Meta-level Language Bridging

Nathanael Schaerli, Franz Achermann, Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
www.iam.unibe.ch/~scg
{schaerli, acherman, oscar.nierstrasz}@iam.unibe.ch

ABSTRACT

Scripting and composition languages offer high-level mechanisms to combine and compose services provided by a lower-level host programming language. Inter-language bridging mechanisms are therefore needed to map host language entities and services to abstractions of the scripting language, and vice versa.

Many popular languages such as Python, Perl, and Ruby use a bridging approach based on *wrappers* that must be written or generated in the host language. Other languages like Jython and Kawa adopt a fixed bridging strategy that exploits *reflective* features provided by the host language. Although both of these approaches are usable, they are cumbersome and low-level. In particular, it can be very difficult to adapt host language services to cooperate seamlessly with abstractions of the scripting language.

In this paper we present a lightweight bridging strategy for scripting and composition languages that simplifies the task of adapting host language services to the abstraction level of the scripting language. This strategy uses introspection facilities of the host language to automate the wrapping process, while providing a hook for programmer-defined adaptation of the generated interface. A meta-level bridging layer is responsible for wrapping and unwrapping both host and scripting language entities so they can seamlessly cooperate.

The bridging strategy employs partial evaluation of wrapping and unwrapping operations to achieve acceptable performance.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *scripting languages, partial evaluation, composition.*

General Terms

Design, Languages, Theory.

Keywords

Scripting Languages, inter-language bridging, higher-level composition, adapting components, partial evaluation

1. INTRODUCTION

Scripting languages offer programmers a high-level view of services implemented in a lower-level host language [1]. *Composition languages* take this idea step further, and offer programmers a way to specify applications as high-level compositions of software components implemented in other, low-level languages [3][4]. Both scripting languages and composition languages must solve a common problem, which is how to bridge the gap between the low-level host language and the high-level scripting view. This language bridge must not only *wrap* components so that they can be accessed from the scripting level, but must also offer the opportunity to *adapt* components that have been designed independently and may therefore not be plug-compatible [2].

We have been developing a general purpose composition language called *Piccola* [5] and have identified a set of requirements that bridging strategies for languages like Piccola should support to achieve seamless, high-level composition. (Since these requirements are also valid for scripting languages, we will use the term "scripting language" in the remainder of this paper to also include composition languages.)

Ready to access. Host language components and services should be *directly accessible* from the scripting language without requiring the programmer to write or generate any wrapper or glue code.

Ready to adapt. Programmers should be able to specify glue code to adapt components directly in the scripting language [2]. It should be possible to define *generic glue code* that will automatically adapt a range of components supporting one interface to another one.

Ready to compose. Host language components should *seamlessly integrate* with scripting language mechanisms [1][6]. This means that scripts can uniformly manipulate both scripting language entities and host language components. Furthermore, expressions that invoke both scripting level and host language services should transparently wrap or unwrap scripting and host level arguments and results as required (possibly making use of generic glue code).

As we shall see in section 3, conventional language bridging strategies adopted by scripting languages such as Perl [7], Python [8], Ruby [9], Jython [10], and Kawa [11] either require wrapper code to be written or generated in the host language, or they adopt a fixed bridging strategy using reflective capabilities of the host language. The wrapping approach violates our first requirement, whereas the reflective approach fails to adequately address the second and third requirements.

In this paper we present the language bridging strategy that we have developed for Piccola. This strategy combines and extends the other two strategies to achieve greater flexibility and thereby address all of our three requirements. The key ideas are:

Conference 00, Month 1-2, 2000, City, State.

- Host language introspection together with a fixed bridging strategy are used to generate *default wrappers* for host components.
- Wrapped components have two identities: (1) the default wrapper used to access its *peer* (i.e., the unwrapped host component), and (2) the current *glue wrapper*.
- Wrapping and unwrapping is performed by a *meta-level language bridge* specified in the scripting language. In this layer, programmers may specify generic glue code and determine when to apply it to host components.

This strategy combines the advantages of both conventional language bridges approaches, while addressing all of our requirements. Furthermore, the approach can be efficiently implemented. By adopting a partial evaluation strategy, it is possible to ensure that only those wrapping and unwrapping operations that are strictly necessary will actually be evaluated at run-time.

The rest of this paper is structured as follows: In section 2, we motivate this work by introducing our view of composition and scripting languages, and by presenting a running example that illustrates our three requirements for language bridging. In section 3 we review and evaluate existing bridging strategies with respect to our requirements and the running example. In section 4 we present the meta-level language bridging strategy. We briefly outline the partial evaluation scheme and outline its correctness proof in section 5. We discuss related and future work in section 6 and conclude this paper in section 7.

2. MOTIVATION

Scripting languages offer a high-level programming interface for specialized tasks, making use of services implemented in a conventional, low-level host language. A *composition language* generalizes the idea of a scripting language, by offering a way to view applications as high-level compositions of software components implemented in conventional languages.

In this section we briefly introduce Piccola, a composition language, and we present a simple example that illustrates the problems of language bridging for scripting languages in general, and for Piccola in particular.

2.1 Piccola — A pure composition language

Piccola is a small language for composing applications from software components [5]. Piccola can be seen as a generic scripting language in the sense that it supports the definition of "compositional styles" for different application domains. Each style determines a set of component interfaces ("plugs") relevant for that domain, and a set of operators ("connectors") that one may use to connect compatible components.

Although the details of Piccola are beyond the scope of this paper, it may help the reader to understand the context of this paper if we explain a few points concerning the intent of Piccola and the design of the language.

Piccola is truly a minimal language in the sense that the language provides almost no built-in behaviour. Everything is accomplished by external components. Even booleans, numbers and strings are provided by the host language (Java or Squeak, in the current implementations).

The requirements we have identified in the introduction are particularly relevant for Piccola. Any host language component should be readily accessible to Piccola scripts. Components will typically need to be adapted to provide high-level plugs corresponding to a compositional style. Finally, since Piccola is a pure composition language, scripts do nothing but coordinate host language components, and so it is essential that connectors specified in Piccola seamlessly interact with the underlying host components, without the need for explicit conversions.

The semantics of Piccola is defined in terms of *agents*, *channels*, *forms*, and *services*. A *script* is the text of a Piccola specification which may be loaded into the Piccola run-time and evaluated by one or more agents. Agents communicate and synchronize using shared channels. Forms are immutable, nested records containing bindings of labels to values. Services are abstractions that may be invoked.

Forms are important because they serve two important purposes: first, they allow us to structure the services provided by external components; second, they double as *explicit namespaces* [12] in which scripts may be evaluated. So forms serve both as primitive objects and as dictionaries.

Syntactically, Piccola resembles the Python scripting language. Semantically it is rather different due to its support for explicit namespaces. The operational semantics of Piccola is defined in terms of an underlying process calculus [13]. An unusual, but important detail is that the sublanguage of forms and services (i.e., without agents and channels) is purely functional. State can only be modeled by communicating information through channels (or by using host components).

2.2 Motivating example

This simple example illustrates the difficulties posed by our three requirements, and applies not only to Piccola, but to scripting languages in general. In this subsection we will assume that our host language is Java and our scripting language is Python (not Piccola).

Suppose that we have an application that uses a Java document component as shown in Figure 1. This component allows a user to access text at the current caret position. Furthermore, it contains services to set the caret position, to check whether the caret is at the end, and to retrieve text up to a certain pattern.

```
class CaretDocument {
  void setPos(int pos);
  void nextPos();
  // Write text and increment position
  void write(String text);
  // Read and increment position
  String read(int len);
  String readUntil(String pattern);
  bool atEnd();
}
```

Figure 1 A host document component

Our application contains services that use this document component. Consider a service `addWordsToDatabase`, which takes such a document as an argument and adds each of its words to a database. Figure 2 shows how such a service might be written in Python.

```
def addWordsToDatabase(doc):
  doc.setPos(1)
  while not(doc.atEnd()):
    word = doc.readUntil(" ")
    database.add(word)
    doc.nextPos()
```

Figure 2 A scripting language service

We use this service in the following scenario that first obtains a new document component from the host language. Then, it adds the words to the database, and finally, it prints the document using an external printer service that is tuned to the document component. (Printer and document come from the same document framework).

```
doc = serviceYieldingDocument()
addWordsToDatabase(doc)
printer = serviceYieldingPrinter()
printer.print(doc)
```

Figure 3 Combining host and scripting level services

Now, we would like to use our application also with other document components that have a different interface. One of these components has the interface shown in figure 4. Basically, this component provides the same functionality as the original one. We can access text at a certain position and also retrieve text up to a certain text pattern. But unlike the original component, the new one does not store the caret position and requires the user to specify this position as an explicit argument when using a service.

```
class TextDocument {
  // Write text at the end
  void write(String text);
  // Write text at a certain position
  void write(int pos, String text);
  String read(int start, int end);
  String readUntil(int start, String pattern);
  int size();
}
```

Figure 4 An incompatible component

A scripting language should allow us to use the new component without changing existing application code. This means that we can adapt the new component so that we neither have to change the higher-level service `addWordsToDatabase` nor the script in which it is used.

In the next section, we review existing bridging techniques and evaluate how well they support this scenario.

3. PROBLEMS WITH TRADITIONAL BRIDGING APPROACHES

In this section, we review the bridging strategies used by popular scripting languages. We discuss how they address the three requirements stated in the introduction of this paper analyze the consequences on our example.

Table 1 gives an overview of the different bridging strategies and shows whether and how they conform to the required properties. In this section we will look at bridging strategies based on reflection and on explicit wrappers and we will review their shortcomings. We will briefly introduce meta-level language bridging, and present it in detail in the following section.

3.1 Fixed strategy based on reflection

Jython and Kawa are implementations of Python and Scheme on top of the Java platform. Although Python and Scheme are rather different languages, the two implementations share a lot of similarities regarding their strategy for inter-language bridging. Both languages provide direct access to all the available Java classes and their constructors, which enables the programmer to directly instantiate them. Both Jython and Kawa use introspection facilities of the Java language to represent instances of host classes as appropriate first class types in their own language model. In case of Jython, this means that

Table 1: Bridging strategies

Strategy	Accessing	Adapting	Composing
Reflection based	Automatic	No support	Scripting level and host services not uniform
Based on wrappers in host language	Needs wrappers in the host language	On the level of the host language	Components not first class entities
Meta-level language bridging	Automatic	On the meta-level	Seamless integration of host components

a Java object is mapped to a Python object with an interface that corresponds to its Java counterpart. As a consequence, the programmer can plug external Java components together by using the higher-level abstractions that are offered by the Python language.

Although this bridging strategy does a good job of allowing host objects to be accessed and composed as they are, this strategy is fixed, because it does not provide a way to adapt components when they are passed to the higher-level language.

In our example this issue urges to programmer to use either manual wrapping or subclassing in order to adapt our new document component. But as we point out in the following subsections, neither approach enables seamless composition of components.

3.1.1 Manually wrapping the component

With this approach, the programmer uses delegation to wrap the new document component into an entity that provides the necessary glue and has the appropriate interface. Unfortunately, this approach suffers from several drawbacks: The main problem is that wrapped components cannot be used as arguments for host services because they do not get converted to the appropriate host objects. This issue is an occurrence of the identity problem that is caused by wrappers because the identity of the wrapper is not the same as the identity of the original entity [2].

We will illustrate this problem with our example scenario. Assume that we have written a service `wrapDocument` that wraps a `TextDocument` so that it provides the same interface as `CaretDocument`. This service allows us to use a `TextDocument` as an argument to the service `addWordsToDatabase`, which is defined in the scripting language. But because of the identity problem, we cannot use this component as an argument to the service `print`, which has been written in the host language.

```
doc = wrapDocument(serviceYieldingDocument())
addWordsToDatabase(doc)
printer = serviceYieldingPrinter()
printer.print(doc) # Fails!
```

Figure 5 Identity problem caused by wrapping

Instead, we have to pass the original document component to the service `print` because only this component can be recognized and used on the level of the host language. Assuming that the wrapped

document contains a service `unwrap`, we must therefore replace the last line and pass the original component as an argument to the print service.

```
doc = wrapDocument(serviceYieldingDocument())
addWordsToDatabase(doc)
printer = serviceYieldingPrinter()
printer.print(doc.unwrap()) # Works!
```

Figure 6 Unwrapping components

It is clear that this manual wrapping and unwrapping of components seriously impairs the higher-level composition process because it does not allow us to deal with both scripting level and host services in a uniform way. An simple but very problematic idea to tackle this problem is to adapt the host services so that they wrap a document when it is passed to the scripting language and unwrap it when it is returned to the host language. In our example, this means that the invocations of `wrapDocument` and `unwrap` are respectively performed in the services `serviceYieldingDocument` and `print`.

In order to realize that, the programmer would have to identify and adapt *all* the host services that have a document component as an argument or as a return value. Besides the fact that this is only possible in a strongly typed language, it also becomes very cumbersome and impractical for non-trivial applications. Already in our small example, we have to adapt the service `print` of all the printer components. This means that we also have to adapt all the host services that are dealing with printer components and then all the components where these services are defined. Obviously, this adoption process has a deeply recursive complexity and results in adoption code that is scatter throughout our application. In his work about integration of independent components, Hölzle refers to this problem as *wrapper explosion* [2].

3.1.2 Subclassing the component

Jython allows one to create subclasses of Java classes. Therefore, it is possible to add glue code to a component by subclassing it. Unfortunately, this approach has only limited applicability for adapting components because subclassing is not designed for adapting interfaces of classes [13]. Since there is no private inheritance in Java (and most other object-oriented languages), a subclass always inherits the whole interface of the base class. This leads to cluttered interfaces, and it makes the new component incompatible to the original one if there is a name clash between the original and the new interface.

In our example, such a name clash occurs because the method `void write(String)` has a different semantics for the two components. (In `CaretDocument`, it writes text at the current caret position, and in `TextDocument` it always writes text at the end). Therefore, it is not possible to subclass `TextDocument` so that it is compatible to `CaretDocument` without breaking compatibility to the base class.

If such a name class does not occur, a subclass is compatible to its base class. This avoids the identity problem that occurs in the wrapping approach because we can use an instance of subclass as an argument for every host service that is written for the base class. Nevertheless, the programmer still has to make sure that all the instances of the original component are replaced with instances of the subclass when they are passed to the scripting language. Especially for non-trivial applications, this is very impractical and leads to the same problems as shown above (wrapper explosion).

3.2 Wrapping in the host language

Many popular languages like Perl, Python or Ruby use a bridging approach based on wrappers and glue code that has to be written in the host language (usually C/C++). This allows a great flexibility in defining the glue abstractions, but it requires the user to specify them on the level of the implementation language, which impairs the higher-level scripting process. Therefore, these languages do not meet the requirement of lightweight and direct access to the host components.

This problem can be tackled by using a tool like SWIG [14] that automatically generates the necessary wrappers from the source code of the components. This grants the programmer direct access to host entities, but it does not preserve the structure of the components. In case of C++ and Python, this means that derived C++ data types such as classes or arrays are not mapped to corresponding Python classes. In fact, instances of derived data types are only available as pointers, and methods are mapped to global functions that need the *this*-pointer as an explicit argument. As a consequence, a component's encapsulated structure and its first class characteristics get lost, which makes it very hard to plug entire components together on a higher level without having to write additional wrapper code first.

Applied to our example, this means that we first have to write or generate wrapping code in order to make the new document component (`TextDocument`) accessible within the scripting language. If we write this code by hand, we can directly include glue code that adapts the component so that it gets compatible to the original document (`CaretDocument`). Unfortunately, this does not avoid the problem that the services and the state of a component are only available as functions respectively pointers on the level of the scripting language. This means that we have to write additional code in the scripting language in order to turn these entities into a single first class object representing the host component.

3.3 Summary and outlook

As a summary, we can say that neither traditional bridging approach meets all the requirements necessary for flexible and higher-level composition. Regarding our example, this has the consequence that it is not possible to reuse a higher-level application for incompatible components without making changes that are scattered across the code of the scripting language or even of the host language.

In the next chapter, we introduce a bridging approach that simplifies the task of adapting host language components to the abstraction level of the scripting language. This strategy uses introspection facilities of the host language to automate the wrapping process, while providing a hook for programmer-defined adoption of the generated interface. A meta-level bridging layer is responsible for wrapping and unwrapping both host and scripting language entities so they can seamlessly cooperate.

4. META-LEVEL LANGUAGE BRIDGING

In this section we present an inter-language bridging strategy that allows the scripting language to meet the requirements stated in the introduction of this paper. We have developed this strategy for our composition language Piccola and we successfully use it in both the Squeak and the Java based implementation [15]. Since the concept of our approach is independent of Piccola, we keep the theoretical part of the description on a general level and mention the Piccola implementation for concrete examples.

Basically, our solution consists of two main concepts: We separate the different aspects of a host component in order to solve the

identity problem. In addition, we move the variable part of the inter-language bridge into the scripting language in order to specify the adaptation and glue code at a higher level.

4.1 Modeling host components

The first step in designing an inter-language bridge is to specify how host components are modeled within the language. According to our requirements, such a model should support the following properties: On one hand, host components that already provide the right structure should be immediately ready to compose, which means that they can be used consistently for services in both the scripting language and the host language. On the other hand, it should also be possible to adapt incompatible components in a higher-level way.

In order to meet these requirements, we separate the two different aspects that are associated with a host component, namely its external identity (i.e. the relation to the associated component) and its higher-level interface together with the glue. This means that we model every host component as a nested structure that embeds the host entity within the higher-level interface. This allows the programmer to adapt the interface and the glue code without affecting the host identity.

In the concrete example of Piccola, forms are the only first class entities, and therefore, we use a nested form to realize this separation. This means that every host component is represented by a form that has the following structure:

Interface and glue. The top level of a host form represents the Piccola interface and the glue that is necessary to adapt the component to a specific compositional style.

Peer. The host form contains a label **`peer`** bound to the form that actually represents the host object. This subform contains a binding for every service provided by the component.

Note that in the rest of this paper, we sometimes use the names *peer* and *interface* to refer respectively to the external identity and the higher-level interface (including the glue code) of a host component.

Figure 7 illustrates the nested structure of a form that represents a document component `TextDocument` with an interface that is compatible to `CaretDocument`. At the top level of the form, there is the Piccola interface, which consists of the services that implement the glue to access to component in the appropriate way. In addition, there is the label peer bound to the peer form representing the identity of the host component. This form contains bindings that are directly mapped to the host component and provide direct access to the lower-level services.

Although this separation is also a variant of wrapping, the bridging process explained in the following section allows us to overcome the identity problem that occurs in the other solutions (cf. Section 3). The reason for that is the fact that all the wrapping is transparently performed on the level of the inter-language bridge rather than in the code written by the user. Every component that is passed to the scripting language is automatically converted into this nested structure. Therefore, the services inside the scripting languages always see the higher-level interface, which can be freely adapted by the programmer. In the other direction, when such a component is used as a parameter for a lower-level service, the bridge only passes the external identity (peer) to the host language. This ensures that the lower-level services always deal with the original object, and it therefore avoids the identity problem.

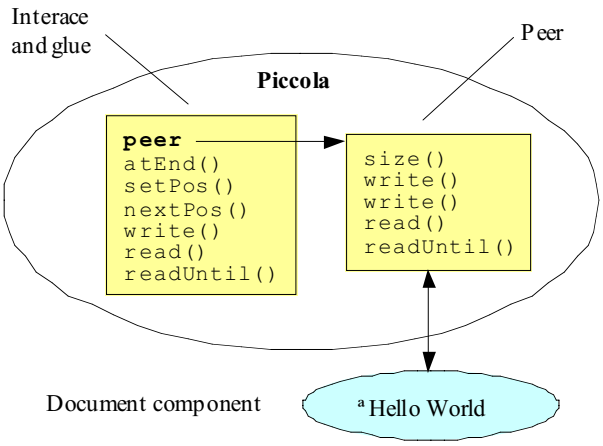

Figure 7 Structure of a host component

4.2 Defining the bridging process

In the previous section, we have illustrated that modeling host components with a nested structure allows the programmer to modify these components without affecting the compatibility to host services. Now, we present an inter-language bridge that supports this concept in a way that allows the programmer to specify glue code that is automatically applied to an entire class of components.

Figure 8 gives an overview of this bridging process. It shows that the inter-language bridge is divided into two parts: The generic part is implemented in the virtual machine and the variable part is located inside the scripting language. When a host component is passed to the scripting language (left side), the generic part converts it into a nested structure as described in the previous section. Then, this structure is passed to the variable part of the bridge. Here, the generic interface gets replaced with a specific interface that can be specified by the programmer. The resulting structure consisting of the specific interface and the peer represents the component within the scripting language. Note that the variable part of the interface may not necessarily provide every object with a specific interface. In this case the generic interface is used within the scripting language.

When a host component is passed back to the host language (right side), the inter-language bridge discards the higher-level interface and only passes the peer. This avoids the identity problem and guarantees that the component has the structure required by the host language.

4.2.1 Passing host components to the scripting language

When a host component is passed to the scripting language, the task of the inter-language bridge can be divided into two parts. First, the bridge has to convert the component to make it technically compatible to the object model of the scripting language. And second, it should adapt the component in order to fit the needs of the application and to cooperate with the other components. Whereas the first part is generic, the second one is completely variable and may depend on many aspects of the component such as its class or the current state (e.g. instance variable values).

In most traditional approaches, both of these parts are hard-coded in the virtual machine, and the user cannot influence how components in one language are mapped to components of the other language. As we have pointed out in Section 3, this is a problem because

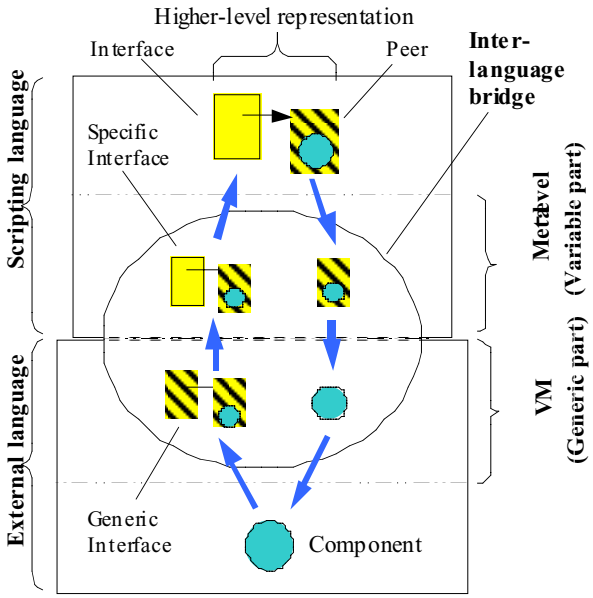

Figure 8 The inter-language bridge

it does not allow the programmer to work with incompatible components and to adapt them according to the specific needs of the application. We avoid this limitation by minimizing the bridging code in the virtual machine to the technical conversion and moving the variable part into the scripting language. This allows the programmer to define the representation of host components by modifying the bridging framework within the higher-level language itself. Controlling the behavior of a language within the language itself is called meta-programming [16], and we therefore say that the variable aspect of the inter-language bridge is located in the *meta-level* of the scripting language.

Performing the technical conversion in the virtual machine

The responsibility of the bridging layer in the virtual machine is to convert a component so that it is compatible to the object model of the scripting language. This is accomplished by building up a nested structure as defined in Section 4.1. This structure provides access to all the available services of the original component. Note that this step is entirely generic, and therefore, the higher-level interface is identical to the interface of the original component and all the services are directly mapped to the corresponding peer services.

For both the Squeak and the Java based Piccola implementation we were able to do this automatically by using run-time introspection functionality offered by the host language. This allows the bridge to create a generic Piccola form for every host component that is passed to the Piccola language. In figure 9, we show the generic component that gets created for the document component `TextDocument`. Note that it has the nested structure defined above and that the services in the interface are direct references to the services in the peer. This structure allows the programmer to access the component as it is provided by the host language. Because the interface is separated from the peer, it can be modified without affecting the external identity.

Since many popular languages and component architectures offer basic introspection facilities, a similar approach can be used for other scripting languages. If the host language does not support run-

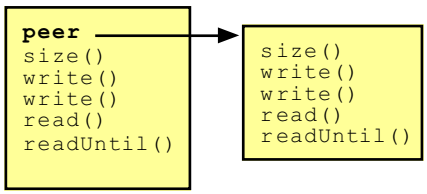

Figure 9 Generic representation of `TextDocument`

time introspection, this gets more difficult. However, if the source code is available, a generic conversion can be realized with source code analysis. This is essentially the same approach that is used by SWIG in order to create wrappers for C++ components [14].

Performing the variable part in the scripting language

After the virtual machine layer of the inter-language bridge has created the generic representation of the host component, this representation gets passed to the variable part of the inter-language bridge, which is located inside the scripting language. The purpose of this part is to automatically adapt certain classes of components in order to make them compatible to one another and to the requirements of the application. Whereas in most other languages, the glue code for specific components is either hard-coded in the virtual machine or has to be written on the level of the host language, our approach allows the programmer to do this on the meta-level of the scripting language. This means that he can use the unrestricted expressive power of the higher-level language to configure host components. For instance, this makes it possible to use the state of the application to dynamically determine how a certain component should be configured.

An interesting aspect of our approach is the question how to organize the meta-level bridging layer and how to activate it from the bridging layer in the virtual machine. In our work on Piccola, we have been experimenting with different approaches [15]. In one of them, we introduce a single hook service that gets called from the generic part of the inter-language bridge whenever a host component is passed to Piccola. As an argument, this service receives the generic structure that is created by the bridging layer in the virtual machine. Depending on characteristics of the component (e.g. its class), this hook service now calls an appropriate service to adapt the interface of the component or leaves it untouched if the programmer has not defined a specific service to configure this component. The programmer has complete freedom in specifying both the hook service and the services that finally adapt the components.

The code in figure 10 shows how this mechanism can be used to automatically provide the `TextDocument` components with an interface and glue code that makes them equivalent to `CaretDocument`. When a `TextDocument` is passed to the scripting language, it gets first converted into the generic form that is shown in figure 9. Then, this form is passed as an argument to the hook service `hook`. Because the host component is an instance of the class `TextDocument`, the service `adaptTextDocument` provides it with the necessary glue code and returns the nested form shown in figure 7. Otherwise, the argument form is just returned as it is.

In the service `adaptTextDocument`, we provide the glue code to make the new component compatible. First, we create a private variable that stores the current caret position and is initialized with 1. Then, we create the label `peer` and bind it to the peer of the argument component. Finally, we add all the services that are needed to make the higher-level interface compatible to the component `CaretDocument`. Note that in Piccola, we use a quote (`'`) to define a tem-

porary (private) binding that is not visible in the return value. Furthermore, we use a form with bindings `val1` and `val2` to invoke host services with more than one argument. For conciseness, we do not add code to make this component thread safe. However, this could be achieved easily by using the synchronization mechanisms provided by the Piccola language.

```
# Adapt interface of a TextDocument component
adaptTextDocument(Doc):
  # Private variable to store the position
    'pos = newVar 1
    peer = Doc.peer
    setCursorPos newPos:
      'pos <- newPos
    nextPos:
      setCursorPos(*pos + 1)
    write text:
    'peer.write(val1 = text, val2 = *pos)
      'setCursorPos(*pos + text.size())
    read len:
    peer.read pos(*pos + len)
      'setCursorPos(*pos + len)
    readUntil pattern:
    'text = peer.readUntil
              (val1 = pos, val2 = pattern)
    'setCursorPos(*pos + text.size())
      text
    atEnd:
    *pos == peer.size()

# Hook service that is called from the VM
hook(Component):
  if className(Component) = "TextDocument"
    then: adapt TextDocument(Component)
    else: Component
```

Figure 10 The meta-level bridging layer

4.2.2 Passing objects to the host language

When an object of the scripting language is passed as an argument to a host service, the inter-language bridge is responsible for converting it into the structure expected by the host language. In our bridging strategy, the actions taken by the bridge depend on whether the passed object represents a host component or is an object that was ex nihilo created within the scripting language.

If the object of the scripting language does not represent a host component, the bridge just passes the lower-level representation of the object to the host language. Since we assume that the scripting language itself is implemented on top of the host language, every object of the scripting language is finally also an object of the host language, and therefore, this bridging activity is trivial.

The more interesting case applies if the passed object is a host component. Then, the inter-language bridge only passes the associated peer rather than the whole structure that represents the component on the higher-level (cf. figure 8). As we have already pointed out in Section 4.1, this avoids the identity problem because the unmodified peer is guaranteed to have the structure expected by the host service.

4.2.3 Example

Using this bridging strategy, we are able to use our example application also for `TextDocument` components without changing either the scripting abstractions (e.g., the service `addWordsToDatabase`) nor the code where they are used. The only thing we have to do is to specify the adaptation code for the new document component in the meta-level bridging layer as shown in figure 10.

When we execute our application, every `TextDocument` component is automatically adapted by the inter-language bridge. In the case of our example scenario (cf. figure 3), this means that the component returned by the service `serviceYieldingDocument` already has the structure shown in figure 6. Therefore, we do not have to manually wrap this component and can directly use it as the argument of the service `addWordsToDatabase`. Because the inter-language bridge only passes the peer to a host service, we can use the same component also as an argument to host services. In our example, this means that the service `print` of the host printer component can be invoked directly with the higher-level representation of the component.

5. IMPLEMENTATION ISSUES

Adapting host components in the meta-level of the scripting language adds an additional layer of indirection. An implementation must avoid the associated performance cost as much as possible. In the Piccola implementation, we use caching and partial evaluation in order to achieve this.

Without optimization, a lot of time is spent in the hook service that dispatches the wrappers for a given host component. Since the dispatching process typically only depends on a few parameters such as the class of the component, we can build up an efficient cache table. This table associates a wrapper with the parameters that were used in the dispatching process and gets cleared when the hook service changes.

In addition we use partial evaluation as a general technique for optimization and in particular apply it to specialize the indirection caused by the wrappers in the bridging layer. Partial evaluation is a source-to-source program transformation technique for specializing programs with respect to parts of their input [21]. With respect to the wrappers, partial evaluation weaves the effectively needed glue code into the application script. This avoids wrapping and unwrapping of services that are not used.

Consider the expression `a = [1,2]` which is syntactic sugar for `a = newList().add(1).add(2)`. When naively executing this expression we first create an empty list in the host language, wrap it in the scripting language, create (or grab) a host number 1, wrap the number and pass this wrapped number as an argument to the service `add` of the wrapped list, where it gets unwrapped and added to the host list. The same applies for the second element. Each time an object crosses the language boundary, a wrapper is added or removed.

In the Java version of Piccola, the service `newList` creates a host object of the Java class `Vector` and wraps it at the meta-level so that it conforms to the scripting view of a list. This view requires, amongst others, services `add` and `forEach`. The service `add` is mapped to the Java method `addElement`, but specifies a different return value. The service `forEach` is specified at the meta-level by using an iterator. In the above expression, we only use the service `add`. Partial evaluation specializes this expressions with the wrapper and generates the following equivalent, but more efficient code:

```
a = Host.class("java.util.Vector").new()
a.addElement(1)
a.addElement(2)
```

Because partial evaluation performs non-standard eager evaluation, we need knowledge about the side-effects in order to apply it

correctly. Therefore, we use a transformation to separate the side-effect part from the referentially transparent part. When specializing we simplify the referentially transparent part, and defer the invocation of the side-effects until run-time. In order to show correctness of the partial evaluation, we use the formal model of Piccola.

This formal model consists of agent expressions that evaluate to forms. The partial evaluation transforms any agent expression to a sandbox expressions, which are also agent expressions. A *sandbox* expression *A; B* specifies the namespace *A* in which the expression *B* is evaluated. The free identifiers of *B* are looked up in the value of *A*.

For instance, the agent expression $\lambda x.\varepsilon; A$ denotes an abstraction over *x*. The body of the abstraction consists of the side-effect ε (no side effect) and a referentially transparent part *A*. Application of this service with the argument *(B; C)*, where *B* denotes the side effect part and *C* the referentially transparent, can be transformed to *B; A[x/C]*. This means that we substitute the referentially transparent part *C* for *x* in *A*, and we keep the side effect part *B*. The free variables of *A[x/C]* are identical to the free variables of *C* and they refer to the same side-effects in *B*.

For the correctness proof we show by structural induction over agent expressions *E* which are transformed to *A; B* that the following holds:

- The value of *A* contains the results of all the side-effect computations in *E*.
- The expression *B* is referentially transparent, and all free identifiers in *B* denote the results of side-effect terms available in *A*.
- The value of *E* and *A; B* are the same

The formal calculus for Piccola and the correctness proof for the specialization algorithm is in Achermann's thesis [13].

6. RELATED AND FUTURE WORK

Throughout this paper, we have already discussed the bridging strategies of many popular scripting languages. Other bridging approaches use a component model to uniformly access external components.

Jones et al. [18] use Haskell to script COM components and make use of higher-order functions. They also use the type system to detect certain composition errors at compile time. IBM's System Object Model (SOM) [19] is another approach that allows a programmer to use components that are written in a separate language. SOM is designed specifically to overcome the main obstacles to the pervasive use of object class libraries. System objects can be distributed in binary form. In addition, they can be used and subclassed across different languages. SOM is based on an advanced object model and an object-oriented run-time engine that supports this model. SOM supports the concepts and mechanisms that are normally associated with object-oriented systems including inheritance, encapsulation, and polymorphism.

A component model pushes the idea of standardized interfaces. Therefore, the adaptation problems addressed in this paper are less frequent, but not solved in general.

In order to adapt host components, our bridging approach uses automatic and transparent wrapping. Büchi and Weck [22] use generic wrappers in order to solve the identity problem in the context of strongly typed languages but not in the context of inter-language bridging.

In parallel to the work described in this paper, we have also been working on distributed Piccola, and our goal is to implement a distribution layer directly in Piccola. Especially for distribution between heterogeneous Piccola hosts, we need a flexible technique to abstract away from the host language. Future work will show how the bridging approach presented in this paper performs in such a distributed environment.

7. CONCLUSION

In this paper, we show that the requirements for a higher-level scripting and composition language are strongly related to the strategy for inter-language bridging. Most of the traditional bridging approaches focus on the technical aspect of making host components available in the scripting language, but they do not provide the necessary abstraction mechanisms to access the components in a higher-level way.

Analysis of these problems leads to the conclusion that it is not possible to achieve the needed flexibility with a generic bridging strategy that is hard-coded in the virtual machine. Thus, we have proposed an inter-language bridge that reduces the activities in the virtual machine to a technical conversion and performs the higher-level configuration in an abstraction layer located in the meta-level of the scripting language. This allows the programmer to use the full expressive power of the scripting language to adapt and configure a host component in order to fit the needs of the application and cooperate with the other components.

Using this bridging strategy, we were able to develop and implement the Piccola 3 standard, which is completely independent of the underlying host language. Since Piccola is a pure composition language, the standard specifies standard components such as numbers, strings, collections, and URLs that are used very frequently. Because all the necessary glue code is written in Piccola itself, the programmer may dynamically reconfigure these components depending on the specific requirements of an application.

8. REFERENCES

[1] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. IEEE Computer magazine, March 1998.

[2] Urs Hölzle. Integrating Independently-Developed Components in Object-Oriented Languages. ECOOP '93 Proceedings, Springer Verlag Lecture Notes on Computer Science.

[3] Jean-Guy Schneider and Oscar Nierstrasz. Components, Scripts and Glue. Software Architectures - Advances and Applications, Leonor Barroca, Jon Hall and Patrick Hall (Eds.), pp. 13-25, Springer, 1999.

[4] Jean-Guy Schneider. Components, Scripts, and Glue: A conceptual framework for software composition. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[5] Franz Achermann and Oscar Nierstrasz. Applications = Components + Scripts — A Tour of Piccola. Software Architectures and Component Technology, Mehmet Aksit (Ed.), Kluwer, 2001, to appear.

[6] Franz Achermann, Stefan Kneubuehl and Oscar Nierstrasz. Scripting Coordination Styles. Coordination Languages and Models, António Porto and Gruia-Catalin Roman (Eds.), LNCS 1906, Limassol, Cyprus, September 2000, pp. 19-35.

[7] Larry Wall, Tom Christiansen, Jon Orwant. Programming Perl (3rd Edition). O'Reilly & Associates, ISBN: 0596000278, July 2000.

[8] Mark Lutz. Programming Python (2nd Edition). O'Reilly & Associates, ISBN: 0596000855, March 2001.

[9] Yukio Matsumoto, Yukihiro Matsumoto. The Ruby Programming Language. Addison Wesley Professional, February 2002, to appear.

[10] Jython, an implementation of the high-level, dynamic, object-oriented language Python written in 100% Pure Java. http://www.jython.org/.

[11] Per Bothner. Kawa, the Java-based Scheme system. http://www.gnu.org/software/kawa/.

[12] Franz Achermann and Oscar Nierstrasz. Explicit Namespaces. Modular Programming Languages, Jürg Gutknecht and Wolfgang Weck (Eds.), LNCS 1897, Zürich, Switzerland, September 2000, pp. 77-89.

[13] Franz Achermann. Forms, Agents, and Channels, defining Composition Abstractions with Style. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, 2002, to appear.

[14] David M. Beazley. Interfacing C/C++ and Python with SWIG. 7th International Python Conference, SWIG Tutorial, 1998.

[15] Nathanael Schärli. Supporting Pure Composition by Inter-language Bridging on the Meta-level. Masters thesis, University of Bern, September 2001.

[16] Gregor Kiczales, Jim des Rivière and Daniel G. Bobrov. The Art of the Metaobject Protocol. MIT Press, 1991.

[17] Clemens Szyperski. Component Software - Beyond Object-Oriented Programming. Addison-Wesley, 1998

[18] Simon L. Peyton-Jones, Erik Meijer and Daan Leijen. Scripting COM components in Haskell. Fifth International Conference on Software Reuse (ICSR5), Victoria, Canada, 1998.

[19] Christina Lau. Object-Oriented Programming Using SOM and DSOM. John Wiley & Sons, March 1995.

[20] Alan Snyder. Inheritance and the Development of Encapsulated Software Components. In Research Directions in Object-Oriented Programming B. Shriver and P. Wegner (eds), pp 165-188, MIT Press 1987.

[21] Charles Consel, Oliver Danvy. Tutorial Notes on Partial Evaluation. In 20th ACM Symposium on Principles of Programming Languages. Charleston, South Carolina, pp.493-501, ACM Press 1993.

[22] Martin Büchi and Wolfgang Weck, Generic Wrappers, ECOOP 2000, 14th European Conference on Object-Oriented Programming, Elisa Bertino (Ed.), LNCS, vol. 1850, Springer Verlag, 2000, pp. 201—225.