and libraries are blurred. There is no specific component responsible for executing the code — the compiler generates machine-code to be executed directly by the processor. A compiler, however, assumes the presence of certain functionality when it generates code. This functionality is normally referred to as the runtime system. For an object-oriented language this would typically include object instantiation, runtime type checking, and garbage collection.

In this section we present a view that unifies the notion of runtime system for interpreted and compiled languages. The resulting notion combines the flexibility of an interpreted language with the efficiency of a compiled language.

3.1 The Runtime System as a Virtual Machine

We think of the runtime system as a virtual machine extending the underlying operating system. For example, the BETA runtime system in concert with the hardware and operating system defines a complete BETA virtual machine. 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, however, it hands the machine-code part of the compiled program to the underlying CPU for direct execution.

The benefit of the virtual machine view is that it conceptually as well as in practice defines an entity to provide the meta-level interface for a compiled language. Traditionally, virtual machines are black boxes designed to hide all implementation details from their clients. In the meta-level architecture the virtual machine is opened up, so implementation details are accessible through the meta-level interface. Hence, an application can perform reflective computations.

This architecture marks a departure from the efficient-code-generator view of a compiler. The compiler is now viewed as a supplier of information to the runtime system about a program. This information is a representation of the source-code that allows ef-

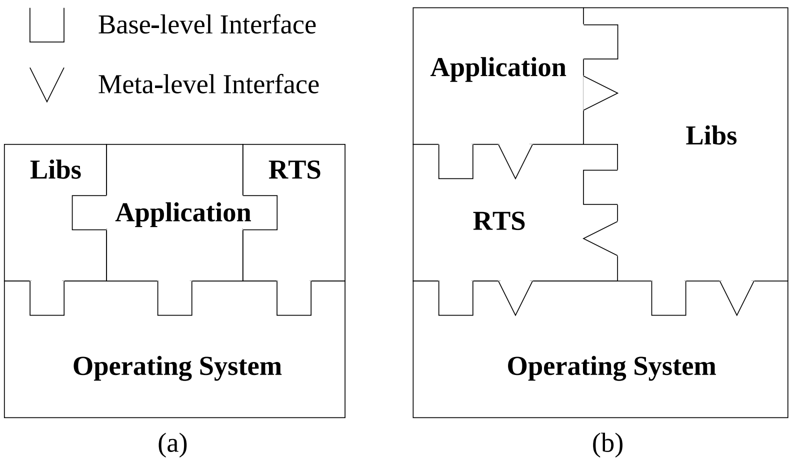

Figure 2: System Structures for: (a) a traditional compiled language, and (b) the meta-level architecture.

ficient execution and, at the same time, maintains a mapping to the source-code. This view unifies the notion of interpreted and compiled languages.

3.2 System Structure

The virtual machine view of the runtime system induces a different system structure than for traditional compiled languages. The difference is shown in Figure 2. 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 base-level interfaces. Libraries also do not have any direct contact with the runtime system and are therefore not able to access RTS information.

In the meta-level architecture (Figure 2b) the runtime system is the central component. It is responsible for running the application and providing information about the current state to the application and libraries through its meta-level interface. The runtime system and libraries can also use a meta-level interface provided by the operating system [Kiczales & Lamping 93]. Use of meta-level interfaces are optional and is only required for specialized services that, for example, implement type-independent functionality or utilize special operating system services.

The definition of a runtime system as a virtual machine for the BETA language pins down the contents of the base-level interface as the needs defined by the compiler. However, the runtime system should be designed to put as few restrictions as possible on the ways it can be extended, i.e., it should be minimal in the sense that only functionality that cannot be implemented as libraries are located in the runtime system. In this sense, the runtime system should be similar to an operating system micro-kernel.

3.3 Execution Model

The runtime system in the meta-level architecture is the only executing program seen by the operating system. To execute a compiled BETA program, the runtime system is started from the OS 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 OS executables for each application.

In line with the virtual machine view, the BETA compiler generates code to the BETA virtual machine (i.e., the runtime system) and not to the operating system. The system structure provides a natural mechanism for dynamic linking of code and access to runtime information, which is essential for implementing an efficient meta-level interface (described in Section 4).

The drawback of this approach is that the operating system's linker and loader is not used, which might make linking with code written in other languages, specifically C/C++, more difficult. However, this problem may be less of an issue when interoperable object standards, e.g., CORBA [OMG 91], get in wide-spread use.

4 The Meta-Level Interface

This section briefly describes the current design of a meta-level interface (MLI) for the BETA language. A detailed description of the MLI can be found in [Brandt 95].

The job of the MLI is to reify certain language and runtime system primitives to allow meta-level code to exploit the reified primitives.

Section 4.1 describes the BETA language concept of pattern references, turning BETA patterns into first-class values. Section 4.2 then describes the novel language concept of *attribute references*, turning pattern and object attributes into first-class values. Dynamic pattern and attribute references allow BETA programs to explicitly talk about patterns and attributes. This enables the MLI, described in sections 4.3 – 4.5, to be expressed simply and elegantly. As seen from a BETA program, the MLI is accessible as a globally visible object.

4.1 Pattern References

The BETA equivalent to a *class* is the *pattern*. More precisely, the pattern is a unifying abstraction for concepts such as class, procedure, virtual, exception, coroutine, and generics. A pattern `P` is declared in the following way:

```
P: Super
   (# Decl1; Decl2; ... Decln
   enter In
   do Imp1; Imp2; ... Impm
   exit Out
   #)
```

where `Super` defines an optional super-pattern for `P`; `Decl1`, `Decl2`, `...`, `Decln` is a list of declarations of reference attributes or patterns; `In` defines optional input parameters of `P`-objects; `Imp1`, `Imp2`, `..`, `Impm` is a sequence of imperatives (statements) associated with `P`-objects; and `Out` defines optional output parameters of `P`-objects.

To reify the pattern concept, allowing them among other things to be passed around as first-class values, “standard” BETA includes the notion of *dynamic pattern references*. A dynamic pattern reference may refer different patterns at different points in time, and can be passed as a parameter to method calls. A dynamic pattern reference `P` is declared as follows,

```
P: ##T;
```

where `T` is the *qualification* of `P`. Thus, `P` is allowed to refer only subpatterns of the static qualification `T`.

Given a dynamic pattern reference, it is possible to create instances of the referred pattern, and to check sub-pattern relationships by comparing it with another pattern reference.

4.2 Attribute References

Dynamic pattern references allow programs to explicitly talk about patterns. A BETA language meta-level interface 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 reference referred by a particular attribute reference. Meta-level code thereby becomes more readable, and the compiler will be able to generate efficient, type-safe code. The syntax for the declaration of an attribute reference is:

```
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 super pattern 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[3] (`^`) 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 may be qualified to refer only attributes referring text objects.

Consider the following example:

[3] A dynamic object reference in BETA terminology is the same as a pointer in Pascal terminology.

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

This BETA code 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`, and is therefore qualified to refer any of the family attributes. An example usage is shown below:

```
   aFamily: @Family;
   aPerson: ^Person;
do ...
   Family.mother:->aFamilyPerson;        (* 1 *)
   aFamily.(aFamilyPerson)[]->aPerson[]; (* 2 *)
```

At `1`, `aFamilyPerson` refers the `mother` attribute of the `Family` pattern. Applying `aFamilyPerson` to the family object `aFamily` thus returns a reference to the mother in the family `aFamily`. At `2`, `aPerson` therefore refers the same person as `aFamily.mother`. Note that in this example, all type-checking can be done at compile-time because of the qualifications of the attribute reference.

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.

Attribute references and pattern references allows BETA code to explicitly talk about attributes and patterns. Used as the foundation for a meta-level interface, they allow the MLI itself to be elegantly expressed, and allows the programmer to express static typing constraints even when doing meta-level programming.

The following sections give an overview of the functionality provided by the MLI.

4.3 Introspective Capabilities

The introspective parts of a meta-level interface are the parts allowing a program to look into itself and obtain detailed information on the current runtime state. An introspective interface does not allow the program to perform any changes.

Introspecting Attribute References

Given an attribute reference, `ar`, the BETA MLI allows extraction of the location qualification, the attribute qualification, the attribute kind, and the name of the attribute referred by `ar`. Qualifications are returned in the form of dynamic pattern references.

Introspecting Pattern References

Persistent stores, object browsers, and schema evolution tools, to mention a few, need a way to scan the attributes of an object. In the BETA MLI, this is supported by the ability to scan the attributes of a pattern p[4]. For each attribute of p, an attribute reference, ar, is returned. Information on ar can be retrieved using the functionality described above. For example, the following, simplified[5], part of the MLI provides a way to scan dynamic object reference attributes of a pattern[6]:

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

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

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

[4] Given an object `o`, a reference to the pattern of which `o` is an instance is returned by the BETA expression `o##`.

[5] The real MLI allows subpatterns of `scanDyn` to specialize the qualification of `current`, thus supporting static type-checking of code inside subpatterns of `scanDyn`.

[6] Note that subclassing in BETA is based on specializing semantics, not overriding semantics. 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.

BETA allows pattern nesting to any level, and thus a pattern, in addition to its set of attributes, is defined by the set of objects making up the lexical scope of the pattern. Thus, to allow meta-level code to extract full information on a pattern, the BETA MLI provides a way to extract the set of environment objects from a pattern reference.

The BETA MLI provides methods to save and restore pattern references from a stream. This functionality allows dynamic pattern references to be exchanged between different processes, for example through a persistent store or a distributed object system.

A BETA method object, (or "activation record"), only differ from other objects in the way it is used, and is therefore an instance of some pattern p. Enter and exit parameters are attributes of the method object. The MLI provides a way to scan attributes corresponding to enter and exit parameters. For example, this allows method calls to be dynamically constructed, a functionality needed by, e.g., debuggers and embedded interpreters.

Finally, a pattern reference can be mapped into a representation (abstract syntax tree) of the source code for the pattern[7]. This functionality is for example useful to implement debuggers, source-browsers, and integrated development environments.

Introspecting Call-Stacks

A *component* is a BETA object with a private call-stack. BETA components can be used as either lightweight processes (threads) or coroutines.

To reify the concept of callstack, the MLI treats it as a list of object references and return addresses. Return addresses are reified by the notion of *bus-stops* [Jul 94]. A bus-stop is an abstraction of a code address where the execution of a coroutine can be suspended. Bus-stops define a subset of code-addresses that can be mapped between different address spaces and hardware platforms, thereby allowing callstacks to be moved between different machines on a heterogeneous network.

Given a callstack, it is now possible to scan the contents of the stack by scanning object references and bus-stops. Like pattern references, a bus-stop can be saved and restored from a stream, giving it an external representation.

[7]Manipulation of the source code returned will not, however, have any direct effect on the running program.

4.4 Invocational Capabilities

Invocational capabilities are the parts of a meta-level interface allowing a program to explicitly call functionality that is normally hidden in the runtime system.

An important invocational capability is the ability to create new object instances from dynamic pattern references. As demonstrated by the following code, standard BETA already supports this functionality:

```
       p: ##Object; (* p may refer any pattern *)
       aPerson: ^Person;
    do Person##->p##;
       &p[]->aPerson[];
```

This code creates a `Person` instance from a dynamic pattern reference to the `Person` pattern.

A very important invocational capability concerns dynamic load and link of compiled code. The BETA MLI defines a concept of code objects and functionality to dynamically link, unlink, and replace such code objects. Replacement of code objects will be the basic mechanism supporting the implementation of reflectional capabilities with no cost paid by non-users. The idea is that default code objects do not provide reflectional capabilities. However, if intercessory reflection (as described in the next section) is needed, the necessary code object can be dynamically replaced by objects providing reflectional hooks. To ensure safety, a code-object is only allowed to replace another code-object if the compiler guarantees that they are compatible. Two code-objects are compatible if they differ only with regard to existence of reflectional hooks.

Finally, BETA source code can be handed to the compiler through the meta-level interface. Upon successful compilation, the compiler will return a reference to a code object. When a code object has been linked in, patterns defined by that code object may then be scanned.

4.5 Intercessory Capabilities

Intercessory capabilities allow a running program to change the behaviour of language primitives at runtime. The BETA MLI provides intercession of object creation and destruction, method dispatch, and access to simple attributes.

Concerning object creation, a callback is provided to trace instantiations of some pattern p. Within the callback, creation of a p object can be replaced by the creation of a p' object, provided that p' is a subpattern of p.

BETA is garbage collected, so objects are never explicitly destroyed. The MLI allows programs to trace garbage collection of specific objects, but does not allow garbage collection of an object to be prevented, since this would put unreasonable demands on the garbage collector implementation.

Another important intercessory capability allows programs to intercept method dispatch and simple attribute access. Method dispatch can be changed using *reflectors*. A reflector may be considered a *metapattern*, monitoring execution of other patterns. If a pattern p is monitored by a reflector pattern r, execution of p objects is redirected into execution of a r object.

The super pattern of all reflectors is the `reflector` pattern:

```
reflector:
  (# methodType:< Object;
     m: ^methodType;
  enter m[]
  do INNER
  #);
```

A subpattern of `reflector` can be associated with a pattern by applying the `reflect` method in the MLI:

```
reflect:
  (# p: ##Object;
     refl: ##reflector;
  enter (p##,refl##)
  do ...
  #);
```

As an example, assume that we want to monitor calls to the `mult` method of `calc` objects:

```
calc:
  (# mult:
       (# a,b,c: @Integer;
       enter (a,b)
       do a*b->c
       exit c
       #);
  #);
```

The following code shows how to use a reflector to monitor calls to the `mult` method of any `calc` object.

```
multreflector: MLI.reflector
  (# methodType:: calc.mult;
  do m; (* execute mult *)
     m.c->screen.putint; (* Show mult result *)
  #);
  c: @calculator;
  res: @Integer;
do
   (calc.mult##,multreflector##)->MLI.reflect;
   (* 1 *) (1,2)->&c.mult->res;
```

First, the `multreflector` is associated with the `calc.mult` pattern. Thus, execution of `calc.mult` objects is redirected into execution of a `multreflector`, which as a parameter receives the `calc.mult` object. The result of the multiplication will therefore be printed on the screen whenever an instance of `calc.mult` is executed. Since the `methodType` virtual is further bound to `calc.mult` inside `multreflector`, the multreflector has static knowledge of the layout of the method objects ("activation records") it reflects upon. For example, it is statically known that the method object `m` has the `c` attribute, which is therefore accessed using standard remote attribute access.

In the above example, the `multreflector` reflects execution of all objects qualifying to `calc.mult`, i.e., all objects that represent an invocation of the `mult` method of `calc` instances. Through the concept of *full object qualifications*, the reflector mechanism also supports reflection on the execution of single objects, as opposed to all instances of a specific pattern. Full object qualifications are introduced in [Brandt 95] as part of a generalization of the BETA qualification concept.

As opposed to most other object-oriented programming languages, BETA allows an object to directly access attributes of other objects. This is known as *remote attribute access*. Reflection on remote attribute access is handled by a MLI mechanism similar to the reflector mechanism shown above. However, while method reflection can be implemented by dynamically replacing the implementation of the patterns reflected upon, remote attribute access is more difficult. The reason is that remote attribute access is compiled into direct memory access. Thus, to apply the mechanism of runtime code replacement, we need to keep track of "uses" relationships between patterns. Currently we do not know whether this is practical.

5 Interactive Programming Environment

The preceding sections described the meta-level architecture for the compiled and statically typed programming language BETA. In this section we will focus on how the meta-level architecture can be used to build an interactive programming environment in the language itself.

An interactive programming environment (IPE) allows the user to directly modify and execute source code, and see the effect of the operations immediately. Such environments break the traditional edit/compile/execute cycle, thereby allowing a more exploratory programming style. An IPE typically provides object browsers, code browsers, editors, and debug facilities.

Most IPEs have been built for dynamically typed languages, such as Smalltalk, Self, and CLOS. Building an incremental programming environment for a dynamically typed language is simpler than for a statically typed language, because all type-checks are done at runtime. This enables the turn-around time between editing and execution to be very low. In a statically typed language, the modified source code has to be type-checked before it can be compiled. This increases the turn-around time, but also allows more type errors to be caught at compile-time.

5.1 Employing the Meta-level Architecture

The meta-level architecture allows a clear separation between the run-time system and the IPE. This approach has several benefits: (i) the runtime system and compiler are independent of the IPE, making it possible for third-party vendors to build their own environments and tools; (ii) the environment can export a meta-level interface of its own, allowing user customization to be written in BETA; and (iii) the programmer can easily move between application development using the IPE, using traditional commandline tools, and stand-alone execution of the application.

The BETA language extended with the meta-level interface contains almost all the needed functionality for building such an environment.

- **Object browsers.** The introspective capabilities of the meta-level interface makes all necessary information available at runtime.
- **Direct execution of code.** Pattern references and attribute references make it possible to interactively execute patterns and assign to variables.
- **Breakpoints.** By using the reflector mechanism, breakpoints can be inserted on each object invocation, creation, or destruction. Code breakpoints can be provided by generating code that contains debugger hooks and dynamically linking it in.

5.2 Dynamic Linking

The meta-level interface provides an abstraction of compiled code: code objects. They are the granularity of the dynamic link mechanism supported by the runtime system. Dynamic linking of code at run-time is an essential component for implementing an IPE. It supports incremental linking of code, reducing the turn-around time between editing and executing.

To facilitate fine-grained dynamic linking, code will no longer be organized in traditional object code files, but will be stored as BETA objects in a persistent store. Notice that this design corresponds with the virtual machine view of the runtime system where executable code is viewed as data. Storing code as objects in a persistent store decouples the granularity of linking from the organization of the source files. Furthermore, auxiliary data structures facilitating, for example, incremental type-checking can be easily stored using the persistent store mechanism as well.

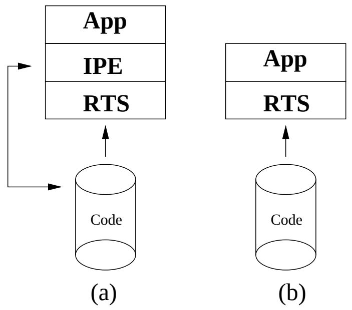

Figure 3: Runtime organization (a) with the IPE and (b) as a stand-alone application.

Figure 3a shows how the IPE is built on top of the runtime system. It implements browsing, editing, and debug facilities, by using the runtime system's meta-level interface. The application is dynamically loaded and executed by the IPE. Figure 3b shows how the same application can, without any modifications, be dynamically loaded and executed by the runtime system, i.e., run as a stand-alone application.

5.3 Incremental Compilation

The dynamic linking mechanism provided by the runtime system provides the basis for implementing incremental compilation. It directly supports extensibility [Malhotra 94], i.e., addition of new subclasses at runtime. However, arbitrary changes of source-code introduces several issues that have not been discussed, for example, incremental type-checking, and how old instances of a modified pattern are updated/removed.

In order to provide a general meta-level interface for incremental compilation, experience is needed. The current design supports extensibility, limited code-modification (insertion of reflection hooks), and strong introspective capabilities, which provides enough functionality for construction of a prototype implementation.

6 Related Work

CLOS [Kiczales et al. 91] includes a comprehensive meta-level interface, (the CLOS MOP, or metaobject protocol), which allows the meta-level programmer to inspect and change several primitives of the basic programming language, including redefiniton of slot access, multiple inheritance semantics, and replacement of meta-classes. Much functionality of the BETA MLI was inspired by the functionality provided by the CLOS MOP. However, BETA is statically typed, whereas CLOS is dynamically typed, leading to a considerably different style of meta-level interface. Most notably, the BETA MLI allows, but does not force, the meta-level programmer to express static typing constraints, thereby leading to more readable and possibly more efficient meta-level code.

ABCL/R [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 meta-level interfaces of these languages, as compared to the BETA MLI, is the issue of static versus dynamic typing. Dynamic typing means that method lookup in ABCL/R, AL-1/D, and Self is basically implemented by a runtime table-lookup. In BETA, static type information allows procedure calls to be implemented either by direct subroutine calls, or by using a single-level indirection jump through the virtual table of the target object. In both cases, the destination address is resolved at link time. In order to retain efficiency, the implementation of the BETA MLI therefore uses dynamic replacement of code objects and dynamic linking techniques.

OpenC++'s [Chiba 93] concept of *reflect methods* was the direct inspiration for the `reflector` pattern of the BETA MLI. However, OpenC++ does not have introspective capabilities, and reflect methods must be known at compile time. The BETA design aims at larger orthogonality, i.e., it allows reflection on patterns that were written without reflection in mind.

The C++ concept of *pointers to members* may be considered a restricted version of attribute references. However, the absence of runtime type information and restriction to function members, renders C++ pointers to members considerably less useful than attribute references.

Octopus [Farkas & Dearle 93] provides a reflective language mechanism for generic manipulation of complex object structures. Like the BETA MLI, the Octopus mechanism enforces static typing constraints, but does not allow the meta-level programmer to express such constraints. Furthermore, Octopus provides only limited intercessory capabilities, through replacement of function pointers in singular objects.

Interactive programming environments has, as already mentioned, mainly been built for dynamically typed languages, such as Smalltalk, Self, and CLOS. IPEs have also been constructed for statically typed languages, such as Simula [Hedin & Magnusson 87] and Trellis/Owl [O'Brien et al. 87]. These environments support incremental compilation as well as incremental execution. Trellis/Owl uses a persistent store for storing source-code and compiled code, which is similar to the architecture presented for BETA. However, there is no clear separation between the runtime system and the IPE, in contrast to BETA where the IPE is built on top of the runtime system.

7 Status and Conclusion

This paper described a meta-level interface for a compiled and statically typed programming language. To the best of our knowledge, BETA is the first compiled and statically typed language for which a comprehensive meta-level interface, including introspective as well as intercessory capabilities, has been designed.

Constructing a meta-level interface for a compiled and statically typed language introduces several problems not present for dynamically typed languages, such as CLOS. The main contributions of this paper are: (i) the structuring of a meta-level architecture for a compiled language, (ii) the introduction of the *attribute reference* that allows the design of a typed meta-level interface, and (iii) a separation of the meta-level architecture from the incremental programming environment, allowing the programming environment to be built on top of the meta-level architecture.

The meta-level architecture defines an entity to provide the meta-level interface for a compiled language, and makes it possible to implement it efficiently. The reflectional capabilities are implemented with no overhead for non-reflective code by dynamic replacement of code objects at runtime, so only users of reflectional capabilities will use code objects with reflectional hooks.

The meta-level interface was designed on top of two language primitives: *dynamic pattern references* and *attribute references*. These two language concepts make classes and instance variables first-class values. Building on top of these concepts, it was possible to design a full-blown meta-level interface with support for the expression of static typing constraints. Furthermore, the concept of attribute references is a natural extension of the BETA language, enabling programs to explicitly talk about attributes of patterns and objects.

Experience gained from the implementation of numerous applications has guided the design of the meta-level interface. Being designed with several real-world applications in mind ensures the usability of the meta-level interface. Implementation work has just been initiated, hopefully leading to valuable experience on the implementation of metao-level interfaces for statically typed languages.

Acknowledgements

We would like to thank Ole Lehrmann Madsen, Jørgen Lindskov Knudsen, Olivier Danvy, Alexandre Sousa, and the anonymous referees at the META'95 workshop who read earlier versions of this paper and provided many valuable comments.

References

[Brandt & Madsen 94] S. Brandt and O. 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, 1994.

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

[Brandt 95] S. Brandt. Reflection in a Statically Typed and Object Oriented Language – A Meta-Level Interface for BETA. Technical report, Department of Computer Science, University of Aarhus, 1995.

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

[Farkas & Dearle 93] A. Farkas and A. Dearle. Octopus: A Reflective Language Mechanism for Object Manipulation. In *Proceedings of the Fourth International Workshop on Database Programming Languages.* Springer-Verlag, 1993.

[Goldberg & Robson 83] A. Goldberg and D. Robson. *Smalltalk-80. The Language and its Implementation.* Addison-Wesley, Publishing Company, Reading, MA, 1983.

[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, Feb 1994.

[Hedin & Magnusson 87] G. Hedin and B. Magnusson. Supporting exploratory programming in Simula. In *Proceedings of the Fifteenth SIMULA Conference*, pages 73 – 88, Jersey, September 1987.

[Hölzle & Ungar 94] U. Hölzle and D. Ungar. A Third Generation Self Implementation: Reconciling Responsiveness with Performance. In *OOPSLA'94 Conference Proceedings*, pages 229 – 243, October 1994.

[Jul 94] E. Jul. Private conversation regarding work on heterogeneous Emerald, May 1994.

[Kiczales & Lamping 93] G. Kiczales and J. Lamping. Operating Systems: Why Object-Oriented? In *Proceedings of IWOOOS'93*, 1993.

[Kiczales 92] G. Kiczales. Towards a new model of Abstraction in Software Engineering. In A. Yonezawa and B. C. Smith, editors, *Proceedings of IMSA'92: International Workshop on Reflection and Metalevel Architecture*, 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.

[Madsen et al. 93] O. Madsen, B. MøllerPedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley, June 1993.

[Malhotra 94] J. Malhotra. *Tailorable Systems: Design, Support, Techniques, and Applications. (Daimi PB-466).* PhD dissertation, Department of Computer Science, University of Aarhus, January 1994.

[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, *OOPSLA'92 Conference Proceedings*, pages 127 – 144. ACM Press, October 1992.

[O'Brien et al. 87] P. D. O'Brien, D. C. Halbert, and M. F. Kilian. The Trellis Programming Environment. In *OOPSLA'87 Conference Proceedings*, October 1987.

[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.

[OMG 91] OMG. *The Common Object Request Broker Architecture and Specification*, Dec 1991. Document number 91.12.1, Revision 1.1.

[Rao 91] R. Rao. Implementational reflection in Silica. In P. America, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 251–267. Springer-Verlag, 1991.

[Sloth & Hem 93] L. Sloth and J. Hem. Object Oriented Model for the Distributed Object Oriented Database. Technical report, EuroCODE, Workpackage WP2 - Deliverable D2.2, 1993.