Generating Bridges Between Heterogeneous Component Models

Kostadin Damevski

SCI Institute, University of Utah, Salt Lake City UT 84109, USA,
damevski@cs.utah.edu

Abstract. Software reuse lies at the heart of component software. It is considered to be the main reason for the wide adoption of this technology. One way of enabling wider reuse is to enable the use of a larger selection of component instances, providing a greater assortment of tools to the component writer when solving a problem. To this end, the approach described here focuses on automatically fostering interoperability between components belonging to a range of component models. By leveraging a generative programming approach, a programmable code generator is used to interpose a translation between components of disparate component models. We describe the design of a code generator for this domain, and extensions that aim to create a robust and flexible method of bridging incompatible component instances.

1 Introduction

Numerous component models already exist. They range from business-oriented to those designed for the purpose of biology or physics simulations. Enterprise Java Beans (EJB) [1], for instance, is a component technology developed by Sun Microsystems. EJB are components based on the Java programming language that are especially suitable for applications that involve database access. Microsoft's Component Object Model (COM) [2] is an architecture that allows applications to be developed through components based on a binary standard for component encapsulation. The Object Management Group (OMG) developed the Common Object Request Broker Architecture (CORBA) [3], which is an open, vendor-independent architecture and infrastructure in which computer applications can work together in a distributed environment. Common Component Architecture (CCA) [4] is a component model designed to fit the needs of the scientific computing community. To this goal, the CCA model provides support for complex number types and parallel programming as well as streamlined execution in order to minimize overhead and maximize performance.

Enterprise Java Beans, COM, CORBA, CCA, and other component models typically do not interoperate with one another. Components in one system are not usable in another. This lack of interoperability hinders the basic design goal of component technology: component reuse. End-users that need the tools to adapt to the requirements of their work are at a large disadvantage. This is because they are required to "buy-in" to one specific model and produce components for one specific system.

It is possible to surpass this limitation by enabling the generation of ad-hoc bridge instances between two heterogeneous components, allowing the user to leverage the

best available tool for the job. This enables the creation of a simulation that may use EJB components to get a dataset from a remote database, use CCA components to perform a computation on the data, and leverage Visualization Toolkit (VTK) components to visualize the output. The approach shown here is far superior to the current "state of the art", which often involves translating each component instance from one model to another by hand or via a hand-written bridge.

Our approach to bridging is to automatically interpose a bridge component between two incompatible component instances. We use a compiler tool called SCIM (SCIRun Component Interface Mapper) to automate the generation of the component bridges. SCIM is a code generator that requires an interface description of the components and an output template specifying the component bridge in a generalized way. Bridging every two specific component models requires a separate template, though many templates involving a single component model may be very similar. A template should be written by a software engineer well familiar with the designated component models. SCIM is made more powerful by the inclusion of script language code (snippets of Ruby) within the template. By virtue of these features and others, SCIM allows flexibility in the code generation as well as the ability to handle many component models.

Thus far, the experiments in bridging component models have been performed within our multi-component model compliant framework, SCIRun2 [5]. The framework uses a meta-component model that is designed to permit the inclusion of many different component models. The meta-component model operates by providing a plug-in architecture for component models. Abstract components are manipulated and managed by the SCIRun2 framework, while concrete component models perform the actual work. The abstract model generalizes minimal features that components and their component models contain. The goal of SCIM is broader than just providing a specialized framework like SCIRun2 with a method of combining execution paths of its component models. Its aim is to provide this type of functionality to an arbitrary component framework.

In the next chapter of this paper, we focus on the SCIM compiler and it current capabilities. In chapter 3 we will discuss planned additions that have the goal of making SCIM more useful and capable of bridging many different component models.

2 Design

A suitable method of bridging components belonging to different models is by using a code generation tool that automatically generates translation code between two component instances. SCIM is a tool that is intended to provide a viable solution to bridging component model differences. SCIM contains an interface mapping front-end and a template-based code generation back-end. The front-end parses component interfaces and ensures semantically similar interfaces with only syntactic differences are matched, while the back-end is intended to handle the actual code generation. Input containing one or two interfaces and a template describing the generated code are necessary for SCIM to generate a component bridge. The tool handles several IDLs and programming languages as interface input. The output template that represents a specification for code generation is expressed in eRuby (embedded Ruby). eRuby is a templating

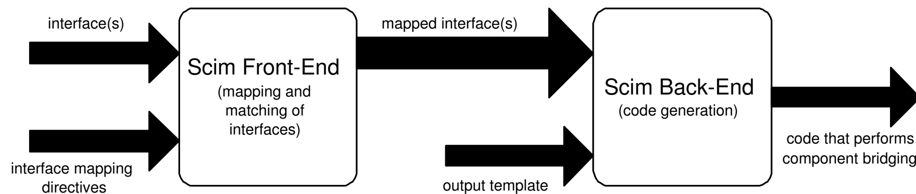

Fig. 1. The structure of the SCIM code generation tool.

tool based on the Ruby scripting language. It has the ability to interpret embedded Ruby language commands within a template. The use of the scripting language within the template allows the expression of many output specifications that in turn will be able to express the bridging of a large number of component models. Figure 1 provides a graphical representation of the entire structure of this compiler tool.

The SCIM front end receives and processes interfaces written in a few IDLs or programming languages. When required, the interfaces can map their differences to express a match between two semantically similar but syntactically different interfaces. Interfaces that do not require this mapping are tagged as *inout*, differing with *in* and *out* interfaces that require an explicit mapping in order to overcome differences between the interfaces. The match between an *in* and an *out* interface is done through interface mapping directives that can be included in the interface file. The result of this mapping is to select matching interfaces and forward them to the compiler back-end to perform code generation.

The interface mapping directives are a series of commands that can be used to create a match between two interfaces. They are very useful in many situations when two components have interfaces that contain discrepancies in names or types of methods and arguments. To better illustrate each of the interface mapping directives, consider an example of two functionally equivalent but syntactically different interfaces:

```
interface scarlet {
  void paint();
}

interface crimson {
  void color();
}
```

To explicitly map between two interfaces, a special *%map* command is provided. The command is only needed in situations where there are syntactic differences between interfaces. It states which differences are in fact just misnamed similarities and should be ignored. After issuing the *map* command, if the two interfaces match completely, they are adequate to proceed to the code generation back-end of SCIM. The *map* command is able to express differences in several parts of an interface. These are: interface name, method names and types, and argument types. The map command for our crimson and scarlet interfaces would look like this:

```
%map crimson -> scarlet
>color -> paint
```

The purpose of SCIM's back-end is to combine the interface mapping with a output code specification template written in embedded Ruby (eRuby). The interface to interface maps that we have devised in the front end and data extracted from the interface are expressed through predefined data structures within the template.

Below is an example template that would perform simple method redirection for C++ interfaces.

```
<interface>
 <method>
<%= $inMethodType%>
  <%= $inInterfaceName%>
    ::<%= $inMethodName%>() {
  obj = new <%= $inInterfaceName%>();
  obj-><%= $outMethodName%>();
}
 </method>
</interface>
```

For our running example of the scarlet and crimson interfaces, SCIM would produce this C++ code using the template above:

```
void
  crimson
    ::color() {
  obj = new scarlet();
  obj->paint();
}
```

3 Features

This section describes the planned and in-progress additions to SCIM: an infrastructure to support the handling of many different types of data and an ability to describe component models in order to automatically generate SCIM templates between them.

3.1 Data Handling

One of the most significant problems in generating bridges for families of component instances is handling the variability of data that comes with each component model. Base types, arrays and object types need to be translated between component models keeping in mind various esoteric rules imposed by specific component models. This is a tall order to handle automatically and generally. SCIM's current capabilities allow a user to manage (through their own Ruby code) different code generation options for

dealing with data. However, we intend to provide an improvement through an extensible framework containing a bag of tools that can help the user in handling the transfer of the data. This framework will guide the user in specifying the data requirements on a component model basis. In this fashion, the various data transfer possibilities will be enumerated making it easy for the user to select whether, for instance, the component model allows pointer types as parameters. In addition to this, we will provide an automatic way to handle base and array types through copy (pass-by-value) of the data, and complicated object types through a redirection (pass-by-reference) mechanism. This will provide default behavior that will be appropriate for many scenarios, but that can also be overridden in special cases. When the default scenario is inappropriate, the user will be able to specify triggers for a specific data translation based on the data type being passed.

3.2 Meta-Model Template Generation

SCIM requires a template closely describing the details of the bridging between two component models. General and useful templates can be hard to write as they require detailed knowledge of each component model. In addition, a template is required per every two component models, making the task of fully integrating many component models to be time-consuming and repetitive. To remedy this, we plan to provide a way to automatically generate SCIM templates. Templates can be generated based on a description of each component model expressed in a general meta-model. The meta-model represents a given component model at a higher abstraction level. Its structure aims to express both structural and behavioral features. The intention of the generated template is to be near exactly what SCIM needs, requiring the user to make the least amount of additions/modifications to it as possible. This provides for the ability to help a user in the difficult task of template creation, once the features of a specific component model are expressed using the meta-model. The instantiation of the meta-model to a specific component model will subsequently apply to all template generation involving said model.

References

1. Enterprise Java Beans: http://java.sun.com/products/ejb (2004)
2. Component Object Model: http://www.microsoft.com/com (2004)
3. OMG: The Common Object Request Broker: Architecture and Specification. Revision 2.0. (1995)
4. Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B.: Toward a common component architecture for high-performance scientific computing. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computation (HPDC). (1999)
5. Zhang, K., Damevski, K., Venkatachalapathy, V., Parker, S.: SCIRun2: A CCA framework for high performance computing. In: Proceedings of The 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments. (2004)