

Flexible Language Interoperability

Torbjörn EkmanProgramming Tools Group
University of Oxfordtorbjorn@comlab.ox.ac.uk**Peter Mechlenborg**

Mu Aps

peter@mu.dk**Ulrik Pagh Schultz**Maersk Institute
Univ. of South Denmarkups@mmmi.sdu.dk

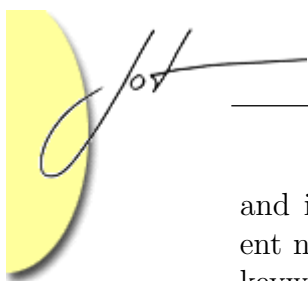
Virtual machines raise the abstraction level of the execution environment at the cost of restricting the set of supported languages. Moreover, the ability of a language implementation to integrate with other languages hosted on the same virtual machine typically constrains the features of the language. In this paper, we present a highly flexible yet efficient approach to hosting multiple programming languages on an object-oriented virtual machine. Our approach is based on extending the interface of each class with language-specific wrapper methods, offering each language a tailored view of a given class. This approach can be deployed both on a statically typed virtual machine, such as the JVM, and on a dynamic virtual machine, such as a Smalltalk virtual machine.

We have implemented our approach to language interoperability on top of a prototype virtual machine for embedded systems based on the Smalltalk object model, which provides interoperability for embedded versions of the Smalltalk, Java, and BETA programming languages.

1 INTRODUCTION

Virtual machines are at the foundation of modern, object-oriented languages such as Java and C#. Virtual machines raise the level of abstraction and enable dynamic features such as code mobility, which is central to the emerging trend of pervasive computing. The virtual machines designed to support Java and C# (JVM and CLR) have significant differences, but are nonetheless designed to support very similar languages with similar object models [3, 13]. Although only the CLR is designed to support multiple programming languages, both virtual machines have been targeted by a multitude of languages [3, 8, 1, 6, 16]. Languages that compile to the instruction set and object model of the target virtual machine have obvious advantages compared to interpretation: apart from an order of magnitude in terms of performance, compilation is also essential for enabling two-way interoperability with other languages, e.g., the ability not only to consume classes implemented in other languages but also to produce classes that can be used from other languages.

The mapping from language to virtual machine must support the semantics of the language and is normally designed to produce code that is as efficient as possible. Nevertheless, this same mapping is critical for interoperability with objects implemented in other languages. Naming is a simple example of a potential conflict,



and is a fundamentally challenging issue when considering languages with different naming schemes, such as Java with type-based overloading and Smalltalk with keyword-based selectors. To enable programming to an interface independently of the implementation language, naming must somehow be transparent across languages, which constrains the mapping of each language to the virtual machine. Differences in type systems are also dependent on the language mapping. For example, preserving static typing in a Java object interacting with Smalltalk requires dynamic checking of parameter and return types. For the sake of performance, it is critical that such dynamic checking is done only when required, that is, when crossing language barriers. In general, language implementors are faced with a fundamental choice: either obey some common language mapping, potentially constraining the language implementation in terms of features or efficiency, or use a dedicated language implementation, obscuring the appearance of functionality made available to other languages.

We have developed a highly flexible approach to multi-language virtual machines based on run-time or compile-time extension of class interfaces to support integration between heterogeneous languages. Language integration is achieved without enforcing a common implementation-level language subset, by enabling components to exchange implementation requirements using a protocol based on shared, language-independent interface definitions. Our system thus provides a very high degree of freedom for the language implementor while preserving component encapsulation and ensuring full interoperability between classes implemented in different languages. The contributions of our work are as follows.

- Simple and efficient approach to language interoperability based on run-time or compile-time extension of method tables to accommodate an open set of programming languages.
- Rich cross-language interaction including cross-language inheritance and run-time type checking for preserving static typing invariants.
- Experimental demonstration of the feasibility of our interoperability approach using realistic implementations of the Smalltalk, Java and BETA programming languages.

We have implemented our approach to language interoperability on top of a prototype dynamic virtual machine for embedded devices. This virtual machine currently supports embedded versions of the Smalltalk, Java, and BETA programming languages.

Motivation

Our work has been conducted in the context of the Palpable Computing (PalCom) project (www.ist-palcom.org) which concerns, among other things, the development



```

// Socket class - in Java
public class Socket {
    public send(byte[] data, String host, int port) { ... }
    ...
}

"Connection class for encapsulating address - in Smalltalk"
Connection = (
    | host port msgService |
    init: host port: n = ( hostname := host. port := n )
    send: bytes = ( msgService send: bytes host: hostname port: port )
    setMessageService: service = ( msgService := service )
    ...
)

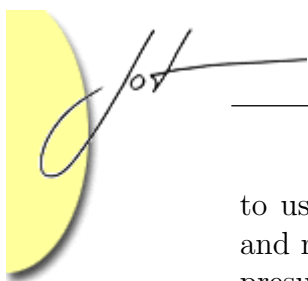
(* Socket wrapper for redirecting a message - in BETA *)
MessageRedirector (#
    sink: ^Socket; host: @text; port: @integer;
    send: (# _host: @text; _port: @integer; data: [0] @int8u
        enter (data,_hostname,_port) do (data,hostname,port) -> sink.send
    #)
#)

```

Figure 1: Network communication in multiple languages?

of a virtual machine for embedded systems. This virtual machine, named PalVM, is based on a Smalltalk object model, but also provides support for the Java and Beta programming languages [9, 4, 14]. Software is currently being developed in all three languages, but language integration is ad-hoc based on language-specific foreign function interfaces that result in a strong coupling between implementations otherwise isolated by interfaces.

As an example, consider the classes shown in Figure 1. The Java class `Socket` provides a basic means of communication using the UDP protocol. We wish to use this class from Smalltalk, in the implementation of the class `Connection` which encapsulates a remote destination. Both classes are compiled to the same virtual machine instruction set, and both compilers use similar calling conventions, but this integration is nevertheless non-trivial. The naming conventions between Java and Smalltalk are incompatible, and runtime type checking is required to preserve static type checking within Java. As a starting point, we could modify the implementation of `Connection` to use the Java naming convention when calling methods; to support overloading, the Java compiler embeds types into the selector name. Moreover, we would probably need to convert the Smalltalk `ByteArray` object given as an argument to the Java native representation. Suppose however that we also wish

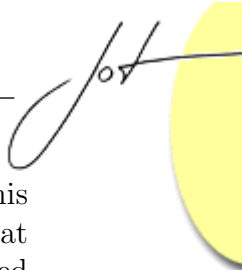


to use the class `MessageRedirector` implemented in BETA, which wraps a socket and redirects messages to a new address. Following the same path, this class must presumably now also use the Java naming convention to be compatible with the existing implementation. This renaming is obviously contagious and detrimental to defining a clean API using the proper naming conventions of each language (e.g., selectors in Smalltalk versus type-based overloading in Java). The Java naming conventions cannot be used globally because they are incompatible with Smalltalk and BETA (the former has no types while the latter for example allows multiple return values). Furthermore, this approach makes it difficult to reuse code, e.g., replacing the `Socket` class by another class implemented in a different language.

As an alternative to manually implementing glue code for integrating implementations across languages, we have implemented a language interoperability framework which allows classes such as the ones shown in Figure 1 to interoperate. More generally, in the PalCom project, software is developed in different programming languages and deployed as fine-grained components. Exchangeability of components is essential, as is runtime efficiency due to limited resources of embedded systems. In general, we require the following properties from a solution: (1) *Sharing* of classes and object instances by supporting method invocation and inheritance between different languages. (2) *Insulation* for method calls that cross languages to allow run-time checks of parameters and return values as well as conversion of value objects. (3) *Scalability* to a large, open set of languages that are not aware of each other. (4) *Flexible naming* to make classes written in one language natural to use from other languages. (5) *Interchangeability* of a class with classes written in other languages but respecting the same interface. Moreover, since we work in an embedded setting, efficiency in terms of memory usage and execution overhead is essential. We note that the .NET approach fails on items (2) and (4) (see Section 6 for a discussion), and that item (5) is essential for properly supporting polymorphism in a multi-language setting.

2 LANGUAGE INTEROPERABILITY

Object-oriented languages vary significantly in terms of basic features such as inheritance (single, multiple, mixins, ...), how objects are created (constructors or factory methods), typing system and so on. Nevertheless, it is our hypothesis that most object-oriented languages can be compiled to a given virtual machine so long as it does not enforce a strict coupling between interface and implementation. Examples of such virtual machines are the JVM and the CLR (where interfaces can be implemented by any class containing the required methods) but also any dynamically typed virtual machine (here the concept of an interface is completely dynamic). Naturally, certain features such as dynamic typing lends itself best to a dynamically typed virtual machine; we return to this issue later, but basically it is mostly independent of our approach. In general, given a set of object-oriented languages that compile to a given virtual machine, our goal is to enable basic interoperabil-



ity between these languages, in the form of message passing and inheritance. This basic interoperability scheme is characterized in terms of a basic object model that dictates both what a language must provide to be accessible to classes implemented in other languages and also what features a language can consume from classes implemented in other languages.

On a virtual machine where different languages are compiled to the same basic object model, interoperability becomes an issue of *duck typing*: “if it walks like a duck and it talks like a duck then it is a duck.”¹ In other words, if objects implemented in one language can be made to behave as objects implemented in another language, then they can be considered to be objects implemented in that language. Thus, the complexity of supporting a very variable set of language features basically boils down to having an interoperable naming scheme. On a typed virtual machine such as the JVM or the CLR, interfaces are used to express this common method naming scheme at the virtual machine level. Nevertheless, fixing a specific naming scheme limits compiler writers and impedes enriching cross-language interaction with features such as insulation. Rather, our approach relies on independent naming schemes for each language based on a symbolic, language-independent Interface Description Language (IDL) description of classes.

The issue of how to support different basic libraries, thread models etc. is not considered relevant for our work: memory constraints dictate that a single library must be used, but it is nevertheless highly relevant to make this library convenient to use from each language supported by the virtual machine.

Basic example

As an introductory example of our approach to language integration, consider a Smalltalk virtual machine targeted by Smalltalk and Java (e.g., the PalVM virtual machine). Java basically compiles to this dynamically typed virtual machine by encoding type names into selector names, e.g., the signature of the Java method “`String substring(int index)`” can be encoded as “`substring.String_int:`”. The unit of deployment supported by this virtual machine is a component, which encapsulates a set of classes. In the context of our work, the classes contained within a component must all be implemented in the same language, but can export interfaces using a language-independent notation (e.g., an IDL).

The Smalltalk class `Point` shown in Figure 2, left trivially compiles to this virtual machine. This class defines x and y coordinates, accessor methods, and a textual display method. The class `Point` is contained within the component `Geometry` (which is implemented in Smalltalk). The class is made accessible to other languages through the component IDL interface `graphics` shown in Figure 2, right. This IDL interface defines the operations of the class including the types of the arguments and

¹The term seems to have originated in Ruby circles by Dave Thomas: <http://www.rubygarden.org/ruby?DuckTyping>.

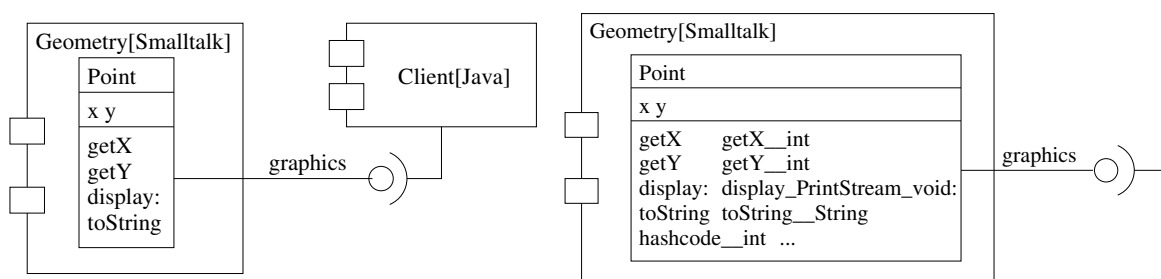
```

Point = (
  | x y |
  getX = ( ^x )
  getY = ( ^y )
  reset = ( x:=0. y:=0 )
  display: stream = (
    stream println: (self toString)
  )
  toString = ( ^('<' + x + ', ' + y + '>') )
)

interface graphics {
  class Point {
    int getX();
    int getY();
    void reset();
    void display(PrintStream stream);
    String toString();
  }
}

```

Figure 2: Point class in Smalltalk and interface in IDL

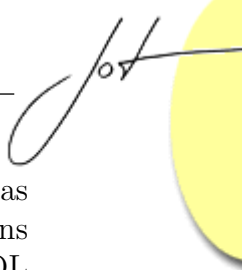
Figure 3: The class `Point` before and after linking with a Java component

the return value. (For this example we assume that the types `int`, `String`, and `PrintStream` are identical in all languages supported by the virtual machine.) By implementing the `graphics` IDL interface, we say that `Geometry` becomes a producer of `graphics`.

Cross-language message passing

Let us now assume that the IDL interface `graphics` is consumed by another component named `Client` implemented in Java, as shown in Figure 3, left. This figure shows the component `Geometry` being used by the component `Client`, but prior to *IDL interface linking*; interface linking is performed when a component is resolved (which is at runtime on PalVM). After interface linking, the class `Point` contains a number of new Java support methods that are inserted into the class by the component loader during interface linking, as shown in Figure 3, right.

For each Smalltalk method defined by the IDL interface, a wrapper method is introduced into the class `Point`, using the naming convention employed by the Java compiler. There are two different kinds of wrapper methods: *alias methods* for providing Java-specific names for Smalltalk implementation methods and *insulation methods* for performing Java-specific processing around Smalltalk implementation methods. Alias methods can be implemented using a standard wrapper method, but on PalVM an alias method is implemented as an extra entry into the method table,



pointing to an implementation method in the same class. Such VM-supported alias methods provide a lightweight approach to decoupling language implementations without requiring dynamic generation of bytecode. As an example, consider the IDL method `reset` (see figure 2) which Java expects to have the name “`reset_void`” (the Java naming convention embeds argument and return types into the name.) This dependency is handled by the component loader during interface linking by introducing an alias into the concrete Smalltalk class `Point`:

```
reset_void_ = alias: reset
```

This alias makes the concrete Smalltalk method available using the Java naming convention. (We here use a Smalltalk pseudo-syntax to indicate aliases.)

Insulation methods

Insulation methods allow arbitrary code to be executed when crossing language barriers. This can for example be used for checking the parameter types before a method call proceeds to the original method in the producer, and checking the type of the return value before returning to the consumer. As an example, consider the IDL method `getX` (see Figure 2) in `graphics`. Since Java is statically typed, the type of the return value from `getX` should be checked. Therefore a consumer insulation method is inserted into `Point`, with the name `getX_int_`, which is defined as follows:

```
getX_int_ = (
  ^IDLJava checktype: #int for: (self getX_int_alias_)
)
getX_int_alias_ = alias: getX
```

This method uses a static method from the class `IDLJava` to check the return type of the method. The alias is used to bind the Java naming convention to the Smalltalk naming convention without requiring dynamic code generation (the insulation method can then be statically generated in advance by the Java compiler). Using a similar approach, producer insulation methods can check the types of parameters when calling Java code from another language.

Cross-language inheritance

A class described by an IDL interface can be subclassed by a class from a different component, implemented in another language. This also makes the subclass a consumer of the superclass. Figure 4, left, shows the class `ColorPoint` (defined in the component `ColorCanvas`) which subclasses `Point`. This class introduces an additional field for representing the color, and overrides the IDL method `toString`. The overriding method is introduced into `ColorPoint` under the Java-based name “`toString_String`”. Calling the method `display` on this object must cause the newly defined method to be subsequently called, and not `toString` defined in `Point`, e.g. `Point` should become a consumer of `ColorPoint`, regarding `toString`. The effect

```

class ColorPoint extends Point {
  int pixel; // RGB and alpha
  int getColor(){ return pixel; }

  String toString(){
    return "<ColorPoint x="
      +this.getX()+" y="
      +this.getY()+" color="
      +this.getColor()+">";
  }
}

```

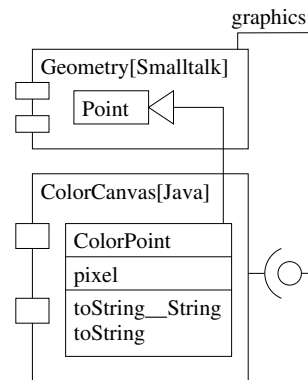


Figure 4: The class `ColorPoint` which subclasses the class `Point`

of making `Point` a consumer of `ColorPoint` is that a wrapper method for `toString` is inserted into `ColorPoint`, thereby overriding the original Smalltalk implementation. The resulting class can now be used from Java. In general, we refer to the set of methods that all correspond to a single IDL method as a *method group*. Subclasses should always override all methods in a method group.

A common object model

Our approach enables integration of components according to their IDL declarations. The IDL defines the *PalCom Common Type System* (PCTS) which is the common object model that the different programming languages in PalCom use to communicate. To be interoperable with other languages, the features of a language must thus be expressible in terms of the PCTS. We have chosen a minimal common type system but with both interface and implementation inheritance, allowing us to express both interface implementation and code reuse across languages (the latter is essential for supporting frameworks). The choice of a minimal type system simplifies the overall system and in particular simplifies the consumer role (because there are fewer features with which to integrate) at the cost of complicating the producer role (because there are fewer features to use when expressing the produced entity).

The PCTS has a common superclass `Object` and a set of primitive types which are represented by the corresponding Smalltalk classes, including strings, and integers. Based on these primitive types, concrete classes and interface classes can be declared. Similarly to Java, there is single inheritance for concrete classes and multiple inheritance for interface classes. A class contains a number of named operations with named and typed parameters and a typed return value. Concrete classes can moreover contain static operations. Parameter and return types are required to be invariant across subclasses. Constructors can be specified for a concrete class (constructors have the reserved name `init` in the IDL). There is no concept of fields, so accessor methods must always be used to access fields across languages.



The PCTS type `Object` maps to the Smalltalk class `Object` which is the common superclass of all object instances in PalVM. Parameters or return values of the type `Object` must thus either be converted to a local type or the language implementation must accept arbitrary foreign objects from other languages. Objects can be wrapped inside a proxy which provides reflective access to the foreign object. Alternatively, language support methods, described in Section 3, can be used to adapt arbitrary objects from other programming languages to make them compatible with the language implementation.

3 FLEXIBLE LANGUAGE INTEROPERABILITY FRAMEWORK

Our approach to language interoperability is implemented in the *flexible language interoperability framework* (FLIF). We now define the FLIF algorithms for supporting cross-language method invocation and cross-language inheritance. Details regarding our implementation are covered later.

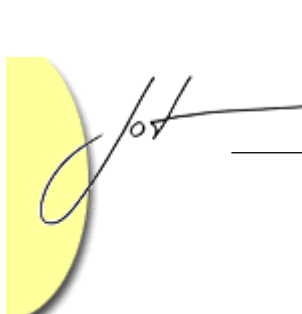
We note that although our approach to language interoperability is expressed in terms of classes organized into components, a component model is not essential. The specific rôle of a component is to aggregate classes into a unit that is treated uniformly, and to provide information about the set of interfaces provided by the classes contained in the component. Thus, the component model need not be supported by the virtual machine, but can simply be implemented by a mapping from classes to their IDL interfaces and implementation language.

Cross-language method invocation

A component encapsulates a set of classes and contains interface export and import annotations as well as an annotation identifying the language-specific name mapping used in the implementation of the component. When the component is loaded, its classes are processed by FLIF, which manipulates the representation of the classes according to their relations to components implemented in other programming languages. In more detail, a component may export selected classes through an IDL interface, which makes the classes available to other components implemented in any language supporting FLIF.

Notation. We use $X: L = (A_1; \dots; A_n)$ to denote a component X implemented in the language L with annotations A_i . In this context, the relevant annotations are import and export. An import annotation `import I from X` specifies consumption of the interface I from the component X . Interfaces are specified in an interface definition language (IDL), but may be automatically generated. An export annotation

`export I bind (C0, C'0), ..., (Cn, C'k)`



```

class Y:C' {
  pi(P) = ...
}
⇒
class Y:C' {
  pi(P) = { ... }
  piLY(P) = { CYP(P); pi(P) }
  piLXA(P) = alias: piLY(P)
  piLX(P) = { CXR(piLXA(P)) }
}

```

Figure 5: Enabling cross-language method calls

specifies production of IDL interface I binding each IDL class C_i in IDL interface I to a concrete class C'_i . An IDL interface defines concrete IDL classes C_i and (Java-style) IDL class interfaces C_i^I , as follows:

$$I = (C_1, \dots, C_n) \oplus (C_1^I, \dots, C_m^I)$$

An IDL class simply defines a set of operations: $C = (p_1, \dots, p_r)$.

The operation names in IDL classes are abstract in the sense that each language has a specific mapping from IDL method names to concrete method names such that any IDL operation name can be mapped to a unique concrete method name. The concrete method names generated from the IDL for a given language must be equivalent to the method implementation names generated from source code written in this language, so that implementation language neutrality is retained (e.g., IDL-generated names are equivalent to native names).

A producer component contains a set of classes and an annotation. The component annotation contains both the IDL interfaces and a binding from IDL classes to concrete classes of the component; the concrete class can either be defined locally or in a class required by the component. Consumer components contain a set of annotations describing which IDL interfaces the component depends on.

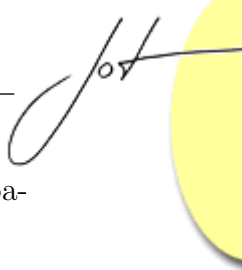
When a component is resolved by the virtual machine, its dependencies on other components are also resolved. In particular, IDL-defined dependencies on other components are resolved, as follows. A component X written in language L_X may import IDL interfaces from any number of components. Let Y be one such component which exports the IDL interface I :

```

I = (C1, ..., Cn)
Ci = (pi1, ..., pim)
Y: LY = (export I bind [(C1, C'1), ..., (Cn, C'n)]; ...)
X: LX = (import I from Y; ...)

```

For each IDL class C in the interface I , let C be implemented by the concrete class C' written in the language L_Y and nested in the component Y . The processing of an operation p_i of class C' taking parameters P is illustrated in Figure 5, and consists of three steps that modify C' :



1. Insertion of a producer wrapper method p_iL_Y ; the function C_Y^P performs parameter insulation specific to the language L_Y .
2. Insertion of a consumer wrapper method p_iL_X which calls the producer wrapper method through an alias p_iL_XA ; the function C_X^R performs return value insulation specific to the language L_X . The alias is used to decouple the implementation of the consumer wrapper from the producer language.
3. Insertion of the alias p_iL_XA linking p_iL_X to p_iL_Y .

It is the responsibility of the consumer wrapper method to call p_iL_XA . If insulation is not required for a given method, a direct alias is used instead as an optimization. On a statically typed virtual machine, an interface containing these language-specific methods would also be added to the class.

The class modifications are performed by the FLIF framework using a language-specific name mapper object obtained from the annotation on each component, as described in Section 5. The language-specific name mapper object is also responsible for introducing language-support methods into classes when appropriate, for example on PalVM to allow a Smalltalk object to be used as a Java object (e.g., be the equivalent of an instance of `java.lang.Object`).

The overall effect of this process is that wrapper methods are created for each method in each IDL class, and language support for the language of the consumer is established in the producer. After IDL dependencies have been resolved, simple cross-language method invocation is supported from the consumer component to the producer component. Insulation methods are named according to the calling conventions of other languages, and hence are never used from consumers implemented in the same language.

Cross-language object instantiation

A concrete IDL class can be instantiated as an object from any consumer language; it is up to the concrete language to represent the constructor. Concretely, our Java mapping uses standard Java constructors, whereas an instance method named `init` is generated for Smalltalk and Beta (neither of which have an explicit notion of constructors). Constructor chaining as known from Java is supported but not enforced between languages, which means calling the superclass constructor is up to the programmer if not enforced by the compiler. If no constructor is specified in the IDL, it is up to each language mapping to generate a default constructor.

Cross-language implementation inheritance

IDL class inheritance allows a subclass to not only extend the superclass with additional state and behavior but also to override existing behavior. The PCTS object

model considers all fields private to the object, e.g., the fields of an object can only be accessed from within methods that have been invoked on this object. Thus, cross-language interoperability is indifferent with regards to adding new fields in a subclass. New methods introduced in a subclass may need to call methods from the superclass, which is supported by making the class a consumer of its superclass. In effect, we use a black-box approach to inheritance where a consumer class can only see those properties of the superclass declared in the IDL interface.

Overriding existing methods from the superclass means that all consumers of the superclass should also be able to use the overridden method. Conceptually, the subclass overrides the method from the IDL class. In more detail, the entire method group (the concrete method from the superclass and all of its consumer wrapper methods) must be overridden. This overriding is implemented by introducing wrapper methods for the overriding method into the subclass, one for each of the languages used by the consumers of the superclass.

Concretely, consider a component Y exporting the interface I . The interface I is imported by a number of components, Z_1 to Z_n , written in different languages. The method group of operation p in C then consists of the original method p' in C' and wrapper methods $p'_{Z_1}, \dots, p'_{Z_n}$ generated for the Z_i components:

```
I = (C, ...) ⊕ (...)
Y: LY = (export I bind [(C, C'), ...])
Zj: LZj = (import I from Y, ...)
```

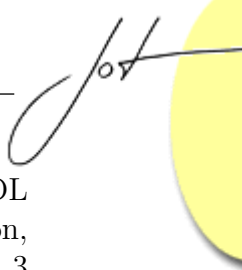
Now let X be a component that imports Y but also defines a subclass D of C that overrides p with p_D . The overriding method p_D should be callable from Y and all consumers of Y . This property is ensured by making all the consumers of Y , and Y itself, into consumers of X , regarding p . Following the algorithm described in Section 3 ensures that wrapper methods p' , $p'_{Z_1}, \dots, p'_{Z_n}$ will be inserted into D , and as a result, all consumers of Y and Y itself will be able to call p_D .

Cross-language interface inheritance

In addition to IDL classes, the PCTS also supports IDL class interfaces. Supporting subclassing of IDL class interfaces is very similar to supporting ordinary subclassing, as described in the previous section. The difference is that IDL class interfaces are an abstract construct, and therefore do not have a direct concrete counterpart. Consequently, the only producers of IDL class interfaces are concrete classes that implement the IDL class interface.

Analogously to the description given in the previous subsection, consider the following definitions:

```
I = (...) ⊕ (CI, ...)
Y: LY = (export I bind [...])
Zj: LZj = (import I from Y, ...)
```



As before, the component `X` imports `Y`, but the class `D` is now a subclass of the IDL class interface `CI`. Being a subclass of `CI`, `D` is also a producer of `CI`. For this reason, all consumers of `CI` are also consumers of `D`. Following the algorithm described in 3 ensures that `D` is usable from all languages that contains consumers of `CI`.

4 LANGUAGE INTEGRATION EXAMPLES

We now describe how language integration works for the PalVM versions of Smalltalk, Java and BETA.

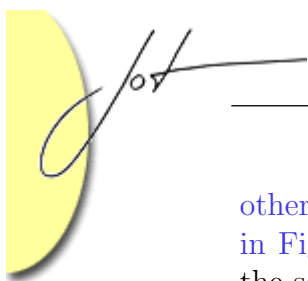
Flexible language integration for Smalltalk

Making Smalltalk a consumer in FLIF is done as follows. Concrete IDL classes in PCTS can be mapped directly to Smalltalk classes, as described in Section 3. As for class interfaces, since Smalltalk does not have an interface construct, implementing an IDL class interface reduces to ensuring that all operations in the IDL class interface are also present in the class that is implementing the IDL class interface. When acting as producer, Smalltalk blocks are mapped to an interface with a single operation named `value`.

While an untyped language can consume IDL interfaces produced by both typed and untyped languages, it is difficult for typed languages to consume IDL interfaces produced by untyped languages. The only type guarantee that an untyped language gives is that every value has the type `Object`. To enable automatic generation of more precise interfaces, type information can be annotated on method declarations and used to generate IDL interfaces produced by Smalltalk. This *optional typing* [7] annotation is not checked by the compiler and has no effect on the runtime semantics, but is included in the component annotation. In effect, the annotation provides a structured means for the programmer to document the interface of each Smalltalk class, but is also used when integrating Smalltalk with other languages.

Flexible language integration for Java

Java packages are used to denote components, meaning that each Java package is compiled to a separate component. The Java compiler automatically integrates with FLIF both as a consumer and as a producer. Insulation methods are used to ensure static typing of Java code; for a given component the Java compiler generates consumer and producer insulation methods stored in a separate class. IDL classes and IDL class interfaces are automatically imported from IDL descriptions in components during interface linking. An IDL interface is visible as a package in Java and all types defined in that interface are members of that same package. The Java front-end takes these interfaces into account during name binding and type checking and can thus statically type-check Java applications that consume classes written in



other languages. Full static semantic analysis can thus be done for the Java example in Figure 3, even though the superclass is implemented in Smalltalk. To perform the semantic checking, the compiler automatically reads the description of the IDL interface `graphics`.

The compiler automatically produces IDL interfaces for Java classes. A Java component will thus have an IDL interface generated for each package. Only language elements that are supported by PCTS are exported, e.g., nested classes are not included. From a consumer point of view, the developer need not be aware of the IDL at all since it integrates transparently with Java. When acting as a producer, the developer needs to ensure that APIs to classes that are intended to be used by other languages can be expressed in IDL and the PCTS.

Flexible language integration for BETA

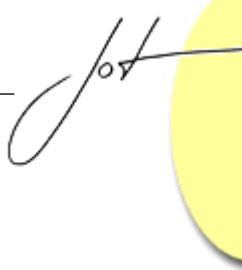
For the larger part, the BETA language integrates straightforwardly with the FLIF on PalVM. When BETA plays the role of consumer, a BETA external class pattern is used to describe concrete IDL classes. Here, a single pattern represents the class and encapsulates a pattern for each method. BETA however does not support multiple inheritance nor Java-style interfaces, so IDL class interfaces are represented separately from the class hierarchy. Such interface types are obtained using special cast operations introduced into all concrete classes.

When BETA plays the role of producer, only patterns nested directly within the “program” pattern can be used from other languages (this corresponds roughly to plain, non-inner classes in a Java program). Such a pattern P must contain a nested pattern P' for each method declared in the IDL interface being produced; the nested patterns P'_i can then be used as methods from other languages. When an instance of the concrete IDL class bound to P is being created, the origin is implicitly passed as an argument by a custom constructor introduced into P (this is possible because of the restriction to a unique origin reference, the program pattern instance). Nested patterns not playing the role of methods must be obtained by calling factory methods.

Note that in this paper we do not investigate the type system. See future work for ideas on how type-checking can be optimized when integrating BETA objects with a statically typed language.

Super vs. inner

Rather than the traditional `super` method combinator, BETA uses `inner`, which propagates calls from superclass to subclass. For example, a function `Inc` which computes $f(x) = x+1$ and a function `IncAndDouble` which computes $f(x) = (x+1)*2$ could be defined using method combination, as follows:



```
Inc: (# f:< (# i: @integer enter i do i+1 -> i; inner exit i #) #);
IncAndDouble: Inc (# f::< (# do i*2 -> i; inner #) #)
```

Invoking the method `f` on an instance of `IncAndDouble` dispatches to the pattern `f` in the superclass; only when the statement `inner` is executed does control dispatch to the method `f` of `IncAndDouble`. The BETA compiler generates code for `inner` as follows. The pattern `f` is represented using a method `f` that creates an instance of the class `Inc.f` and then calls the method `do` on that instance. This method calls a hook method `inner`, which is empty in `Inc.f`. The pattern `IncAndDouble` compiles into classes `IncAndDouble` and `IncAndDouble.f`; the latter is a subclass of `Inc.f`. This subclass overrides the method `inner` with the implementation of “`i*2 -> i`”.

Following the notion of black-box inheritance introduced in Section 3, we integrate `inner` and `super` as follows. When a BETA pattern is subclassed through FLIF, we allow all methods produced by the BETA pattern to be overridden using the standard semantics of the consumer language. This means that a BETA method which has been overridden is not called unless a super-send is generated from the subclass. Conversely, when a BETA pattern is a subclass of a class produced through FLIF, BETA uses override semantics when methods from the superclass are redefined.

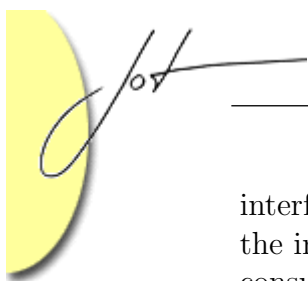
FLIF limitations for BETA

There is no run-time cross-language type checking of formal parameters or return values for calls to methods from other languages; we expect insulation methods can be used exactly as for Java to provide a similar functionality. Moreover, external class declarations cannot be generated automatically from IDL interfaces, but must be written manually. The module system of BETA currently makes it impossible to generate a set of external class declarations in a separate file that can both be used from the program and refer to types in the rest of the program. We plan to overcome this restriction by extending the BETA compiler to use a simple inclusion mechanism for auto-generated files, similar to `#include` from C.

5 IMPLEMENTATION

Runtime infrastructure

FLIF is implemented in Smalltalk and has been integrated as a post-processing phase for the PalVM component loader (also implemented in Smalltalk, as opposed to the class resolver which is implemented by the virtual machine). When a component has been loaded at run time, it is passed to the FLIF IDL-mapper object. Based on the component annotations, all producers are inserted into the FLIF interface repository. The repository contains a symbolic representation of the IDL



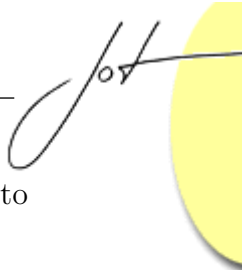
interfaces with references to the concrete components and classes that implement the interface. Once all producers have been placed in the interface repository, each consumer in the component is associated with the corresponding producer, in effect building a graph of dependencies. To complete the resolution process, any producer P (either a concrete IDL class or an IDL class interface) that has been modified is processed. A producer may have been modified either because it is new, because a new subclass has been added, or because new consumers have been associated with it. Processing of P proceeds as follows. The set of implementation languages of the consumers of P is computed, and each consumer of P is extended to support the entire set of languages. Extension of a consumer proceeds as explained in the two previous sections. Annotations on the producer and consumer components identify a language-specific name mapper class which contains static methods that map IDL names to language-specific names. Classes are processed with superclasses before subclasses, to ensure that the language support is in place before processing the subclass.

Of static and dynamic virtual machines

Our approach to language interoperability is implemented on PalVM, a dynamically typed virtual machine. Nevertheless, our approach is not tied to specific features of the PalVM virtual machine, and could be used on a more traditional virtual machine, e.g. on Sun's KVM, albeit with slightly different semantics, depending on the properties of the virtual machine. Virtual machines for statically typed languages typically do not support adding new methods at runtime, which makes it impossible to add new language support to already loaded classes. Nevertheless, the JPDA (Java Platform Debugger Architecture) could probably be used to reload classes with additional interfaces and methods added, allowing dynamic loading to be used with our approach; a detailed treatment of this implementation approach is however out of the scope of this paper. Moreover, we note that due to its dynamic nature, PalVM in principle allows arbitrary cross-language inheritance relationships to be simulated, as long as the subtype comparison operation (e.g., “`instanceof`” in Java) is reified as an operation that can be overridden by FLIF. On a statically typed virtual machine such as the JVM the same degree of flexibility is not possible: at the virtual machine level cross-language inheritance can only be expressed in terms of interfaces.

Name spaces

FLIF is targeted towards an open set of languages, and so it is important that adding new languages does not cause conflicts with already existing languages. A potential problem is if two languages use the same naming convention for methods. If two languages map an IDL method name to the same concrete name, only one of the languages can be supported by that class. To avoid this issue we require that



all languages supporting FLIF use a unique prefix for all method names; we refer to such a prefix as a *name space*.²

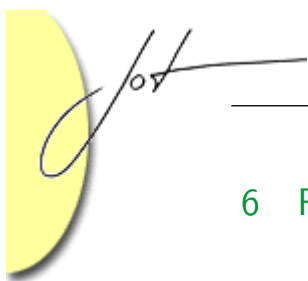
Status of the implementation

The FLIF has been used to implement the run-time environment for the PalCom environment. The Smalltalk part of the class library totals 5kLOC, the Java part of the class library totals 13.7kLOC, but only around 0.5kLOC has been implemented in BETA for PalVM. FLIF is implemented in roughly 1000 lines of Smalltalk code. We note however that apart from the language-specific name mappers, the rest of the framework is language-independent. The size of each of the language-specific name mappers for Smalltalk, BETA, and Java is between 70 and 100 lines of Smalltalk code.

Regarding the virtual machine, the only non-standard primitive operations we require is the ability to create method aliases and the ability to copy methods from one class to another (which may involve updating literal and symbol tables, depending on the implementation of the virtual machine). The virtual machine is currently implemented in two versions, a Java implementation (for desktop experiments) and a C++ version (for deployment in embedded systems). Nevertheless, due implementation differences regarding the component model in the C++ version of PalVM, our full implementation currently only runs on the Java-based version of PalVM. The Java-based version of PalVM uses objects to represent integers and stack frames, making it much less efficient than the C++ version, and moreover causing a significant part of the execution time to be dedicated to integer arithmetic. On this version of PalVM, FLIF currently causes a significant overhead for component loading, but does not impose an overhead at runtime, apart from insulation methods. Preliminary experiments with FLIF for the C++ version of PalVM reveal a minimal initialization overhead and again no runtime overhead apart from insulation methods. Regarding memory usage, method definitions can be shared when making aliases or when copying methods, so the only additional overhead is the additional space consumed in the method tables. The space overhead is currently 36 bytes per alias in the C++ version of PalVM, but we expect that this figure can be significantly reduced by improved implementation techniques.

A Smalltalk IDL compiler, a build utility for generating components by invoking compilers for different languages, and a generic component IDL annotation facility has been implemented in roughly 1300 lines of Common Lisp code. The Smalltalk and BETA compilers have been implemented by other members of the PalCom project, whereas the Java compilation system (described in the appendix) has been developed by one of the authors of the paper.

²We note that such a naming scheme is trivial to optimize with regards to memory consumption in the virtual machine, given that the common prefix can be stored in the component annotations.



6 RELATED WORK

CLR and JVM

The CLR is designed to provide a language-neutral environment, and relies on the CLS specification to define a common subset for language interoperability [11, 3]. The CLR however has no language-independent notion of method names, which means that e.g. Smalltalk selector names cannot be used from other languages without introducing a coupling on the implementation language. Also, insulation methods cannot be used directly with the CLS, since these methods only should be used when invoking methods from a different language. Generating differently named methods for language-internal calls and language-external calls would alleviate this problem, but impede polymorphism and perhaps even require the programmer to explicitly differentiate between language-internal and language-external code. Generating methods with both naming conventions would provide the desired behavior, but is essentially an ad-hoc approach to the more general approach presented in this paper, except that we provide a general framework for runtime language interoperability to reduce the burden of implementing a new, interoperable language. Regarding dynamically typed languages, since the CLR is statically typed, it is ill-suited to representing dynamically typed languages. Moreover, statically compiled languages with more advanced type systems are encumbered by redundant dynamic type checks.

The SML language can be implemented efficiently and mapped to a CLS-compatible compilation scheme, as shown with SML.NET by Benton et al [6]. SML structures can be exported as classes and CLS-compatible classes can be imported as SML structures, given certain restrictions on the external classes. In general, C# classes are mapped to multiple SML entities, depending on how the features of the class should be used. Exporting SML code causes glue code to be generated, and so we can see this as an ad-hoc version of our more general approach, as discussed in the previous paragraph. Naming issues are not explicitly addressed, since only integration with C# is investigated as opposed to arbitrary languages running on the CLR.

Jython and IronPython are implementations of the Python language running on top of JVM and CLR, respectively [8, 1, 17]. Both systems achieve performance comparable to the native Python virtual machine, and both systems allow Python programs to act as consumers for Java/C# classes [12]. The techniques used in implementing these systems are mostly unpublished, but basically consist of using static and dynamic code generation to optimize for common cases with generic (and highly inefficient) fall-backs for uncommon cases [2]. We speculate that both systems benefit from running on highly optimized virtual machines with mature dynamic compilers and efficient garbage collectors, features apparently not found in the C implementation of Python. Moreover, in both cases Python cannot act as a producer; the interpreter can be embedded into Java or C# programs, or in the



case of Java, Python classes can be cross-compiled to Java code (but then they are no longer Python classes).

Smalltalk has been ported both to the JVM and the CLR, but is currently only available for CLR [16]. Smalltalk classes are compiled directly to CLR classes, and every selector declared in a class of the system is installed as a method in the Smalltalk object class, with a default behavior of “does not understand.” Thus, new classes cannot be added dynamically. Smalltalk can act both as producer and consumer, but using a naming convention which gives a high coupling to the implementation language when Smalltalk acts as a producer, and with partly auto-generated selector names when Smalltalk acts as a consumer.

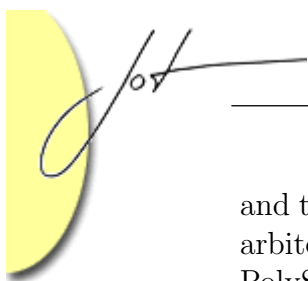
Language integration

We support language integration at component granularity, similarly to standard component models such as CORBA and COM . The concept of an IDL was pioneered by CORBA with a focus on client-server systems, but CORBA objects differ significantly from standard objects, and CORBA moreover introduced a significant overhead in terms of time and space. COM defines a binary representation of components, and allows a tight integration with languages compatible with the COM object model such as C++ and Visual Basic; we believe that our approach to language interoperability would be useful with COM to provide the same advantages as on PalVM.

The integration of languages with very different runtime environments has been investigated by Mecklenburg [15]. This approach is similar to CORBA, using stub methods and translation of objects when crossing language boundaries (although proxy objects can also be used). A language-independent object definition language is used, similarly to our use of IDL, but we can exploit the homogeneity of PalVM, which makes our approach more lightweight. As an example, foreign objects can be referenced directly instead of going through copies or proxies.

We use a simple black-box approach to unifying `super` and `inner` method combination, by restricting `inner` to only be used within languages that use this method combinator (in this case, BETA). Alternatively, `super` and `inner` can be used together to provide more flexible method combination, as shown by Goldberg et al [10]. Here, `inner` dominates `super` so that `super` only dispatches to methods not already included in an `inner` chain. This approach is the dual of our approach, however our approach is much simpler and arguably more pragmatic from a language integration point of view, since `super` is a better fit for most object-oriented languages.

The idea of using a framework to arbitrate method names at run time depending on the specific languages being used has also been investigated by Barrett et al in the context of *polylingual systems* [5]. Here, the PolySPINner tool determines, for a given program written in multiple languages, the set of object types that are being used in the program. Compatible object types are then matched across different languages,



and the implementation of each method is modified to consult the runtime language arbiter at each invocation, to decide whether to perform data conversion. Thus, PolySPINner can be said to rely on a call-by-value semantics (always converting objects) whereas FLIF primarily relies on a call-by-reference semantics with an option of performing value conversion between languages. Moreover, PolySPINner has a significant runtime overhead compared to FLIF, but on the other hand does not rely on a virtual machine.

7 CONCLUSION AND FUTURE WORK

Virtual machines for object-oriented languages provide numerous advantages in terms of adaptability and robustness, but tend to be dedicated to a restricted object model. In this paper, we demonstrate how a virtual machine can host languages with different object models and yet still provide interoperability between these languages. Language interoperability is achieved with only minor restrictions on how each language is compiled to the platform and notably with only a minor implementation effort for each language. We believe such flexible language interoperability to be essential for embedded devices where sharing of implementation is critical to reducing resource consumption.

In terms of future work, we are interested in allowing a tighter integration between specific pairs of programming languages. For example, cross-language type checking can often be eliminated between Java and BETA since both languages are statically typed. Moreover, by using runtime reflection during interface linking to inspect the exact types of BETA classes with covariant attributes, type checking can be reduced even further e.g. for container classes (note that this optimization is in general only possible at runtime during interface linking, since the type of a covariant attribute only is fixed for a given pattern instance).

REFERENCES

- [1] Jython home page. www.jython.org.
- [2] Personal correspondence between a paper author and Dino Viehland on the IronPython mailing list.
- [3] *Common Language Infrastructure, ECMA-335*. ECMA International, 3 edition, 2005.
- [4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley, third edition, 2000.
- [5] Daniel J. Barrett, Alan Kaplan, and Jack C. Wileden. Automated support for seamless interoperability in polylingual software systems. In *SIGSOFT '96*:

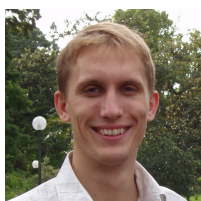


- Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 147–155, New York, NY, USA, 1996. ACM Press.
- [6] Nick Benton, Andrew Kennedy, and Claudio V. Russo. Adventures in interoperability: the SML.NET experience. In *PPDP '04: Proceedings of the 6th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 215–226, New York, NY, USA, 2004. ACM Press.
 - [7] Gilad Bracha and David Griswold. Strongtalk: typechecking Smalltalk in a production environment. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 215–230, New York, NY, USA, 1993. ACM Press.
 - [8] Microsoft Corp. IronPython Workspace Home. www.gotdotnet.com.
 - [9] Adele Goldberg and David Robson. *Smalltalk-80: The language and its Implementation*. Addison Wesley, 1983.
 - [10] David S. Goldberg, Robert Bruce Findler, and Matthew Flatt. Super and inner: together at last! In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 116–129, New York, NY, USA, 2004. ACM Press.
 - [11] Jennifer Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, 2003.
 - [12] J. Hugunin. IronPython: A fast Python implementation for .NET and Mono. In *PyCon 2004*, Washington, D.C., March 2004. <http://www.python.org/pycon/dc2004/papers/>.
 - [13] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
 - [14] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Language*. ACM Press/Addison Wesley, 1993.
 - [15] R. W. Mecklenburg. *Towards a Language Independent Object System*. PhD thesis, University of Utah, June 1991.
 - [16] The Refactory, Inc. #Smalltalk. <http://www.refactory.com>.
 - [17] G. van Rossum and F.L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, September 2003.

ABOUT THE AUTHORS



Torbjörn Ekman is a Research Fellow in the the Programming Tools Group at University of Oxford, UK. He received a PhD from Lund University in 2006. His research interests include extensible compilers, scriptable refactorings, domain-specific languages, and aspect oriented programming. He can be reached at torbjorn@comlab.ox.ac.uk



Peter Mechlenborg received a Master from the University of Aarhus in 2006 and is now employed in the Danish company Mu ApS. He is currently working in the area of highly scalable and fault tolerant systems for the financial sector using Erlang, Common Lisp and domain-specific languages. He can be reached at peter@mu.dk



Ulrik Pagh Schultz is an Associate Professor in the Software Engineering Group at the Maersk McKinney Moeller Institute, University of Southern Denmark. He received a PhD from the University of Rennes I in 2000. His research interests include software engineering for modular robotics, program generation and transformation, and domain-specific languages. He can be reached at ups@mmmi.sdu.dk