COMPONENT MODEL INTEROPERABILITY FOR SCIENTIFIC COMPUTING

by

Kostadin Damevski

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2006

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Kostadin Damevski

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Steven Parker
	Matthew Flatt
	Gary Lindstrom
	John Regehr
	James Kohl

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Kostadin Damevski in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date	Steven Parker Chair, Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

The ability to create connections between previously incompatible software is the motivation for the work of this dissertation. We focus on devising a paradigm for interoperability between software components, a popular and often used software methodology, while making the practical concerns of this endeavor our main driving force.

Component software interoperability is the ability for two or more existing software components to cooperate despite differences in component model, interface or execution platform. It is also the ability to connect and use existing servers though they may be plug-incompatible to the clients. The capability of combining any two components regardless of implementation platform creates a new realm of possibility in using common off-the-shelf components. Unfortunately, solving the general problem of component interoperability using a fully automatic mechanism is daunting. In this work, we describe and analyze a set of techniques and tools that we developed to make component interoperability attainable in a practical sense. Our approach simplifies a difficult but necessary goal of component interoperability and our user-driven generative tools provide several abstractions enabling flexible and structured component integration.

By leveraging a generative programming approach, we develop a programmable code generator that interposes a bridge to translate between components of disparate component models. We extend this bridge generation with tools aimed at different aspects of the problem to create a reliable and flexible method of performing this task. Focus is given to mapping of incompatible interfaces and data types and to manipulating data between two heterogeneous components. In order to include semantic understanding of component models into our design, we define a component meta-modeling method and a bridging language. The bridging language leverages component meta-models, which semantically model the individual features of each component model, to produce a translation between two such models. We combine these methods into our stack of tools that aim to bridge difference between heterogeneous component instances and provide component interoperability.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. Steven Parker for supporting my work and providing weekly guidance through the last five years. Without his continued effort none of this would have been possible. I would also like to thank the members of my committee for their support. Thanks to the CCA (SCIRun2) group at the Scientific Computing and Imaging Institute at the University of Utah for listening to my ideas and an atmosphere of collaboration. I would like to acknowledge David Walter, David Goldberg and Joel Daniels each of who carefully reviewed a text or commented on a presentation several times during my graduate career. Finally, I'd like to thank my family and friends for their unflinching support.

CHAPTER 1

INTRODUCTION

This dissertation introduces an approach for creating software that mediates the communication between two previously incompatible component instances, usually belonging to two separate component models. Our approach defines differences among component models and exploits similarities between groups of component instances that belong to the same component model. This approach focuses on several different ways that components may differ, providing a practical difference mitigation strategy. We implemented our interoperability method using two software tools: a code generator and a domain-specific language.

Our work contributes a code generator aimed at software interoperability that is expressive and flexible, enabling it to be used in many situations. The generator enables the handling of data and interface differences that often exist in a set of incompatible component instances. To specify higher-level differences between component models we devised a domain-specific language for component-model interoperability. Our domain-specific language is designed to express component model differences quickly and accurately and generates code that can be interposed between two component instances to enable intercommunication.

1.1 Background

A software component is "a unit of reuse that specifies a particular abstraction and can be independently deployed. A third party need not be aware of any of the implementation aspects of the component and treats it as an off-the-shelf piece of software" [41]. The component follows a set of rules for behavior and configuration, and provides a well-defined external interface. These rules and interfaces provide it with the capability to be composed with other components, often dynamically. The rules that define a component, also called a component model, trade off various capabilities (such as performance, ease of use, component isolation, etc.) These trade-offs and design considerations are based on the needs of specific audience of users, concentrating the component model to a target application domain.

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) [18], 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) [13] 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) [33], which is an open, vendor-independent architecture and infrastructure in which computer applications can work together in a distributed environment. Common Component Architecture (CCA) [3] 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.

Unfortunately, the 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.

The goal of this dissertation is to provide a method to alleviate this problem. The approach used is based on interposing automatically generated code between two components of disparate component models. This generated code is able to translate that interaction between the components at a reasonable cost. We will also describe approaches in flexibly generating the necessary code between instances belonging to two different component models. We will attempt to show that this is practical and feasible even when component models differ radically in their approach.

1.2 Introduction to Component Software

A component is a unit of software development that promotes reuse. Logically, a component consists of a group or a framework of objects that are ready for deployment. The difference between component software and object-oriented software is somewhat subtle; it is in the deployment of the software. Components are individually and independently deployed, often with rules that manage this deployment. These rules in several component models consist of managing the instantiation of objects within the component, dependent upon workload or resource limitations.

Component programming makes a clear separation between a provider (server) and a user (client). In fact, this separation is such that it is clearly and strictly defined by an interface agreement. This interface agreement, in turn, is defined by a special language known as Interface Definition Language (IDL). An interface describes the services a provider component provides to the world, and also serves as a contract when a user component asks for these services. A typical IDL interface description consists of class and method declarations. In addition, each argument is predicated by the keywords *in*, *out*, or *inout*. These keywords are necessary in order to provide support for distributed components and they are present in most IDLs. *In* results with the argument being sent into the method implementation, *out* causes the method implementation to return the argument, and *inout* performs both of these actions. Here is an example of the OMG IDL which is used by the CORBA component standard [33]:

```
module Example {
  struct Date {
    unsigned short Day;
    unsigned short Month;
    unsigned short Year;
  }
  interface UFO {
    void reportSighting(in Date date_seen,
                        in int num_contacts,
                        out long ID);
  }
}
```

The *reportSighting* method description above requires three parameters. The date of the UFO sighting and the number of contacts should be meaningful parameters to the method. However, the ID argument has pass-by-reference semantics. The method will place this argument in the buffer provided during its execution and return it to the calling code.

When compiled using the IDL compiler, two corresponding parts are produced from the IDL specification: the stub and the skeleton. These represent the “wiring” which needs to be in place for two components to interact. The stub is associated with the client code and the skeleton is coupled with the server implementation. The stub and skeleton code act as proxies for the method invocations from the client to the appropriate implementation routines of the server. Figure 1.1 shows how everything fits together when there is only one client and one server component.

One important goal of component oriented programming is to support distributed objects on any platform using any OS. To achieve this, the stub and skeleton are coupled with all of the necessary marshaling/unmarshaling and networking code. The work by Szyperski [41] gives an excellent overview of components and various commodity component models.

People often confuse the defining features of components with that of objects. The main difference between them is that components are individually deployed and exist as self-contained entities while objects might not be. We outline some of the basic characteristics of a component that we assume in this thesis. These capabilities are simple and observed by all component models we are aware of. They are the following:

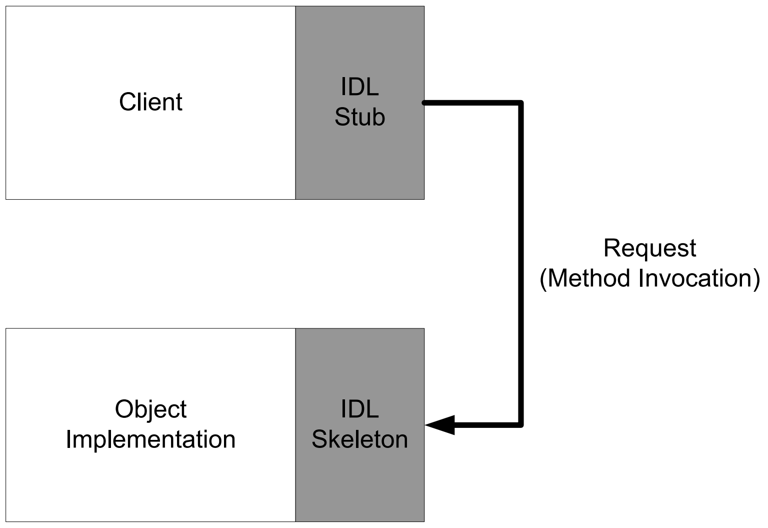


Figure 1.1. A remote method invocation (RMI) of two components.

• **Clearly specified interface.** A component has an interface clearly expressed in an IDL.

• **Independent deployment.** Each component is a relatively independent entity. If a component is dependent on outside entities (e.g., virtual machines, web services, etc.), we ignore this dependency.

• **Following a component model standard.** Components belonging to one component model contain regularities in their design.

1.3 Motivation

Consider the problem of component interoperability in the field of scientific computing. In recent years we have seen increasing growth in the size of the average simulation due to increases in computing power. Legacy software that took a long time to develop and grow is pervasive throughout many scientific fields. Modern scientific computing attempts to tie in related simulations in order to produce combined simulations with greater accuracy and effect. For example, scientists need to combine oceanic simulations with atmospheric models in order to get more accurate models of global climate change. Ocean and atmospheric simulations are quite complex individually and rely on many software components [6].

Several software frameworks are targeted at high-performance scientific computing applications of this nature, including SCIRun [34, 26], SCIRun2 [46], Cactus [24], and ESMF [6]. These systems enable creation of complex high-performance software through the assembly of software components. In each of these systems, facilities are provided for efficiently communicating with other components in the same model. However, these frameworks produce islands of functionality; they are difficult to operate with other component-based software, such as sensor networks, databases, spreadsheets, and so forth. Furthermore, these frameworks seldom interoperate with other frameworks with the same design goals. These challenges are not unique to the domain of scientific computing; many component models have limited mechanisms for interactions with other models.

One motivation for this work in component interoperability is the creation of an environment that facilitates the coupling of software components from a variety of scientific domains in a single high-performance application. For example, parallel components based on CCA could be connected to dataflow components based on SCIRun or other models. In this manner, the scientist is able to use the "right tool for the job" in a flexible software framework. In climate modeling, this enables scientists to tie in several simulations of physical phenomena in order to produce a high quality result.

Another motivating scenario for component interoperability is common to many domains. It arises when an application needs to use one or more heterogeneous resources, such as software components, web services, or software libraries. Although not all of these are component based, each of these resources is accessed through a well defined interface. We tie ourselves to that specific interface when writing software that uses such resources. In reality interfaces can change, resulting in the need to modify our implementation. New competing tools may arise that use completely different interfaces. We can use the techniques and tools described in this paper to handle incompatibility between interfaces even if the software is not strictly component-based.

1.4 Problem Analysis

The problem of component incompatibility when simplified is analogous to differences in electrical plugs when traveling. If one is traveling from Switzerland they would need a simple plug adapter to make all of their electronic equipment usable in Sweden. This adapter would simply provide a conversion from the three pins used in Switzerland to the two used in Sweden and most of Europe. If a traveler, on the other hand, was arriving to Sweden from the United States then the electricity conversion would not be as simple. The traveler would not need only a plug converter, but also an additional power converter to shift the U.S. voltage of 120V to the European voltage of 220V. In similar fashion, two component instances may be different from each other through their interfaces (or plugs) or they may differ in component model (or voltage); they may require a simple adapter or a complete conversion utility. We separate the discontinuities between two component instances into interface and component model based. These are independent in the sense that two component instances may differ in any one of them or both.

Differences among interfaces can range from minor incompatibility in argument names to complete argument and method chaos. Combining any two arbitrary component interfaces may not always make sense. The goal of interoperability is to make two compatible components communicate and not to impose communication where there is no such basis. Therefore, we are interested only in interfaces that are naturally connectable, so that the semantics of the server is what the client is requesting. As such, interface discrepancies are something that can be expressed and remedied through mapping. Later we will present our interface mapping directives aimed at mitigating the problem of interface incompatibility.

Component models act as abstraction layers between component instances and the outside execution environment. Component model differences can be varied and numerous. A component belonging to one component model depends, at a basic level, on that model for naming and discovery, to provide a component communication protocol, and to define data types. Naming and discovery enables server components to publish their existence and client component instances to discover servers whose services it needs to use. This process usually takes place independent of the physical location of each component instance. A predefined interface supplied by the component model is used to activate the naming or discovery functionality. These interfaces are specific to each component model. Component intercommunication is based on a predefined protocol. A communication protocol is sometimes well defined, such as in CORBA, and sometimes hidden and difficult to decipher. Component models can also be based on different means of execution control. Examples of control paradigms are (remote) method invocation, events, software bus, dataflow, and so forth. Control furthers the discrepancy of component models and complicates the semantics of their combined functionality. Component models often define special ways to handle more complex data types, such as objects or arrays. These datatypes should all be accommodated. In addition, some other defined types may be used that are specific to a particular domain. For instance, a visualization model, such as the Visualization Toolkit (VTK) [38], may use colormaps, contour data and others as predefined types, but others like EJB do not.

To combine two existing component instances, we need to be able to translate between all the possible differences mentioned above. Since the differences between component models can be vast and complicated, an approach with maximum flexibility and expressibility is needed. To this end, we choose to interpose a piece of code that translates between the components. Our plan is for this bridge code to perform the translation task without modification to the existing components. In turn, this will allow bridged component instances to be available for use both through the bridge and directly by completely compatible components. Using bridge code between two components has been used in many similar efforts (such as language interoperability or object-oriented interoperability). We will treat this bridge code as a separate entity, although in practice it can be easily coupled (entirely or in part) with the client or server component. In the following we expand on the design possibilities and justify our decisions by categorizing the types of bridge code.

1.4.1 Component Bridging vs. Standardization

A decision in component bridge creation involves the use of an intermediate component representation (or protocol, like Simple Object Access Protocol (SOAP)). An intermediate representation would act as a common language between component models. A benefit of the intermediate approach is that a bridge would only need to specify a complete mapping from a given component model to the standardized intermediate representation. Once all component models have this bridge enabled, an arbitrary combination of component instances can be made through the intermediate representation. This is a very powerful possibility as it enables faster growth of the number of components from different models available to be used. The other alternative is to directly bridge two component instances without regard for an intermediate representation. The intermediate representation approach can be seen as a specialized case of this direct approach. The direct approach is more flexible and does not require a specific intermediate representation. A chosen intermediate representation would need to be all-inclusive, which may not be easily achievable for many component models. In order to allow the most flexibility we base our design on creating bridges specific only between two component models. We allow users to choose a special intermediate representation, model or protocol but do not require them to use one that we have predefined.

1.4.2 Static vs. Dynamic Bridge

There are two approaches in designing the structure of a component bridge: static and dynamic. The static approach requires the system to create a bridge instance based on specific client and server instances. It requires a new bridge to be created (or code generated) per every two instances, based on their static interface information. Parsing of IDL (Interface Definition Language) specifications is necessary to extract the interfaces. Another possibility is not to use these single-purpose bridges, but to design one bridge component instance able to handle the bridging of all component instances belonging to two disparate models. This dynamic bridge would be able to perform the bridging without any previous knowledge of the component interfaces. This is possible and relatively straightforward in some component models and it requires the use of generic stubs and skeletons. In CORBA this would be done through the Dynamic Skeleton Interface (DSI) and Dynamic Invocation Interface (DII). DSI and DII are intended to be used when no interface information is available, such as in this case, and are quite costly in terms of performance. The downfall of the dynamic bridge is that it is not possible to implement in many other component models; a significant disadvantage in combining component models. To keep our tool general and the bridge it generates fast, we have opted for the static bridge approach.

1.5 Design Overview

Given these different practical alternatives in component interoperability, we define our task to be the creation of a static, direct and independent bridge between two existing component instances. We made these design choices in order to emphasize flexibility that would enable our approach to be used in a wide range of scenarios. Our bridge acts as a translating entity between component instances belonging to two different component models. To perform this task the generated bridge code exists between two cooperating component instances and mediates their communication.

We choose a user-driven approach that enables flexible and easily attainable bridge creation. Our method consists of an interoperability framework that combines two complementary but separate approaches to generate a bridge enabling interoperability: a bridging domain-specific language and a bridge code generator. These approaches are intended to automate different aspects of the task, each exploiting regularity within the structure of component software. This regularity we exploit enables automation and rapid creation of bridge code, which we implement in our approach. The bridge code generator focuses on easily producing bridges for different interfaces, once component model differences are understood and expressed in a particular format. On the other hand, the bridge domain specific language focuses on defining component model features and enabling easier creation of a ontology between sets of features in component models. We combine the abilities of these two complementing tools into a inclusive and effective paradigm for fostering interoperability between incompatible software models.

1.6 Applications

We applied our interoperability framework to a varying set of applications. Our intention was to demonstrate practical situations where our methods can be used while verifying that our toolset is applicable to many problem scenarios.

A scientific application of significant size was selected, such that it combines computational problem solving and visualization. This application requires several software tools based on components in order to produce a solution that was optimal for the end-user. It solves a heat distribution problem that uses the Finite Element Method to approximate the Laplacian equation and it is implemented in several component models: CCA, CORBA and VTK. The component models were bridged successfully using our interoperability tools enabling a three-dimensional visualization of the solution. We discuss this effort and results in more detail in Chapter 6.

We also pursued interoperability between two linear solver mathematical software libraries. This was an effort to determine the usability of our tools outside the realm of component software, while enabling the use of software applications written for one numerical library to use the other without any modifications to the application itself. Our code generator proved effective in creating code that communicates between the two sets of interfaces that the linear solver libraries provide. We leveraged the interface mapping directives and data models of our code generator to specify the differences and make library interoperability easier to perform.

1.7 Contributions

The approach described in this dissertation is the only generative method for scientific component cooperation known to us. Several other projects exist that have succeeded in enabling component model interoperability in other ways [32, 37, 47]. These are usually not appropriate for the scientific domain and are not as flexible.

Our method introduces a bridging domain-specific language based on meta-modeling techniques that adds semantic information to the bridge generation process. We design meta-models that succinctly describe the generative features of each component model. This work is novel in that it expresses software models and their differences for bridge code generation to enable software interoperation. Reusable meta-models enable a reduction in the lines of code needed to make two component models interoperate.

Our code generator is significant because it addresses several axis of differences between component models: data, interfaces, and communication protocol. The code generator provides the user with an abstraction for each axis that can be used to express complex bridge scenarios. The code generator is tailored to achieve economies of scale in producing bridge instances that bind components belonging to a pair of component models. In our experience, the code generator induces a several fold reduction in effort over the course of a project lifetime compared to hand-coding a bridge.

1.8 Thesis Organization

The related and background work is given in the next chapter. In Chapter 3, we discuss the design goals of our implemented system as well as a broad overview of the system parts: the SCIRun Component Interface Mapper (SCIM) code generator and the semantic tools. Chapter 4 contains a detailed discussion of the code generator, and Chapter 5 focuses on the design of the semantic tools. In Chapter 6 we present the results of applying our component interoperability system and we analyze some of the overheads of the system. Chapter 7 evaluates the merit of this dissertation in supporting the thesis statement. In this chapter we also analyze the value of our system to the end-user. Chapter 8 contains the conclusions and future work segments of our approach.

1.9 Thesis Statement

A broad and practical method to express differences between software models into a specification can be leveraged for the creation of a bridge between two specific heterogeneous software applications belonging to separate models that supports efficient application execution.

CHAPTER 2

RELATED WORK

Related work of this dissertation fits into three broad categories: code generation, component modeling and other implementations of component model interoperation.

2.1 Code Generation

Code generation is often a useful utility that can be used to raise the programming abstraction in a particular domain and eliminate writing repetitive code. It introduces quality and consistency in the software, while reducing the time spent on writing detailed code and increasing the time available for planning and design. We use code generation in component software interoperability for the above-mentioned benefits. Our approach reduces the time spent writing bridges between components belonging to the same set of component models. Several research projects use code generation to combine disparate computer applications into a larger application. Many tools exist that enable multiple approaches to code generation [23, 25]. Likely the most visible of these is SWIG (Simple Wrapper Interface Generator) [9, 39]. SWIG is a compiler that parses interfaces written in C/C++ and generates means of translation between a C/C++ implementation and a specific scripting language (such as Perl, Python, Ruby, and others). It generates a translation that takes the form of C code and mediates all scripting language invocation of C/C++ routines in the original application. This method allows the user to easily communicate between a programming and scripting language based upon a common interface. The end goal of SWIG is to provide easy manipulation of large monolithic codes written in C/C++ using a scripting language. In addition to the C/C++ interface, SWIG accepts several commands as input that enable easier manipulation of the application by the scripting language. The commands can be very useful when scripting language references are needed for objects and pointers. The approach of our code generator follows SWIG in that it offers user commands to fine tune the code generation, though the tools differ in purpose and design objectives.

Often the debugging of applications using SWIG wrappers can be difficult. To remedy this, the creators of SWIG offer an additional debugging tool called WAD (Wrapped Application Debugger) [10]. The work of this dissertation does not include support for the debugging of component bridges, which can produce similar difficulties in debugging as wrappers. Therefore, a debugging system would be a valuable addition to which end the WAD approach should be closely examined.

Component software has been a popular area of research for several years. However, not many research projects have focused on devising methods of interoperation between different components. The Concert multilanguage programming system [4] uses a component software methodology that places its IDL as an intermediate language and allows the specification of components to be performed by extensions to the programming language. By allowing language extensions and providing a special Concert Contract Language (CCL), this programming system is able to provide interoperability between the same abstractions written in different programming languages. The language used to specify this interoperability between components has been of interest to our project since it introduces a method to abstract common features shared by most component models. Our approach is similarly concerned with leveraging the regularities in the design of component software.

Generative Programming (GP) is a methodology that shifts the focus from individual systems to entire software families. This broad description encompasses several specific popular research topics such as Aspect-Oriented Programming (AOP) and Domain-Specific Languages (DSL). The GP paradigm bases itself on a generative domain model that models an entire system family [14]. A generative domain model consist of problem space, configuration knowledge and solution space. The solution space contains components that form the implementation, while the problem space consists of concepts and features used to model a systems family. The configuration knowledge acts as a map between the problem and solution spaces. In an abstract way, the GP approach is the approach used in this dissertation. However, GP literature is not concerned with interoperability and defines the GP of components as the automatic selection of components to form an application on demand[15]. It advocates that well-designed components should be connected automatically into an application. In this way, the user can be concerned only with the final product and not with the components that constitute it.

The Babel [27] compiler is aimed at facilitating components following the CCA [3] component standard. It is interesting because it is designed to create glue code for interoperating components based in several programming languages: C, C++, Java, Fortran, and Python. The purpose of Babel is to enable the creation, description, and distribution of language independent software libraries for scientific computing. This project faces component heterogeneity, albeit on a different axis than our work, and enables normally incompatible software written in different languages to communicate.

2.2 Component Meta-Modeling

Architecture Description Languages (ADL) [1, 16, 31] are languages that can be used to describe and manipulate the architecture of a system of components. They describe the connection and interaction of components that form an application. Most ADLs also represent a formal basis for component interoperation. We investigated several ADLs and concluded that each one's goals were inconsistent with ours; making the use of an ADL as our component meta-model inappropriate. We are interested in creating a component meta-model amenable to fast expression and code generation, while ADLs are more tedious to use and express detailed system architecture and behavior. This is why we choose to develop our own component meta-model that is most amenable to an expressive and generative approach.

Model-Based computing [21] contains a similar high level description of a system, but this description is more commonly leveraged to subsequently generate an application. Similar to most definitions of generative programming, model-based computing often depends on implementation of basic functionality within components and configures and connects the components to form the described application. Model-based computing is commonly used in embedded systems programming.

Ptolemy [11] is a component system also intended for embedded systems. It allows the use of multiple models of computation that provide a wide array of possibilities to express a problem domain. The Ptolemy models of computation are fundamentally different between each other and include: dataflow, time triggered, synchronous, discrete events, process networks, etc. The design of the interaction of these different models [29] to form a Ptolemy application is of significant interest to this project. We are interested in combining component models that may rely on different models of computation. The use of fundamentally different tools that follow a prescribed regular structure to solve a problem is common with the goals of our project. However, Ptolemy's computation models are ingrained in the system and do not need to be artificially bridged in order to communicate. Also, Ptolemy is a very low-level tool and we are interested in combining component models using bridges specified in a high level language. Though not directly applicable, there are several common goals of interoperating between different software architectures between this dissertation and Ptolemy. In the current design of our software, we do not plan to use any of Ptolemy's interoperation paradigms; however, these might become useful at a later point.

A paper by Floch[19] describes an automatic code generation methodology meant to generate source code from high level functional (i.e., behavioral) and implementational (i.e., structural, describing the concrete system) descriptions. The approach stresses flexibility and is capable of easily adapting to programming language or hardware changes. It is similar is spirit to DSLs and the emphasis to flexibility is much like our approach of generating code generation templates based on specifications of different component models.

Meta-modeling is simply the modeling of models. Component meta-modeling is of interest to this project, so we investigated existing approaches to it. Several attempts have been made in this area. Rastofer [36] defines a component meta-model suitable to define both component instances and the component model. The abstraction used is simple but effective and several popular component models are modeled. The approach concentrates on modeling the component model structure, but does not consider modeling the behavior. A formal model that defines the behavior and in-depth details of the Enterprise Java Beans (EJB) component model is provided in the work of Sousa et al. [40]. The authors performed formal validation on the structure of EJB and were able to find several inconsistencies. The Eclipse Modeling Framework [17] provides a simple meta-model for software description called ECore. This model is a subset of the Unified Modeling Language (UML) [43], which is itself a meta-model, that is aimed at modeling the class structure of an application. It is used to represent user code as it is being written within the Eclipse framework.

2.3 Component Bridging

The interoperability in component software builds on previous work on object-oriented interoperability. Object-oriented interoperability is an research area that is concerned with the interoperation of objects written in different languages and based on separate interfaces. It was investigated by a few projects [28, 45]; however the research topic seems to have gotten less attention over the years. Some of the concepts used to solve the object interoperability problem also apply to components. Our project attempted, used and learned from the approaches in object-oriented interoperability. Many similarities exist in specifying the mapping of interfaces, data, and in other aspects of the problem. We adopted several approaches that were still applicable to component software. However, features specific to components require a new set of interoperability problems to be solved. In fact, component models themselves were considered a solution to the problem of object-oriented interoperability.

Probably the most visible component bridge is the one built between the COM and CORBA component models [22], which essentially connected applications built and tuned for the Windows platform to ones built on the industry-standard any-platform CORBA specification. The designers differentiate between two possible bridges between COM and CORBA: static and dynamic. The dynamic kind is more general as it opts to use generic data types and invocation reflection. These special functionalities are not available in many component models. The static approach is the one described in this work, where a compiler/code generator places bridge code between two components to facilitate the communication. Our approach is an extension and a generalization of this work aimed to make a bridge between any two given component models.

Uniframe is a component framework whose goal is to build interoperable distributed computing systems [35]. More specifically, applications in Uniframe are built by assembling predeveloped heterogeneous software components. The framework accomplishes this by the use of the unified meta-model to constrain the features of each component model whose component(s) it contains and by generating glue/wrapper code to directly connect components. The unified meta-model provides a formal way of describing and comparing components and component models. It is also constructed to provide a QOS service guarantees, which represents an additional goal of the Uniframe framework. The glue/wrapper generation is performed by explicitly modeling the technology domains that are encompassed: component model domain, programming language domain, operating system domain etc. The technology domain specifications are combined with meta-model specification of the component model to create the glue/wrapper code [47]. The generated code is able to translate to and from the SOAP protocol for any two component instances.

The protocol enables the interaction between the instances. The goals of the Uniframe project are very similar to that of this dissertation. Our approach differs from Uniframe's in several ways. One aspect is that Uniframe is interested in bringing component models to use a common protocol (i.e., SOAP) and we are focused on combining component models directly. This decision makes the integration of each new component model to be reduced to bridging it to SOAP. Our system is able to work in this way, but it does not impose any specific protocol. We also see this as a possible performance optimization for our system in situations where a common protocol isn't required. It allows a direct efficient bridge to be tailored to fit only two component models. Another difference is that Uniframe's approach is tied to the use of their framework, while the approach we chose is more general and should be applicable to many framework domains.

The Component Mill [37] has similar general goals as the work in this paper. It describes a container that can be used to integrate applications belonging to different component models and defines several XML data structures that are used to facilitate the communication across component model boundaries. It also broadens the definition of a component, as our project does, to include most software that contains an interface including data. Component Mill focuses on providing a container for interoperation, and we focus on creating bridges between existing component interfaces. The creation of bridges (called adapters in Component Mill) is not discussed in their current effort. The adapters are to be provided from an outside source, presumably the user. It is conceivable that both tools can be used in unison with SCIM providing the adapter generation for Component Mill. This would be an interesting prospect as it could combine the capabilities of both systems into a larger comprehensive integration framework.

The creators of the ASSIST [44] parallel computation framework devised a binding between ASSIST programs and the CORBA component framework [32]. The bridge between the two environments is generated by a custom compiler. The ASSIST program takes the form of a component containing a single `execute` method within a CORBA framework. CORBA components are able to invoke the `execute` method to interface with the parallel ASSIST program. An asynchronous scenario that uses events for communication with the parallel program is also provided. The interoperability of ASSIST and CORBA is an example of a hand-written bridge scenario for which we try to provide abstractions that facilitate easier conception.

TENT is a CORBA workflow management system that attempts to interface multiple types of application by enabling the creation of wrapper components [20]. The wrappers range from components utilizing CORBA Dynamic Skeleton Interfaces (DSI), which are able to communicate with any CORBA component, to "black-box" OS mediated communication. TENT falls in a broad category of scientific workflow management applications whose goal is to manage the execution of simulations spanning many disparate and widely distributed (usually data-centric) resources. Due to this, their design goals and interoperation approach rely on a set of hand-written adapters produced for each new resource. Many software systems intended for scientific workflow management [2, 30] utilize concepts of binding different applications, databases and web-resources together. Our approach could provide these sets of interoperability application with a consistent method of expressing and generating cooperative software. A goal of our project in the future is to attempt interoperability for several of these applications by using the set of tools for component model interoperability that we developed.

CHAPTER 3

SYSTEM OVERVIEW

Given the different practical alternatives in component interoperability that we discussed in Section 1.4, we define our task to be the creation of a static, direct and independent bridge between two existing component instances. In the remainder of this text, we will focus on describing a specific approach to component interoperability. This will demonstrate that we can build a framework that makes bridge writing possible and practical. Our approach is composed of four connected parts (see Figure 3.1). The main part is a code generating utility. This is augmented with interface mapping directives and data models. The purpose of the interface mapping directives is to create an ontology between the interfaces of component instances. Data models, assisted by the interface mapping directives, create a definition of the data types and their handling per each component model. The fourth part is a prototype of a semantic bridge creator based on meta-models of component models and a bridge domain-specific language, which adds semantics to component model interoperability in addition to the bridge code syntax. Coming full circle, we use the output of this meta-model tool as template input to our code generator.

In this chapter we provide a high-level description of our system for component interoperability. We begin by discussing the design goals that guided us through the implementation followed by the underpinnings of the code generator SCIM. Finally, we describe the semantic tools we have designed: the meta-model and the bridge language.

3.1 Design Goals

The primary goal of this work is to produce a stack of tools that will enable component interoperability through the means of a bridge. In the process of completing this practical task that is valuable in many software application domains, we want to contribute to the general understanding and modeling of component models.

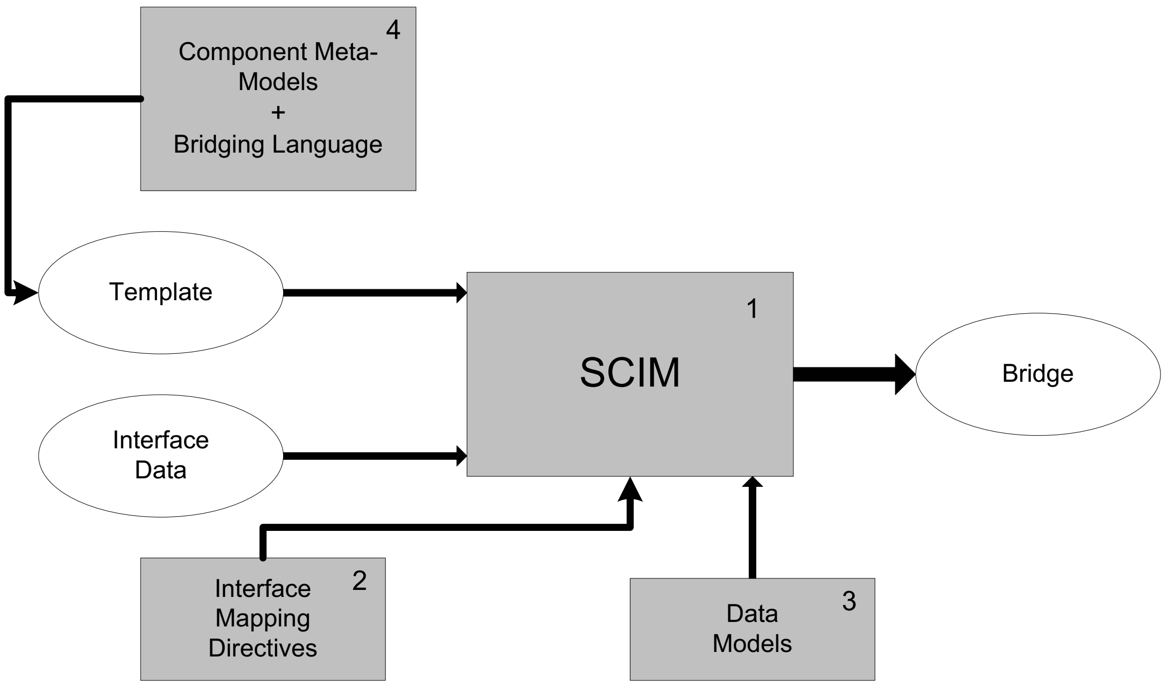

4
Component Meta-Models + Bridging Language
Template
1
SCIM
Bridge
Interface Data
2
Interface Mapping Directives
3
Data Models

Figure 3.1. Four parts to the system used for bridge code creation.

The main set of design goals that guided our approach to component interoperability are:

• **Inclusive.** Accommodate the interoperability of any two software components, component models or frameworks.

• **Model-centric.** Standardize modeling of component models for this particular domain and purpose. This would provide a framework to express all relevant implementational features of a component model.

• **Automatic.** Enable the automatic creation of a bridge between two component instances, once component model differences are expressed.

• **Easy to use.** Keep user supervision to the minimum, while preserving a large degree of inclusiveness.

• **Non-prohibitive performance cost.** Achieve acceptable bridge performance roughly proportional to an additional function invocation in the target software domain.

3.2 SCIM

As a subset of our broad goals, we defined a subset of design principles for our bridge code generator SCIM:

• SCIM should accommodate generating bridges for any two component models.

• SCIM should be able to express different kinds of data that may be passed across heterogeneous component boundaries.

• Component instances whose interfaces differ syntactically but describe a similar abstraction should be combined.

• In order to include more component models the code generator should have an ability to interpret as many interface descriptions as possible, including source languages such as C, C++, and FORTRAN.

Our approach to interoperability is to automatically interpose bridge code between two incompatible component instances. We use the SCIM (SCIRun Component Interface Mapper) code generator to automate the generation of the bridge. Code generation is the most logical possibility for static bridges, as a new bridge instance is required per every two component instances. SCIM parses an a component interface description and uses an output template specifying a generalized version of the bridge code. Producing code for pairs of component models requires a separate output template, though many templates involving a single component model may be very similar. A template should be written by a software engineer well versed in the designated component models. SCIM's code generation process is made more powerful by including script language code (snippets of Ruby) within the template.

When performing code generation, it is a common goal to justify that the code generation is useful by means of its large frequency of use. We have a similar design goal in mind: by insisting on general SCIM output templates we hope that we can generate suitable bridge code for many different components instances belonging to the specified models. A related design goal is increasing the usefulness of SCIM by allowing as many interfaces to be easily included in the tool as possible. We have accomplished this through including several input languages (C++, Fortran, Scientific Interface Definition Language (SIDL)) and by allowing the mapping of two similar but not syntactically equivalent interfaces.

Figure 3.2 provides a very simple example that demonstrates a user's interaction with SCIM. Apart from interfaces and an output template, SCIM takes in two optional inputs:

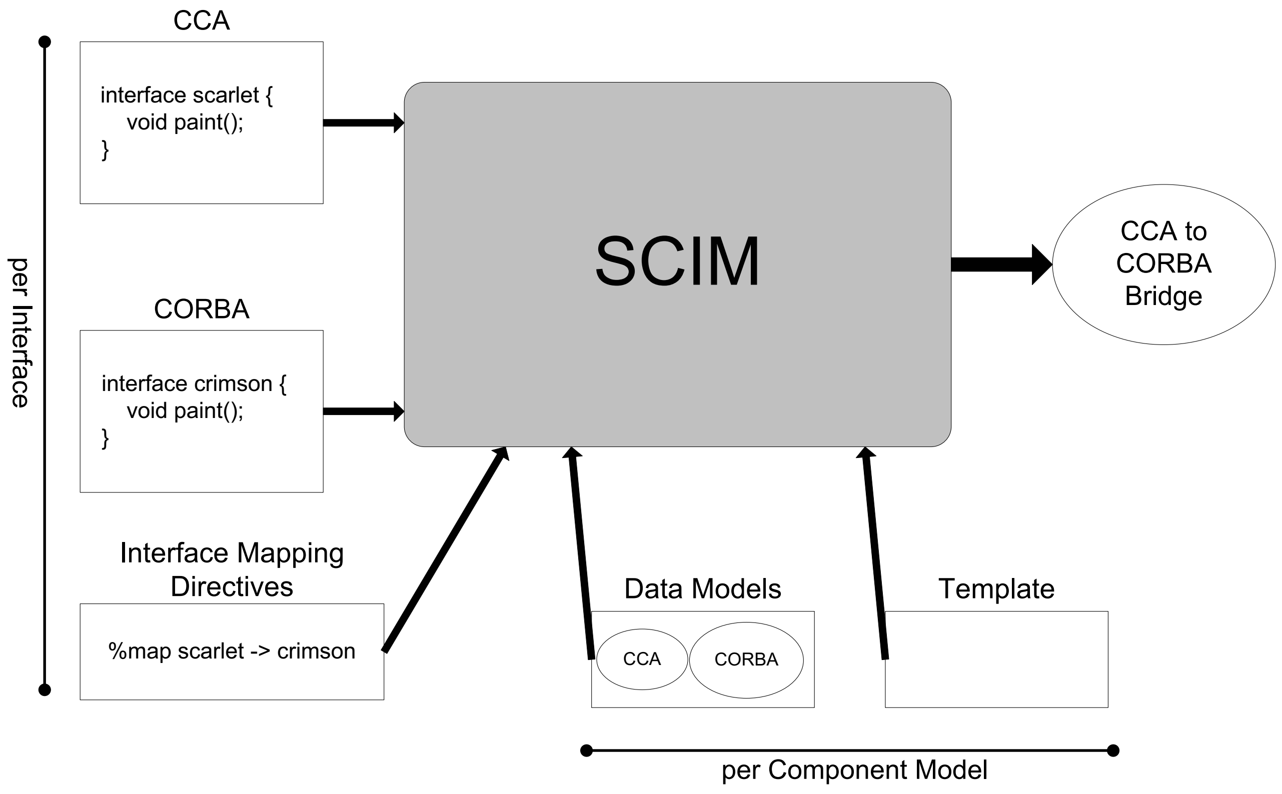


Figure 3.2. Simple example of creating a CCA to CORBA bridge with SCIM.

interface mapping directives and data models.

SCIM accepts a series of interface mapping directives that can be used to create a match between two interfaces. The interfaces are similar semantically but because they belong to different applications in different component models they may differ syntactically. The interface mapping directives are very useful in many situations when two components have interfaces that contain discrepancies in names or types of interfaces, methods or arguments. SCIM is able to create bridges that affect multiple interfaces, and therefore it needs to be able to process lengthy interface files. For instance, when parsing C/C++ code SCIM can follow a trail of include files containing many interfaces that a user is usually not interested in. SCIM parses each interface it encounters and stores it in intermediate representation form producing interface bloat that can bother the user. Some of the interface mapping directives provide a way to select or exclude groups of interfaces for the code generation done by the compiler back-end.

Data model classes are used during SCIM's code generation step whose purpose is to generate bridge code responsible for translating the data between two component instances. One could have easily encapsulated the data models with the SCIM template and kept them away from the end-user. However, the ability for the user to extend these classes to define special rules for a given component model is very valuable. Since data rules exist on component model basis, a data model should exist for each component model. Each data model contains a mapping database from IDL types to types specific to a given programming language. It also allows the expression of translation functions and data rules that are converted into the translation part of the generated bridge. Although code generation can proceed based on a generic data model, it will not produce the right result in many cases. Our implementation also duplicates some of the capabilities of the data model classes in the interface mapping directives to provide flexibility to the user. The interface mapping directives are also used to specify extra information to the data model if necessary.

SCIM, its interface mapping directives and data models are appropriate for code generation of component bridges. However, SCIM is not straightforward to use and requires that a knowledgeable software engineer bootstraps the code generation by creating a template and data models. The software engineer needs to understand SCIM as well as both of the bridged component models. This is a limiting factor in the wide use of the tool. Furthermore, SCIM itself does not have any semantic knowledge of what it is generating, which makes the process become very detail oriented and error prone. As mentioned before, SCIM is great at achieving economies of scale once the bootstrapping of the bridge is finished, but SCIM also has large entry barriers, requiring a large investment of time and resources. To reduce these problems, we attempt to provide a framework that would be used to describe the semantics of component models and be able to generate a SCIM bridge template. This framework consists of two co-dependent semantic pieces: a component meta-model and bridging language.

3.3 Semantic Tools

The purpose of the semantic tools is to provide a high-level design tool that raises the design abstraction of generating component bridges. A bridging language provides the user with the ability to worry only about the logic of the bridge domain, ignoring for a moment the minor details of the underlying component models. Comprising the bridging language is a set of commands and logic that can express the high-level structure and behavior of a bridge. A small number of commands written in this language can produce a large effect and many lines of generated code.

The semantic tools also consist of a component meta-model that is used to express each component model. This generative meta-model contains structure used to generatively define a specific component model. The bridging language is based on the component meta models and expresses how two or more meta-models interoperate. These tools are very closely intertwined: the bridging language depends heavily on component meta-models in order to describe a bridge (see Figure 3.3).

Meta-models and a domain-specific language, such as our bridging language, are used in many domains [42]. Similar to our use of them, they are commonly leveraged to provide a high-level abstraction of a particular domain used to lower the bar for writing programming logic specific to that domain. Our domain is particularly dynamic in that we are attempting to bridge any two component models. This, in turn, requires that a new meta-model exists per every component model. In most other domains one or more meta-models are written early on and remain for the duration of using the domain-specific language. Our approach can often be costly in terms of time spent by a software engineer knowledgeable of a particular component model. However, this cost can be amortized by building several bridges between existing meta-models. Each new meta-model written in the system multiplies the number of theoretically possible bridges. Meta-models also provide a structured way to express a component model, as opposed to a SCIM template that requires greater trial and error. In the end, we claim that our choice of use of this paradigm to represent and define component bridges is valuable and beneficial.

Several principles guided our design of these tools:

- They should be able to express any component model and any component bridge.
- The tools should be as easy as possible to use.
- The component meta-model should provide a comprehensive set of information to differentiate component models and generate a bridge.
- The bridge language should be processable in to a SCIM bridge template or directly in to a bridge instance.
- The bridge language should be able to express a bridge spanning any number of component models.

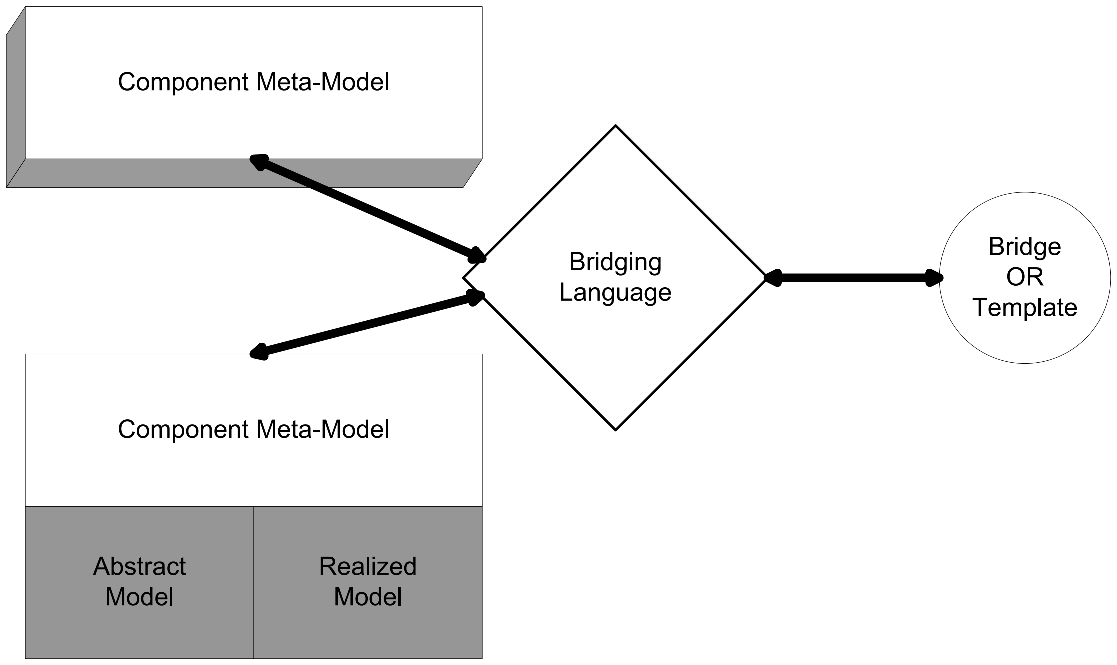


Figure 3.3. Two component meta-models coupled in bridging language definition can be processed into bridge code or a SCIM template.

3.3.1 Component Meta-Model

A component meta-model is simply a model of a component model. To define a component meta-model we use two models that create a mapping between the component software world and the programming language world: an abstract model and a realized model. By defining this mapping we introduce a generative component meta-model that is able to express how features of a specific component model are represented by programming language code. In expressing a component meta-model we aim to abstract the defining features of a component model. For instance, we want to capture that expressing a component in the CCA component model means implementing a class that inherits from the `gov.cca.Object` interface. On the other hand, features that are individual to a subset of all components are expressed as another case (or type) within the component meta-model.

The abstract model part of the component meta-model is concerned with providing a basic prototype of the structure of a given component model. Its purpose is to express various structures and features that a specific component model may have. These structures may include the framework, components, and various ports that a component model contains. By ports and components here we mean types (or categories) of these that may exist within one component model. For instance, in CORBA we may differentiate between event-based and RMI (remote method invocation) ports. Each of these should be expressed as individual ports in the abstract model. As mentioned above, if a special asynchronous RMI component instance exists in CORBA it will require a new port structure to be defined. Naturally, this need be included only if the user intends to generate to code that will interoperate to or from that specific component instance.

Each of the abstract-model's parts, in turn, uses the realized model to express their inner structure. The realized model is a high level abstraction of object-oriented programming language concepts. It specifies constructs used to provide abstractions of classes, methods, and other language structures all the way down to lines of code. The realized model defines only a small subset of a general-use object-oriented programming language for expression simplicity, but allows embedding of target code as a catch-all. A complete abstraction of the underlying programming language may be useful to provide complete detachment and enable pluggable programming languages. However, this requires a complete model able to express all features of all programming languages which is an endeavor that we wanted to avoid. Therefore, the realized model is only a partial abstraction.

The relationship between the abstract model and the realized model within the component meta-model is meant to describe how component model abstractions correspond to an object-oriented language. A user is required to express features of a given component model into the abstract model and make appropriate links to the realized model. These links are the crucial part of expressing our generative component meta-model.

The user expresses the abstract model in terms of a component framework, components, and ports using a hierarchical set of abstract classes. Our implementation prescribes that the user extends a set of base classes to represent the abstract model. This set of base classes is self-explanatory and contains: `Abstract Framework`, `Abstract Component`, `Abstract Port` and `Abstract Data Model`. The `Abstract Data Model` class exists as a way to begin specifying the data model at this abstract level with the help of the bridge language. By providing a data model handle in the abstract model, we are able to begin the description of certain data types and their translation from within the semantic tools. The `Abstract Framework`, `Abstract Component` and `Abstract Port` classes are all intended to be extended appropriately to model various frameworks and various categories of ports and components.

The realized model is closer to a programming language description than the abstract model. The abstract model contains constructs that are common in component models, while the realized model contains similar constructs common in an object oriented programming language. Some of the structures of the realized model are: `MetaModel`, `MetaClass`, `MetaPort`, `MetaMethod` etc. These structures are instantiated and contained within the abstract model structures. Figure 3.4 depicts the relationship between units of the abstract and realized models.

3.3.2 Bridging Language

Our bridging language is used for the purpose of expressing a bridge between two (or more) component models. The language lends itself to describing both trivial and complex bridging scenarios. It is powerful since it often requires very few lines of code to express complex bridges. The bridge language combines two or more component meta-models through event-based expressions to describe the semantics of cooperation between the models. The definition provided by the language is high-level, allowing the realized model that is part of a component meta-model to provide most of the corresponding low-level target code.

The bridging language we designed is relatively easy to learn and does not overwhelm the user with a large number of constructs. It is concerned with performing two main tasks: describing the structure of the resulting bridge and describing bridge behavior. The structure of the bridge defines where this resulting code will be used (within or outside of a component framework etc.) as well as the ports and structures of the bridge. We often use the structures of the involved component meta-models to extrapolate and define the bridge's structure. The bridge behavior is provided by describing the incurring behavior of component models upon given events.

Commonly the first action taken by the bridge language code is to acquire a base bridge structure upon which to build the bridge structure. To do this, the language contains a construct that acquires a specific framework's starting bridge structure. A base bridge structure is usually defined by an `Abstract Framework` in a component meta-model and often it is a subclass of `Abstract Component`. This also indicates that the generated bridge will exist within one modeled framework. Once we acquire a base

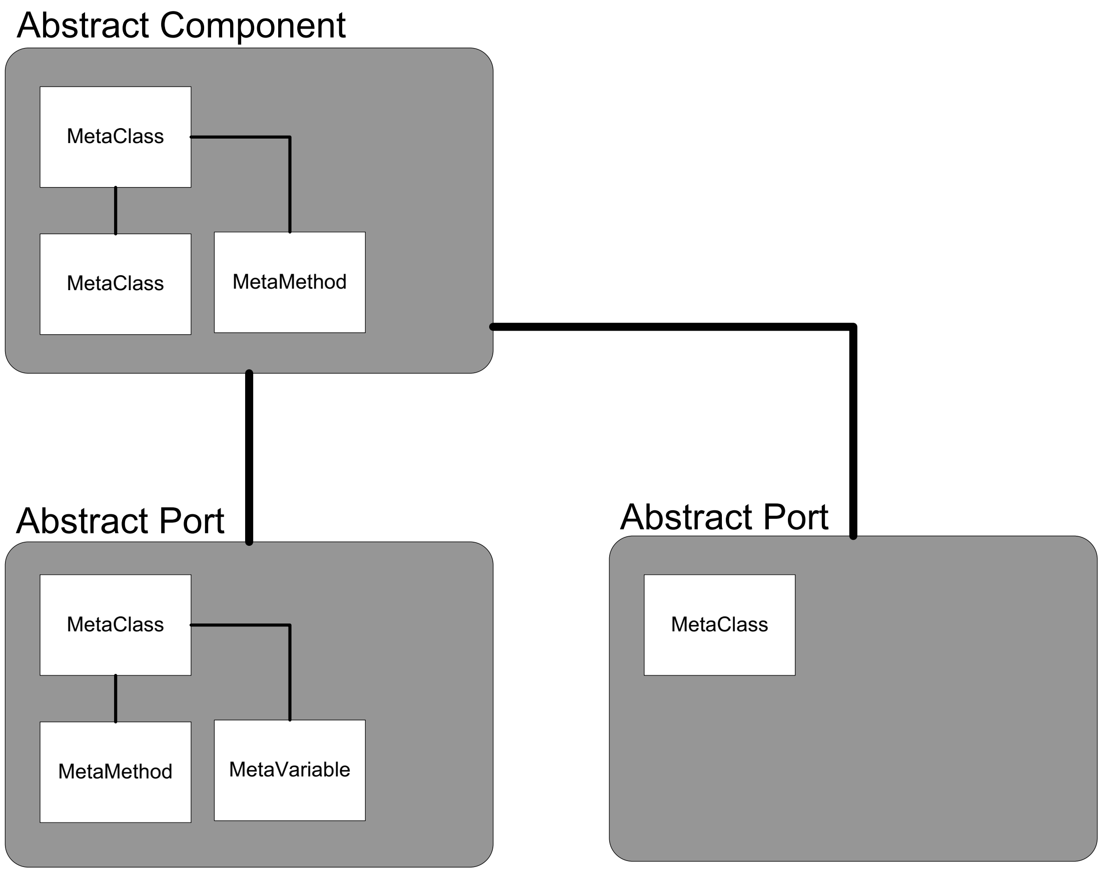


Figure 3.4. Abstract model's component and port contain Realized model structures to describe a component meta-model.

bridge structure, the language contains several methods to add ports or other structures to it. By doing this, we formulate the structure of the end bridge code.

After a bridge structure has been expressed in the bridging language, we focus on defining the bridge behavior. The behavior can be expressed using several common predefined events as well as by using a generic event call that enables the use of new events. By using various predefined events (such as `onInitialize`, `onExecute`, `onCreate`, `onDestroy` etc.) provided by the language we enable defining the behavior of bridge parts. For instance we may define that within the `onInit` event of a bridge's component, we initialize two specific meta-model ports. On a lower level, this may represent calling the constructor of the port classes within the bridge component's constructor. An `onMethodInvoke( methodName )` clause is provided that can be used to define the behavior of any specific method call, which corresponds to an undefined event. Many other definitions, methods and events exist that we will elaborate on in subsequent chapters.

The bridging language's event based approach provides us with the capability to define the behavior of many bridges using any number of defined component meta-models. Once the language specification is finished, we are able to process it into a bridge instance or a bridge template that can be used in SCIM. Within SCIM we can further modify this generated template, as well as use the facilities of the interface mapping directives and data models.

3.4 Bridge Deployment

The generated bridge code exists between two cooperating component instances because it needs to mediate their communication. This is the only requirement we impose on the bridge code, which allows large amount of deployment variability to the user of our system. Two choices exist: the bridge code can be compiled into the client or the server or loaded dynamically at runtime. The circumstances of the application can be a guiding principle into which deployment approach is better as ones would likely perform better than others in certain scenarios. In most of our use cases we place the bridge into a component instance of its own. This follows the component software design principle of component existing as separate entities. It also preserves the original component instances to be connected both through a bridge and independent of it, depending on the application.

CHAPTER 4

SCIM DESIGN

Several parts of SCIM are aimed to solve different aspects of the component interoperability problem. In this section we consider the design of several SCIM features: interface mapping directives, data models, and SCIM's template. These features enable SCIM's code generation and increase its utility.

Component interfaces are often tailored and used in different application scenarios. Components that implement a particular abstraction in one component model may use a different interface than components that use that very same abstraction in another component model. To combine interfaces that describe similar abstractions and yet differ in the way they were expressed, we use interface mapping directives.

Data can differ widely from one application to another. In a task where we are attempting to combine separate component instances, it is consequential to provide a way to handle data differences. Data models are a set of classes providing data handling rules for each component model. Populated with rules and information in SCIM's front-end, they produce data manipulation code during the code generation process.

The primary idea behind SCIM is the template concept. Templates consist of both static and dynamic information. The static information is plain text that is replicated on each bridge created with the same template. The dynamic information are snippets of programming language code that are executed on code generation to produce a value that is placed in the output. The use of SCIM's templates enables a large degree of flexibility in expressing code generation.

4.1 Interface Mapping Directives

The SCIM front end receives and processes interfaces written in a few IDLs or programming languages. When needed, differences between interfaces can be mapped to express that a match exists between two semantically similar but syntactically different interfaces. This ontology expressing mechanism is the main purpose of the interface mapping directives. However, we have also leveraged them to perform other tasks such as interface selection and data definition.

Interfaces that do not require a mapping are tagged as *inout*, differing with *in* and *out* interfaces that require an explicit mapping in order to surpass 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.

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 paint();
}
```

Two directives exist to denote whether a specific interface is *in* or *out* with respect to control. These directives are *%ininterface* and *%outinterface*. An interface designated as an *in* interfaces is intended to map to an *out* interface and vice-versa. If an interface is defined outside of the scope of either of these statements, it is classified as an *inout* interface. This means that the interface will map to itself, such as in situations where both components follow the same interface. An additional reason for *inout* interfaces is that they allow untagged interface files to be processed by SCIM if no explicit mapping is needed. In our example, we designate the component that is defined by the scarlet interface to provide a service, and the component behind the crimson interface to use a capability. Therefore, we define:

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

```
/%ininterface

%outinterface
interface crimson {
  void paint();
}
/%outinterface
```

To explicitly map between an *in* and an *out* interface, a special *%map* command is provided. As pointed out before, the command is only needed in situations where there are differences between the two interfaces. It states which differences are in fact just misnamed similarities and should be ignored. After the issue of the *%map* command, if the two interfaces match completely then they are adequate to proceed to the code generation back-end of SCIM. The map command for our crimson and scarlet interfaces would look like this:

```
%map crimson -> scarlet
```

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. If we create a slight modification to the interfaces in our example we can see how the map command is able to specify the mapping.

```
%ininterface
interface scarlet {
  void paintscarlet(in Object obj, out ScarletObject sobj);
}
/%ininterface

%outinterface
interface crimson {
  void paintcrimson(in Object obj, out CrimsonObject cobj);
}
/%outinterface
```

```
%map crimson -> scarlet
>paintcrimson -> paintscarlet
>>CrimsonObject -> ScarletObject
```

The maps are expressed in a hierarchical format of three levels in order to show if the mapping is meant for an interface, a method, or an argument type. However, by omitting a level we can express a mapping that spans all interface or all methods within a specific interface. For instance, to map the CrimsonObject type to ScarletObject in all methods of the scarlet and crimson interfaces we write:

```
%map crimson -> scarlet
>>CrimsonObject -> ScarletObject
```

It is crucial to note that the map command only specifies that a relationship exists between the two identifiers. If actual data conversion is required, it is done within the code generation step.

An identifier in a map that represents an argument can be expressed in several ways allowing the maximum ability to adapt to differently constructed interfaces. Arguments within a method can be expressed by type, name, or by an integer number that follows the special character #. The integer represents the ordered number of the argument as it appears in the method definition. It is useful to differentiate arguments in cases where the argument names are omitted and there are several arguments with the same type.

To enable the expression of an argument map that creates a connection with an argument belonging to a method not expressed at the higher level of the map, we use a qualifier expressing the outside method's name. This type of map enables us to express argument maps between methods that are not themselves mapped. In practice this usually results in generating code that saves an argument into a global variable and then uses this variable in another method. The following is an example of different map expressions:

```
%map crimson -> scarlet
>paintcrimson -> paintscarlet
>>#1 -> SomeOtherMethod::SomeOtherObject
```

The identifier *return* within a map represents the return type of a method. It is used to express scenarios where the return value is copied into an argument or an argument returned as the return value of a function.

We also allow a map that connects more than one entity on each side, though we impose some rules to ensure that the maps expressed are useful. That is, we allow one *in* port or method to map to any number of *out* methods enabling a more complicated relationship to be expressed. An any to any map is disallowed because it is unable to bring forward useful information for code generation.

Since SCIM will assume that all *inout* interfaces map to themselves, it is useful to provide a directive to remove one or all of these maps. The *%omitinout* command will remove the automatic mapping for a specific interface. Issuing this command without specifying an interface name will remove all *inout* interfaces that have been selected up to that point in the interface file. We also allow the reversal of this process for explicit interfaces. Interfaces can be reselected (remapped) through the *%remapinout* command. The reason for remapping interfaces is that since *inout* interfaces are mapped by default, we are able to omit all of the interfaces and then remap one in order to select that specific interface for code generation. The interface to interface maps that we have devised by the interface mapping directives and data parsed from the interface become special variables and expressions we can leverage inside of our SCIM template.

We also provide support for components whose interface is not specified in a SCIM accepted format through the interface mapping directives. This support consists of the ability to define new interfaces and methods to SCIM. For instance, if a *red* interface exists within a component model and its definition was unable to be read by SCIM we can define:

```
%map crimson -> NEW:red
> paintcrimson -> NEW:paintred
>> CrimsonObject -> NEW:RedObject
```

Defining SCIM-unreadable interfaces by hand allows us to proceed with code generation and create a bridge. We intend this approach to be used to surpass minor incompatibilities. If the component model of the *red* interface is to be used frequently a new plug-in should be implemented within SCIM to enable the parsing of those interfaces.

SCIM offers several commands to specify handling of data types. The purpose of these commands is to augment rules that are specified for a particular data model. By these data commands specified with the interface mapping directives, we add special handling of data at the granularity of an interface or method, not only for a specific component model.

Many of the data mapping commands we use are similar to ones used in object-oriented interoperability. We will discuss these further in Section 4.2, where we describe data models used to handle the data translation problem.

4.2 Data Models

In order to make component interoperability attainable, we need to solve the problem of migrating data from one component instance to another, keeping in mind that these instances belong to completely different component models and environments. The work on object-oriented interoperability by Konstantas [28] defines the following categories of data relationships: *equivalent*, *translated* and *type matched*. Two data types belonging to a pair of component instances are *equivalent* when a data type with same structure exists in both environments. This relationship is most common for basic types (e.g., int, char, float, etc.). *Type matched* types do not match completely but contain the same data parts in a different representation; for example the passing of similar data objects between two component instances. These types can be mapped to each other by using a proxy. Finally, *translated* data types contain a large mismatch in the data that is only remedied by a user-defined conversion function. This will happen if the data uses different measurement or temporal bases. The data categorization we describe transcends from object-oriented into component interoperability. We address each of these data relationship categories in SCIM's data models.

When examining interface data from component models, we have to account for cases where the type in the interface corresponds to a different type in the stubs, skeletons and implementation. A type named `Number` within the IDL definition may correspond to a regular C/C++ `int` within the actual component instance. This is a very common scenario in component software. Since SCIM parses interfaces to acquire data for the bridge generation, we rely on SCIM's data models to be able to accomodate this translation.

A data model in SCIM is a series of data definitions belonging to one specific component model. Each data model is represented by a object instance and inherits from an `EmitModel` base class (see Figure 4.1). A new instance is created per every SCIM compile. In most cases this involves instantiating two data models at a time. The `EmitModel` class contains methods and variables tailored for the problem of data translation and by inheriting from it, all component model-specific data models receive much of their functionality. In addition, the `EmitModel` class is used as the default case when a inherited data model does not exist or is not specified. To simplify our design of the `EmitModel` class we currently limit the data definitions to C/C++. Extending the data models to support any programming language is trivial if certain features are removed, such as the ability to perform automatic pointer manipulation of data types. We have not yet devised a way to make these features that are available only in C/C++ available to any programming language.

The `EmitModel` class defines a structure to handle several aspects of component data translation. To cope with the discrepancy between interface data definition and implementational data definition (discussed above) the `EmitModel` class provides the `idlToEmitType` method. This method is intended to be implemented by a derived data model capable of describing the correspondence between its IDL and implementational types. For every IDL type supplied as an argument to the `idlToEmitType` method, the return value should be the corresponding implementational type. This method is used pervasively throughout the data model to provide this important utility and as such it ought to be implemented by most data models. Within the interface mapping directives we provide an additional way to map between IDL and implementational types. To do that we use a special operator '+>'. This operator can be used anywhere within an interface mapping directive specification and its definition will be automatically added into the appropriate data model.

To define a map between a type in one component model and a different type in another we use `EmitClass's addMappedType` function. A data model collects these maps and uses them during the code generation. Since a data model class instance persists only during one SCIM run, we do not have to worry about purging our collection of mapped types. Although available to invoked by anyone, the `addMappedType` function is most often referenced by the *%datamap* interface mapping directive. This directive is a *%map* command aimed at data. We use it in two-level hierarchical sense to define class and class variable maps. Using two objects `CrimsonObject` and `ScarletObject` the *%datamap* directive may look like this:

```
%datamap ScarletObject -> CrimsonObject
> scarletObjValue -> crimsonObjValue
```

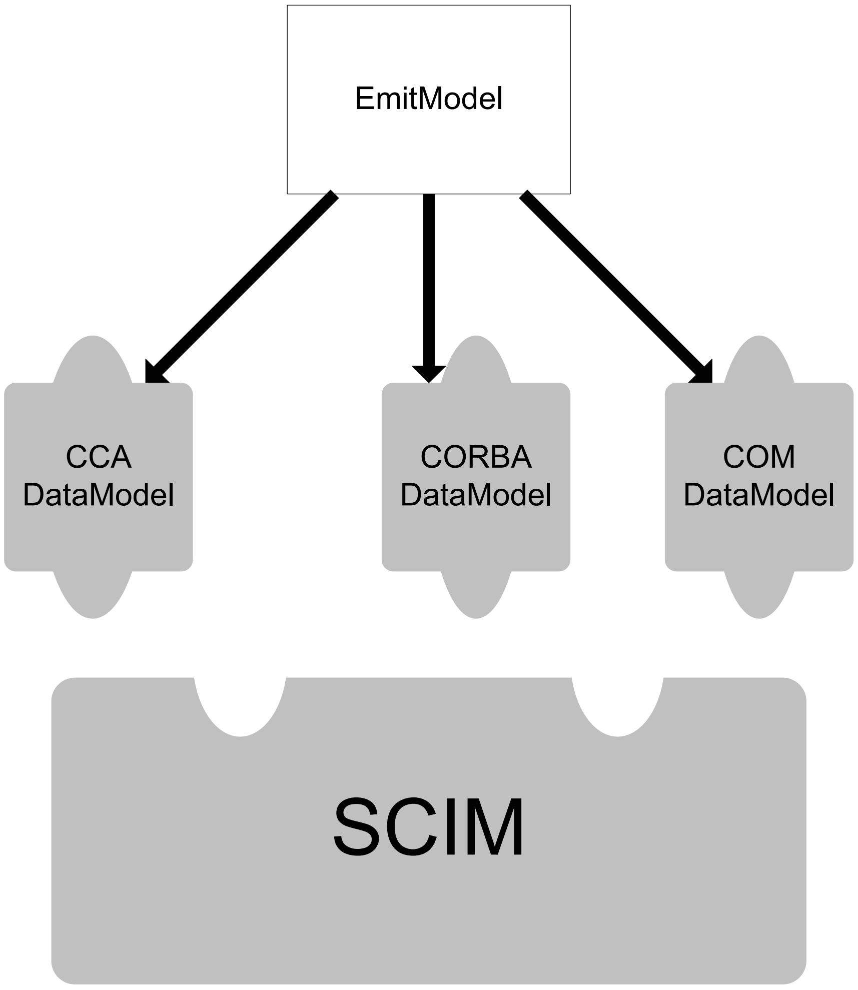


Figure 4.1. Data models inherit from the `EmitModel` class and are used two at a time within SCIM.

Some matching types are harder to convert and are not easily expressible in this fashion. These *translated* types contain a non trivial map between each other. It can be that a data structure was expressed in inches, and a similar one with a different component requires centimeters. To perform this translation we need a specially tailored function to be provided. In order to register this function into the SCIM's data models, we use `EmitModel's registerConvertFunction` method. The method takes in two typenames and a function pointer, and on any occurrence of those two types in two corresponding interfaces it produces code performing the necessary translation.

The `EmitModel` base class is also equipped to handle C++ calling convention argument passing. The base class and derived data models automatically determine if, for instance, a method in one component expects a pointer to an object, while the calling component passes the object by value. The `EmitModel` class is able to emit code to account for this difference throughout the generated bridge.

All data models inherit from the base `EmitModel` class and therefore contain all the capabilities we described. Data model instances usually implement the `idlToEmitType` to provide known data type information. Interface mapping directives are leveraged to provide information about data maps or translation functions. Implementing a data model for a new component model is usually not a very consuming task. In rare cases the data model implements a functionality of its own that was not specified by the `EmitModel`.

After discussing the design of the data models, we reconsider the groups of data relationships first introduced by Konstantas [28] and how out design applies to each of them:

• **Equivalent** types are handled automatically by SCIM; these are simple and require no special treatment. A simple match determines equivalence of these argument and the bridge code passes them through.

• **Type Matched** types require interface mapping directives that enumerate the differences between the two data types. We use the *%datamap* command to specify this mapping. The command is extended to express mapping deeper to within an argument. The data model leverages this description to generate a proxy object. The generated proxy exists in the server environment and relays requests to instantiated client data. When passing objects through a bridge we often leverage the tools intended for *type matched* types.

• **Translated** types are handled in SCIM by using both the data models and interface mapping directives to provide a way to express and specify a function that translates between two data types. When both types are encountered, the function is used to generate translation code into the bridge. An example of this is translating between a 2D array and a unstructured mesh.

4.3 SCIM Templates

SCIM's back-end combines the interface mapping and data models with a output code specification template written in embedded Ruby (eRuby). eRuby is a templating 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. The main purpose of the SCIM template is to provide a very flexible mechanism to express the variety of possible bridge output. The template is very similar to the resulting code.

Ruby language expressions are evaluated and their result is placed into the template. The interface to interface maps that we have devised in the front end and data extracted from the interface will become special variables and expressions we can leverage inside of our template.

Many variables are available within the template. Below is a list of some of the variables that a template can use. Specific variables exist to retrieve the package/namespace names, interface names, method names, method types and method arguments. The values of these variables differ as different methods and interfaces are being emitted:

```
$inPackage   //name of ``in'' model's package
$outPackage   //name of ``out'' model's package
$inInterfaceName   //name of the current ``in'' interface
$outInterfaceName  //name of the current ``out'' interface
$inMethodName  //the current ``in'' method's name
$inMethodType  //the return type of the current ``in'' method
$outMethodName  //the current ``out'' method's nam
$outMethodType  //the return type of the current ``out'' method
$MethodArgs[]   //array of arguments belonging to a method
$outMethodArgs[]  //array of arguments belonging to a ``out''method
$outMethodType  //the return type of the current ``out'' method
$iDM      //''in'' data model
$oDM      //''out'' data model
```

References to data models exist as template variables in SCIM. We depend of strategically placed invocations of the data model within the template to enable the insertion of the right data translation code. To specify that some text is specific to an interface or a method, we use:

```
<interface> </interface>
<method> </method>
```

```
<outmethod> </outmethod>
```

These denote that the code contained within each of them is repeated upon each mapped interface/method. Some of the template variables are available only within the scope of the *interface*, *method*, or *outmethod* command. This allows the SCIM code generation to operate on collections of many interfaces.

To better visualize how a SCIM template is used we use a simple example. Below is an example template that would perform simple method redirection for C++ interfaces. This is a basic procedural bridge that receives an invocation from one method and redirects it to another method of a different name, passing all the same arguments:

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

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

```
void crimson::paintcrimson(ColorObject cobj) {
  obj = new scarlet();
  obj->paintscarlet(cObj);
}
```

It can be noticed that all the arguments and variables have been filled out with their values to produce a valid C/C++ piece of code. The arguments of a given method are stored in a separate *MethodArgs* array that can be accessed only within the <method> command. The array contains three string fields: mode, type and name. We can leverage the arguments array to create loops that will extract the arguments and generate lines of code that perform method definition or invocation.

To exemplify the potential usefulness of these templates that allow the embedding of code, we modify our example to one that tests (at code generation time) if we are using the template for the `scarlet` interface. When this template is used for some other interface SCIM generates a statement that prints an error message as the body of the method. This slightly more complex example may not be of any practical use; however, it shows the expressiveness of the templates:

```
...
<%
  if($inInterfaceName == 'scarlet')
    obj = new <%= $inInterfaceName%>();
    obj-><%= $outMethodName%>(<$= $iEM.outCallArgs%>);
  else
    printf(``This method is not used for the scarlet interface\n'');
  end
%>
...
```

4.4 Chapter Summary

SCIM is a tool that automates the creation of bridge code between component instances, once it is provided with an expressive template describing the solution to the interoperation problem between the component models. While augmented by the semantic tools described in Chapter 5, SCIM can be very useful when used in isolation. The code generator contains interface mapping directives, data models, and templates that allow embedded code that enable the code generator to become very useful in many interoperability tasks. In Chapter 6 we apply SCIM in isolation to combine two software libraries and found it amenable to task not involving component models. Although the example shows that SCIM can be a very useful tool in a variety of tasks even when used alone, the semantic tools make SCIM easier and faster to use for component model interoperability.

CHAPTER 5

SEMANTIC TOOLS DESIGN

Two or more component-meta models and a bridging language description of their cooperation can be used to automatically generate a bridge in a semantic fashion. A meta-model provides a general specification that is intended to model both the structure and behavior of a component model (such as CORBA, CCA, COM, etc.) Each meta-model is based on an abstract and a realized model. The interplay of these two base models form the meta-model specification. The bridging language expresses the behavior of a bridge between two (or more) component models based on the description of two corresponding component meta-models. The language's purpose is to formulate the connection between meta-models (and corresponding component models); often the connection is nontrivial and requires a sophisticated expression mechanism. Much of the bridge language's functionality is expressed through the meta-models themselves.

We use this modeling approach that we developed to generatively capture the semantics of component models, and to produce a SCIM template based on this model. Generating a SCIM template significantly improves the ease of creating an initial bridge between a set of component models. The modeling approach exhibits further benefits in productivity that accrue as more component models and expressed by meta-models. Each addition of new component meta-models increases the ability to easily combine component models as new interoperability needs arise. We begin to explain our method by describing the meta-model's details and continue to describe how we leverage them to formulate a bridge through the bridging language.

5.1 Component Meta-Model

An abstract and a realized model combine to form the component meta-model. The abstract model describes the structure of the meta-model from a component software viewpoint and uses the realized model to create a programming language representation. Together they form a clear mapping of a component model into a programming language enabling SCIM template or direct bridge generation. Both of the submodels of the component meta-model are implemented as a set of base classes. To model a specific component architecture, the user needs to extend the abstract model classes and use the realized model classes.

The meta-model we present can be produced quickly and is amenable for code generation. In defining its structure we make a few limited assumptions about the component model that is being modeled. We define our meta-model to use component model abstractions such as ports, methods and connections. In our system, these are used as labels that contain a hierarchical relationship between each other. All of the component models that we are aware of conform to this basic architectural structure. While our approach works best for component models, it is possible to create models of entities that may not contain ports or connections. In this section we provide an example of modeling UNIX pipes and signals, which are not based on our assumed hierarchy. We used the Ruby programming language to create the current version of our abstract and realized model implementation.

5.1.1 Abstract Model Design

The abstract model consists of the following Ruby classes: `Abstract Framework`, `Abstract Component`, `Abstract Port` and `Abstract Data Model` (see Figure 5.1). Some of the methods that are in the abstract model's classes are directly invoked by the bridging language. In fact, much of the bridging language functionality is implemented through these classes. We divide the methods of the abstract model classes in three categories: acquirers, code emitters and events. The acquirer methods handle pointers to the classes; allowing the user to get a pointer to a framework's components, a component's ports, etc. The code emitter methods are invoked to produce code. They contain the links to the realized model classes and therefore are the ones that are provide the information to create a bridge. The acquirer and code emitter methods are implemented by the user when inheriting from the abstract classes and forming the description of a component meta-model. The event methods come free to the user and are an integral part of the bridging language's function. In this section we will discuss the acquirer and code emitting methods, leaving the discussion of the event methods for the bridging language chapter.

The `Abstract Framework` class is the highest level class in the abstract model hierarchy. It represents a component framework as an environment where components can

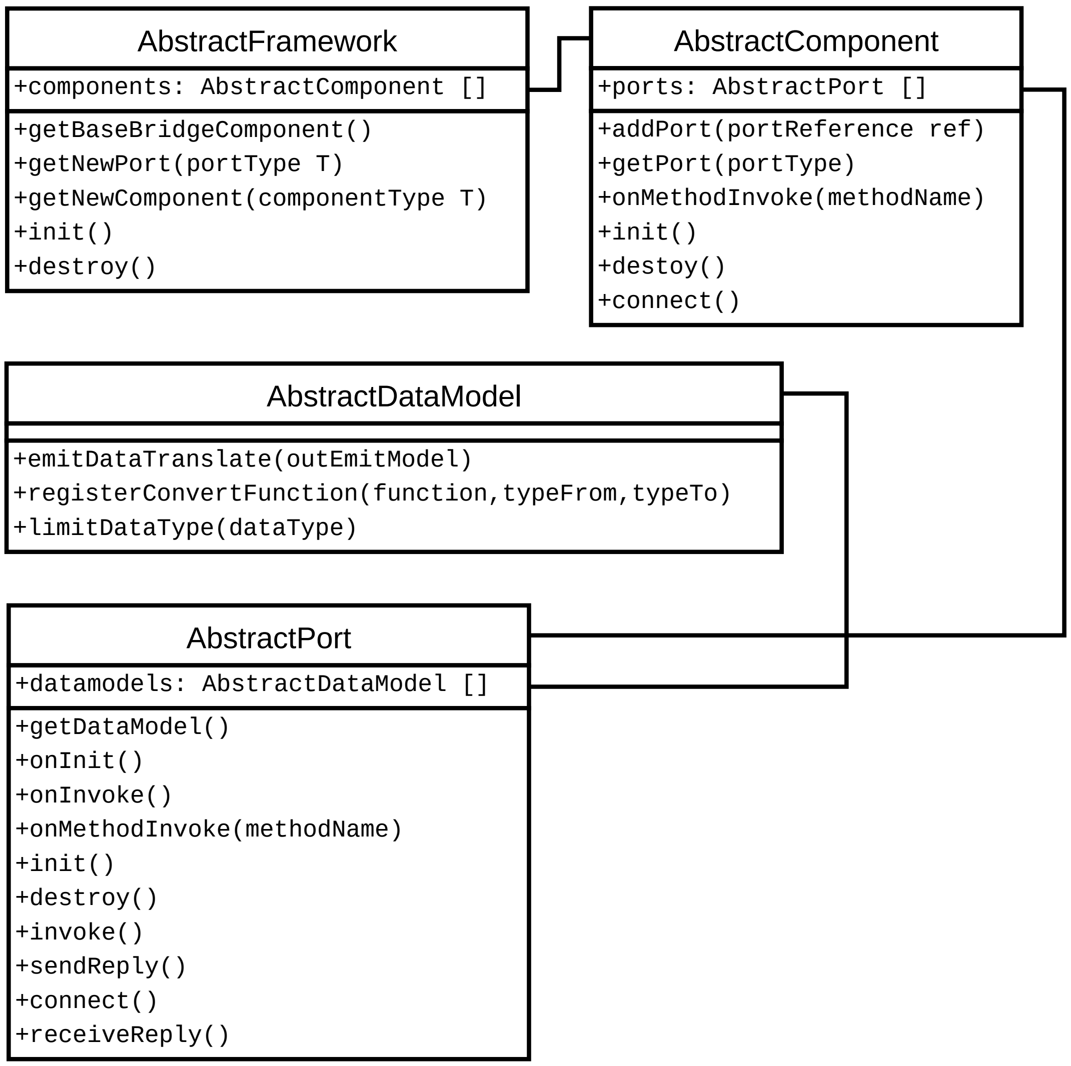


Figure 5.1. Structure of the abstract model classes. The abstract model is used as a model of component model architecture.

exist. It contains acquirer methods that get a handle to a base bridge component, a component, and a port: `getBaseBridgeComponent`, `getNewPort`, and `getNewComponent`. A base bridge component is a component[1] that represents a generic bridge container in that specific framework. This construct may in practice be a component, or a class, or a procedure. The user has the burden and flexibility to specify the structure of a bridge through a base bridge component so that it can exist and be used in a given framework. The `Abstract Framework` contains two code emitters: `init` and `destroy` that the user

[1]The term component is used very loosely here

implements based on the behavior of this framework when initialized.

Within the `Abstract Component` class of the abstract model we place capabilities to model any component type. By component type we mean a paradigm of components within the component model. For instance, we differentiate components based on model of computation (method invocation, dataflow, parallel, or distributed). However, components that differ only in interfaces or number of ports belong to the same `Abstract Component` derived class. The component class offers aquirer method to add or retrieve an existing port of the component, or to get handles to the framework to which the component belongs to. The `Abstract Component` class contains three code emitters: `init`, `destroy` and `connect`. The first two are doubled in the `Abstract Framework` class. The `connect` method, when implemented by a user in a class that extends `Abstract Component`, describes the behavior of a component to connect to another compatible component.

The `Abstract Port` base class is based on a similar principle as all of the other abstract classes we have described: it describes a unit's type as a specific paradigm. Also, similar to the other abstract model classes, it supports a variety of acquirer methods to obtain handles to other classes within the hierarchy. The `Abstract Port` class also contains a large number of predefined code emitting methods: `init`, `destroy`, `invoke`, `sendReply`, `connect`, `receiveReply`. Not all of these are implemented by the user in each scenario, however, they describe a variety of behaviors a port can exhibit. The `sendReply`, `receiveReply` and `invoke` methods are intended for the frequent case that a port instance contains methods. When a port instance's methods are invoked, all of these scenarios may occur. The `onMethodInvoke( methodName )` event exists in each of the `Abstract Port` and `Abstract Component` classes to enable the creation of a new event. In this fashion we maintain some predefined events and provide flexibility to define new ones if needed.

`Abstract Data Model` provides a way to include a group of data handling commands into the generated bridge. This class provides a way to specify some of the data conversion early on. Thus far, the `Abstract Data Model` class provides three commands that we can use from within the bridge language: `emitDataTranslate`, `registerConvertFunction` and `limitDataType`. The `emitDataTranslate` method's purpose is to mark the spot where the data translation needs to occur for a specific type. Using this command we can specify that we need to translate a data type, for instance, after initializing the data and before invoking the target method. In order to register a data conversion function for translated types within the bridge language we use the `registerConvertFunction` method. This method carries the same name as a method with the same purpose within SCIM's data models. The `limitDataType` method is used to limit a particular bridge to a certain type; useful in situations where we are only able to translate one type and all others produce an error.

5.1.2 Realized Model Design

The realized model is aimed at modeling a programming language implementation down to a line of code. It consists of the following structures: `MModel`, `MComponent`, `MPort`, `MClass`, MMethod, `MParameter`, and `MCodeLine` (see Figure 5.2). These classes were the minimum we were able to select in order to describe an implementation. We were not concerned with providing an all-encompassing model, just one that serves the purpose of describing the abstract model into programming language terms. As a true test of its expressibility, the realized model is self expressible: it is able to model its own class structure.

To use the realized model structures, classes derived from the abstract model instantiate the realized model classes directly. For instance, to define that a port P belonging a component model M is programmatically represented by an empty class C_1 the user does the following:

• Defines a class P_1 that extends `Abstract Port` of the abstract model

• Within P_1's constructor OR within P_1's `init` method the user instantiates a `MClass` class with the name parameter set to C_1.

The code emitter methods in the abstract model is usually where the connection between the abstract and realized model is implemented by the user. In our experience, the lines of code that are written to create this connection of the models are not numerous.

5.2 Bridging Language

We designed a bridging language to express a bridge among any set of component models. Meta-models represent the structure and implementation of component models. The bridging language manipulates these meta-models to express a bridge. The process of bridge expression in the language consists of expressing the structure of the bridge (e.g., classes and methods) and the behavior of the bridge (e.g., lines of code in the right places). The language expresses both the structure and the behavior of the resulting bridge, relying

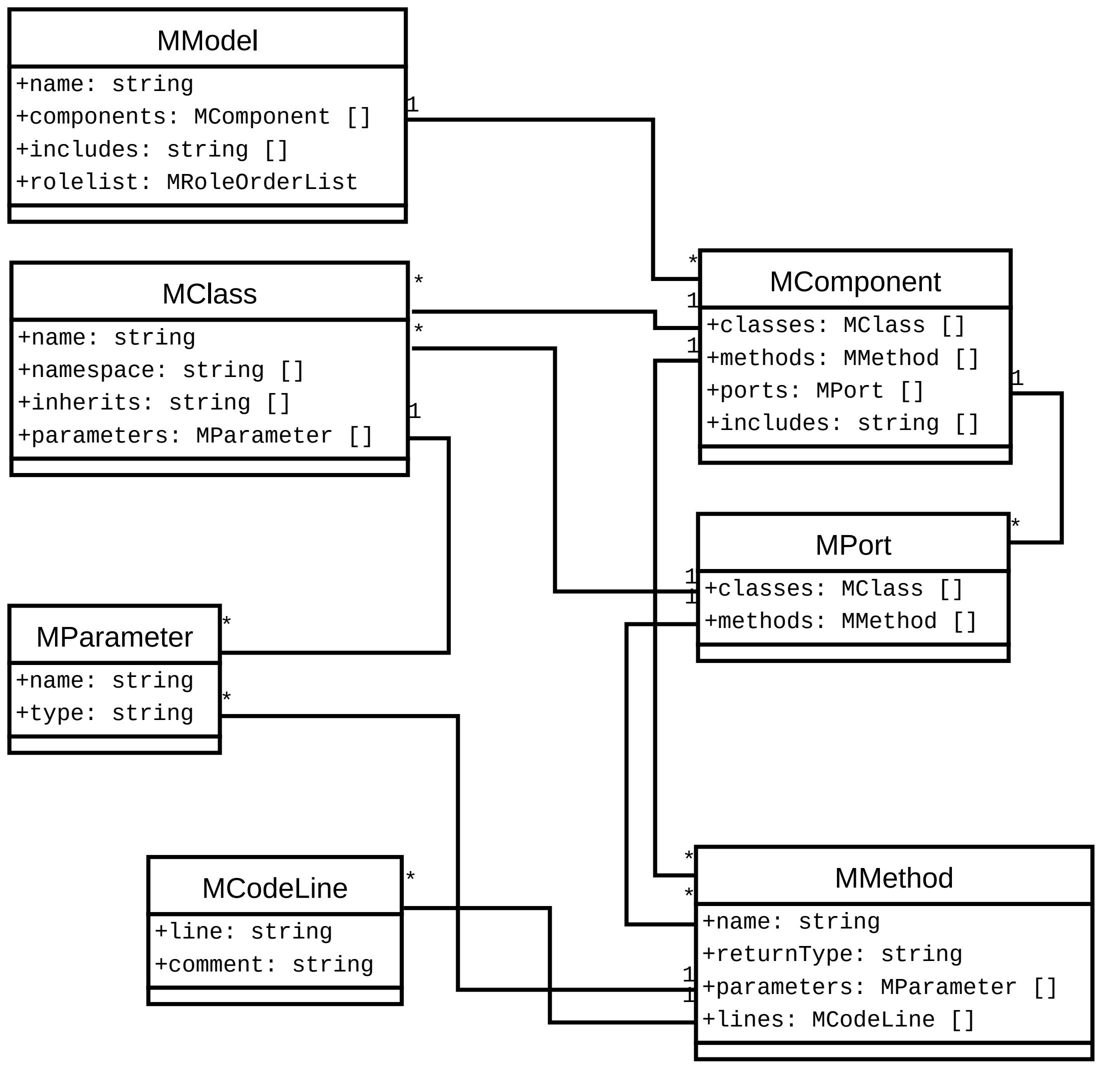


Figure 5.2. Structure of the realized model classes. The realized model is used as a programming language expression of a component model.

heavily on the meta-models for both pieces. Every code written in our language usually begins by selecting the framework in which the resulting bridge will exist. This can be the same framework as one of the meta-models or one that is completely unrelated. The bridge code queries the meta-model of that framework for a base bridge component. We use the base bridge as our starting structure to which we add ports and other structures. We acquire ports from the meta-models involved and tack them on to the base bridge component. For instance, we add a server port for component model X and a client port for component model Y to the base bridge if we intend to bridge X's client port to Y's server port. In this fashion the resulting bridge will be a perfect match to mediate a X component to Y component communication.

After the structure building phase of the bridge is finished, we use event methods to describe the bridge's behavior. The event methods are provided by the abstract meta model and they define common component, port, or framework events. The bridging language is used to specify that upon a given event, certain code emiting methods of the abstract meta-model are to be invoked. These code emiting methods of the abstract models relate the abstract model to the realized model. The purpose of this relationship is to provide a concrete programming language representation of certain component features. This task of relating events to code emiter methods and doing this in an imperative way that preserves the order of operations is the main purpose of the bridging language.

To increase the ability to express different bridges, we find it useful to include the notion of state variables. Through variables we intend to provide a method to clearly express the parameters of a bridge. Tweaking these parameters can produce a different bridge. An example of this usually takes the form of a meta-models definition that uses a parameter we can set within the bridging language. Through this approach we can produce more general meta-models and postpone certain detail-level decisions to the bridge language code. We provide the ability to define and use variables through a pair of accessor and mutator methods. The methods are: `getVariable` and `setVariable` and they take a variable name that can be assigned a value or accessed anywhere within the meta-models or bridge language code.

The bridging language also provides a locking synchronization mechanism through two methods: `waitLock` and `releaseLock`. The methods provide simple conditional variable support for multithreaded bridges that usually span several component models. The result of using the synchronization abstraction we have provided within the bridge is to produce subroutines that provide this functionality within the generated code. Currently, we provide only a C/C++ reference implementation, but allow abstract meta-model implementors to extend these methods to provide support for other languages.

Another feature of the bridging language is support for including data model commands. These commands have the purpose of manipulating the data model within the bridging language. This is also useful for placing data commands in their proper location of the bridge. Using this method we get a more general perspective on the relationship of the data translation to the other functions that the bridge performs. The actual code that gets generated as a result of these commands depends on the implementation of the abstract data model. We provide a default implementation that manipulated SCIM's data models. A user is encouraged to provide an implementation of these commands applicable for another generative domain. We provide three commands that are defined within an `Abstract Data Model` class of the abstract meta-model: `emitDataTranslate`, `registerConvertFunction` and `limitDataType`. The `emitDataTranslate` method's designates the location where the data translation code is placed. To handle translated types by registering a data translation function we use the `registerConvertFunction` method. This method carries the same name as a method with the same purpose within SCIM's data models. The `limitDataType` method is used to limit a particular bridge to a certain type and is useful in scenarios where the bridge is able to accept only a specific type.

5.2.1 Execution of Bridge Language

We designed a set of Ruby scripts to perform the task of generating code from the bridge language into C++. These scripts execute the language, visiting the component meta-models to retreive much of the generated code. They also provide some intellegence in organizing this code within the proper methods and class definitions. However, they do not check the correctness of the code they produce. Incorrect code is possible not only due to bad modeling or bad bridging language code, but also because their combination may not be operational. In our exprerience, for generally written component meta-models this does not occur very often. A different set of scripts exist that adapt the code to be used as a SCIM template.

5.3 Semantic Bridge Generation Example

To illustrate our design, we will show a very simple example. The example is of converting UNIX signals into a UNIX FIFO and it is written using the Ruby programming language and produces C code. Our concept implementation of the bridging language and component meta-model is in Ruby. The example is not particularly applicable but it is simple enough to be contained in these pages, and shows how disparate functions can be bridged.

The bridge we are designing has to receive UNIX signals, convert them into strings and write these strings onto a UNIX FIFO pipe. We start by defining specific types of ports that may belong to a UNIX component framework [2] in order to obtain a meta-model. This framework should contain ports to send and receive signals and ports that read or write to FIFOs. Note that these are not specific port instances, but rather types of ports that are supported by their own communication primitive. Here is a definition of a `RecvSignal` port that acts as a UNIX signal receiver:

```
class RecvSignal < AbstractPort
 //class constructor
 def initialize
   @methods = MMethod.new("void","urg_handler_func")
 end

 //from AbstractPort
 def init
   $stream += 'signal(SIGURG,urg_handler_func);\n'
 end
end
```

The abstract classes (e.g., AbstractPort) define a set of situation functions that a port can choose to implement. These functions are: `init, destroy, invoke, sendReply, connect, receiveReply`. Their purpose is to define an action that this port performs in a given situation. For instance, the `RecvSignal` port, when initializing through the `init` method, writes out a line of code that represents invoking the `signal(...)` system call. The `$stream` variable used in this method represents its direct output. The `initialize` method is a Ruby defined constructor for a class. The similarity between the `init` and `initialize` methods is accidental as one of them is a constructor and the other one is a defined port situation method. Any UNIX signal receiver needs to supply a function that will accept and process the signal. The realized model provides a signal handler function. Our signal handler is not general; it handles only SIGURG signals. To make ports more general, the bridge language allows defining variables and setting/getting of variable values. This enables the description of a port that requires external information. Above we described a basic signal handler port, but the bridge we intend to design for

[2] We use the terms component, framework, and port loosely in this example

our example also requires a port that enables writing to FIFOs:

```
class WriteFifo < AbstractPort
 //from AbstractPort
 def init
   $stream += 'mkfifo(aFIFO,"w");\n'
 end

 //from AbstractPort
 def invoke
   ttw = getVar("textToWrite")
   $stream += 'write(aFIFO," + ttw + ");\n'
 end
end
```

The `WriteFifo` port's `init` method sets up a FIFO for writing through the `mkfifo(...)` system call. In the `invoke` method a variable is read that describes the text to write into the FIFO. The text is then written using the `write(...)` system call. We have finished describing the structure of the ports and now we focus on implementing the logic using the following bridging language code:

```
require 'inc-bdsl.rb'

fwk = selectFramework("unix-basic")
$bc = fwk.getBaseBridgeComponent()

recv_sig = fwk.getNewPort("RecvSignal")
wr_fifo = fwk.getNewPort("WriteFifo")

$bc.addPort(recv_sig)
$bc.addPort(wr_fifo)

$bc.onInit {
  recv_sig.init
14    wr_fifo.init
15  }
16
17  recv_sig.onInvoke {
18    setVar("textToWrite","SIGURG")
19    wr_fifo.invoke
20  }
```

The bridging language uses the component meta-models to express a bridge between two component instances or models. Line 1 of this snippet of code is includes the standard component meta-model libraries. In line 3 a reference to the "unix-basic" framework is acquired. The "unix-basic" framework is a container for the UNIX signal and FIFO ports. Line 4 gets a reference to a base bridge component, which represents a foundation structure upon which we can build a bridge. In our example, this base bridge component defines a main function and an additional class "BridgeClass" used to clarify the code. In a more challenging scenario where we want to connect components in CCA and CORBA, for instance, we may need the bridge code itself to be contained within a CCA component. To this end, we can express the basics of a CCA component and use that as our base bridge component. The base bridge component expresses static features meant to make the code fit in the intended environment. In lines 6 and 7 we instantiate and gain references to the ports we implemented, and we add these to our base bridge component in lines 9 and 10. Lines 12 through 15 express the `onInit` method that is called upon initialization of the base bridge component. Since the sample base bridge component is unsophisticated, the initialization of each of the ports (lines 13 and 14) will be the only code in this part of the output. In lines 17 through 20 we implement the onInvoke method of the `RecvSignal` port. This places the result of lines 18 and 19 within the implementation of each method that the `RecvSignal` contains. The only method defined in this case is the `urg_handler_function`. After processing this code and the defined ports the following bridge is produced:

```
int main() {
  BridgeClass* bc = new BridgeClass();
}
```

```
BridgeClass::BridgeClass() {
  //RecvSignal.init
  signal(SIGURG,urg_handler_func);
  //WriteFifo.init
  mkfifo(aFIFO,"w");
}

void urg_handler_func() {
  //WriteFifo.invoke
  write(aFIFO,"SIGURG");
}
```

The resulting code registers a signal handler function called "urg_handler_function" to handle the SIGURG signal and opens a UNIX FIFO pipe. The handler function writes the string "SIGURG" to the pipe when it is invoked. A listener to this pipe, perhaps in another process, should be able to get information that the SIGURG signal was handled.

The bridging language and component meta-modeling contain more methods and capabilities than exhibited in this example. They are meant to support more advanced bridge expression, such as when dealing with different models of control. Some of these methods make expressing the implementation more straightforward, some are meant to help data translation, and yet others are meant as synchronization primitives. To tie this tool in with SCIM, we define a set of SCIM's template expressions and use those in the generation of the output. This process produces a SCIM template instead of a bridge instance that we showed above.

5.4 Chapter Summary

We present a technique to model component models that enables fast model creation and code generation, while being applicable to many component models. The technique is based on bridging the gap between a component model and a programming language, expressing how component model structure corresponds to a (in most cases, object-oriented) programming language. We propose two modeling entities that perform this task, a realized and an abstract model. The interaction of these two entities explains the interplay of component model and programming language code.

A bridging language is used to express interaction details of two given component meta-models. This language expresses the task of combining a set component models through a bridge by expressing the similar task of combining a corresponding set of component meta-models. The bridging language can be used to combine any number of component models. It uses a event-based language style enabling quick and efficient expression of interoperation details.

CHAPTER 6

BRIDGE RESULTS

To demonstrate our design, we implemented a proof-of-concept example of bridge generation between three component models: CCA, VTK and CORBA. The example is of a realistic, yet straightforward, implementation of the finite element method that we use to solve the heat distribution through an L-shaped domain. We use component instances from all three component models and leverage the VTK component model to display a 3D image. The target framework is SCIRun2, which is our in-house component framework developed to support management of instances of several component models. We describe SCIRun2 and its meta-component interface ahead of describing the example and its results.

6.1 SCIRun2

Our code generation tool was inspired by SCIRun2's meta-component interface. This is an approach to including many separate component models into a component framework. SCIRun2 is also the only domain thus far where SCIM and the bridging language have been evaluated and used although it has the potential to be used more widely and in many different domains.

SCIRun2 is a component framework whose goal is to create an environment that facilitates the coupling of software components from a variety of domains in a single high-performance application. For example, parallel components based on CCA could be connected to serial or distributed components based on CORBA or other models. In this manner, the scientist is able to employ the 'right tool for the right job' in a flexible software framework

Component models can differ widely. For example, they can use a different network communication scheme, rely on a different communication paradigm (method invocation, publish/subscribe), be synchronous or asynchronous etc. The differences are inherent as the models were constructed to satisfy certain usability or performance requirements.

Although component model differences are large, it is possible to extrapolate a few basic similarities between typical models. For instance, each component model has a number of components that in turn contain one or more ports. These ports either push or pull some data and therefore are either an in-port or an out-port. Based on assumptions of this kind, we can design an abstract meta-component interface. The goal of this meta-component interface is straightforward: provide an interface through which component models can be plugged into and manipulated within the SCIRun2 framework.

Each component model supported in SCIRun2 implements interfaces based on an abstract meta-component interface specification. The SCIRun2 framework manipulates concrete implementations of this abstract meta-component interface and through it is able to handle component instances belonging to an unlimited number of component models.

The interfaces defined by the meta-component interface primarily abstract the process of component creation, deletion and assembly. A component model is required to implement the `ComponentModel`, `ComponentInstance`, and `PortInstance` interfaces. To better illustrate the concept of the meta-component interface, below we present part of it in Figure 6.1.

In order for a concrete component model to exist within the SCIRun2 framework it needs to implement the above meta-component interfaces. This should be done in a way that preserves the functionality and requirements of the model itself.

The Visualization Toolkit (VTK) [38] uses a pipeline architecture as a means to visualize multiple kinds of data. It consists of a series of C++ objects that interoperate through a demand-driven dataflow model. VTK is a commonly used tool in many projects that require visualization. The implementation of VTK within the SCIRun2 framework consists of two aspects: one is a concrete implementation of the meta-component interface for VTK (from here the VTK component model) and the other is a VTK wrapper used to encapsulate each VTK object. These are intertwined in the same set of classes, but are aimed at supplying different functionality. The meta-component interface's VTK implementation provides a handle for the SCIRun2 framework to manipulate VTK components. On the other hand, the VTK object wrapper provides the ability to interface with VTK objects. The VTK object wrapper provides a layer on top of a VTK object whose purpose is to perform on the object certain tasks required by the framework: instantiation and connection of composable objects. Currently, the user needs to implement this relatively simple VTK object wrapper consisting of very few lines of code. We plan to automate

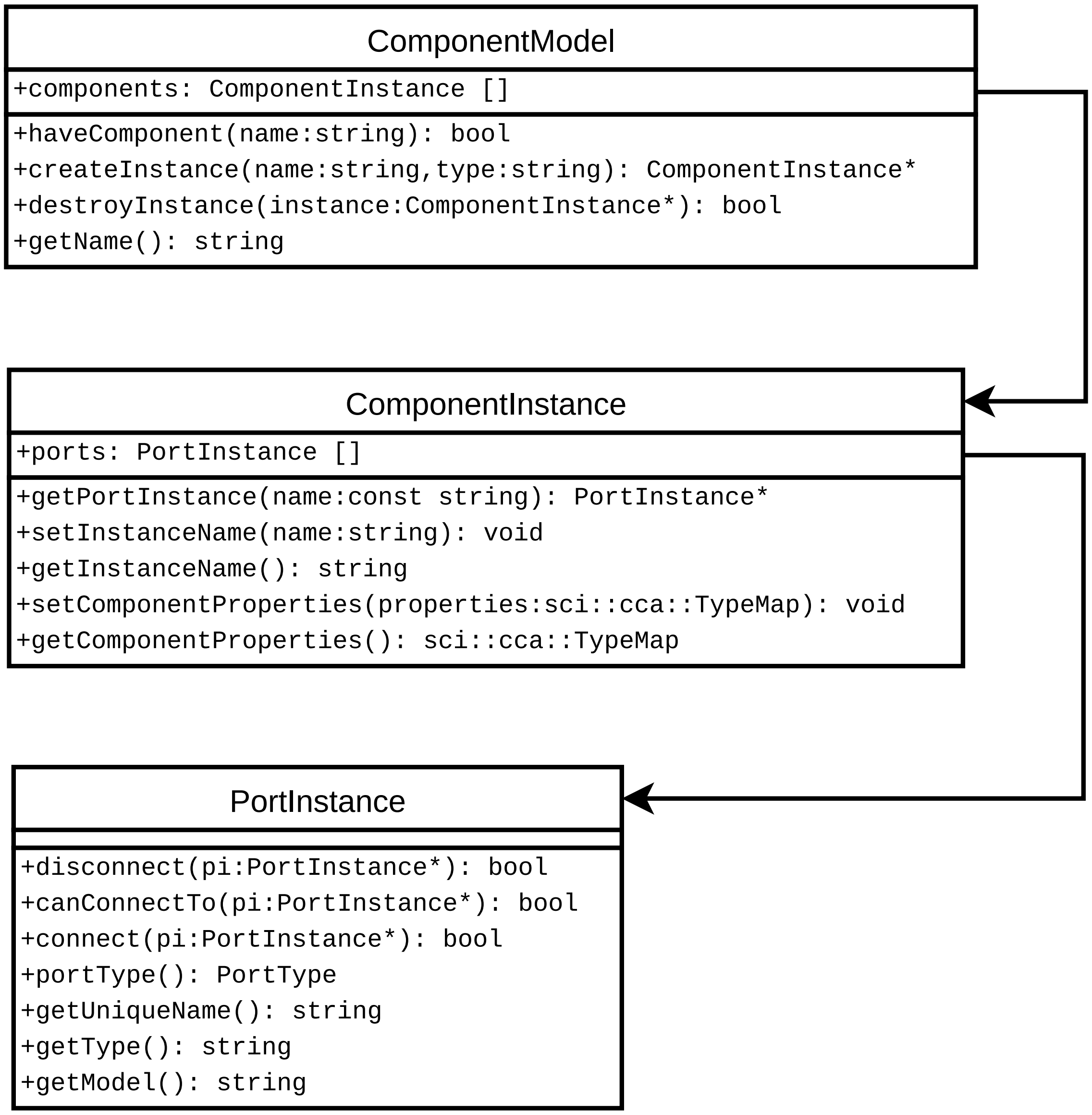

ComponentModel
+components: ComponentInstance []
+haveComponent(name:string): bool
+createInstance(name:string,type:string): ComponentInstance*
+destroyInstance(instance:ComponentInstance*): bool
+getName(): string
ComponentInstance
+ports: PortInstance []
+getPortInstance(name:const string): PortInstance*
+setInstanceName(name:string): void
+getInstanceName(): string
+setComponentProperties(properties:sci::cca::TypeMap): void
+getComponentProperties(): sci::cca::TypeMap
PortInstance
+disconnect(pi:PortInstance*): bool
+canConnectTo(pi:PortInstance*): bool
+connect(pi:PortInstance*): bool
+portType(): PortType
+getUniqueName(): string
+getType(): string
+getModel(): string

Figure 6.1. A UML diagram of SCIRun2's meta-component model interface that enables multiple component models to be included in the framework.

the wrapper creation task in the future because it is simple and repetitive. SCIM can be used for this task and it would be able to generate all wrappers based on a single template specification. The design of the VTK component model is shown in Figure 6.2.

SCIRun's VTK component model does not interfere with the execution of the VTK pipeline; however, it is concerned with initializing and assembling this pipeline in an accurate way. In order to perform this task, we designate two methods in the VTK object wrapper: `accept` and `connect`. We require these methods to be implemented by

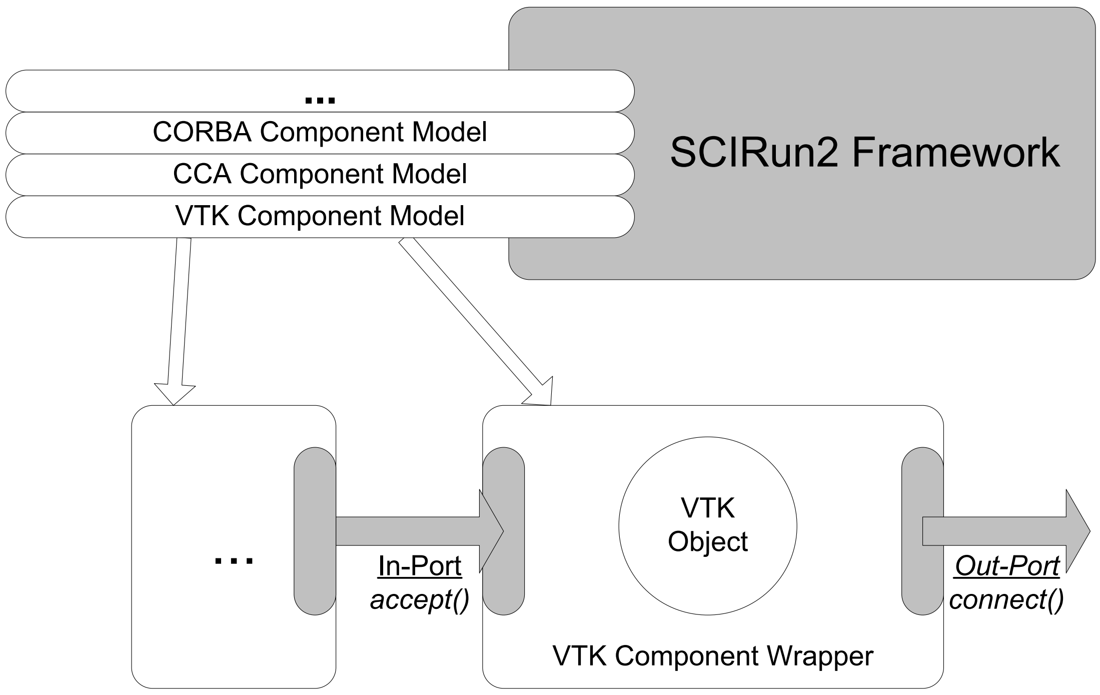


Figure 6.2. SCIRun2's meta-component interface abstraction of VTK.

each VTK object wrapper such that `accept` ensures the two VTK objects are compatible and `connect` performs the actual connection. Upon the user's attempt to connect two ports that belong to VTK objects using the SCIRun2 GUI, the framework invokes the `canConnectTo()` method on the instance of the VTK component model. The abstract definition of this method is given above in the `PortInstance` class. In turn, this invokes the `accept` method on all the instantiated VTK wrapper components in order to build a list of compatible connections. Once one of the compatible connections is chosen by the user, the `connect` method is invoked on the VTK component model and propagated to the method of the same name in the VTK wrapper. This exemplifies how the VTK component model and VTK wrapper components operate within SCIRun2. Many other models operate similarly; however most of them already behave as components and do not require a specific wrapper.

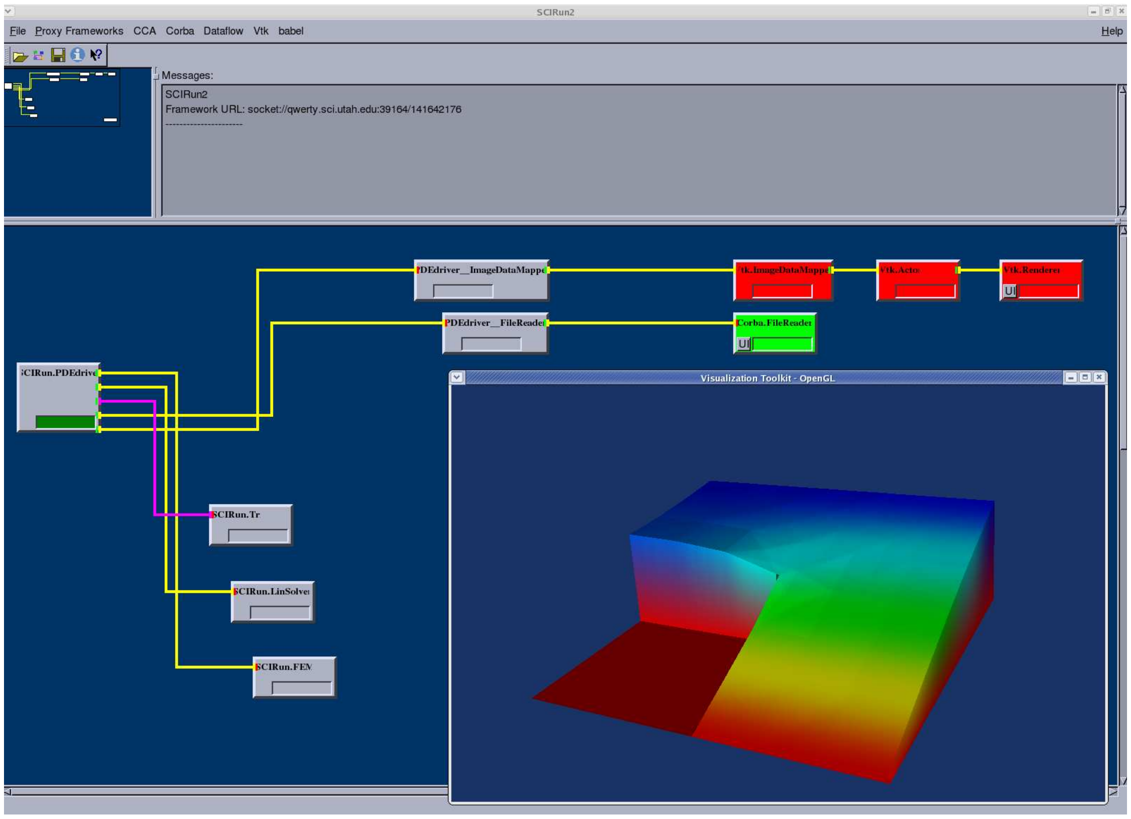
Though multiple models may coexist in SCIRun2 through an abstraction like the meta-component interface, an even greater problem still lingers: allowing two components belonging to different component models to communicate in order to form one application. The VTK components within SCIRun2 behave very differently and use completely different data types than those belonging to any other component model. The goal of automatic bridge generation is to provide a semiautomatic mechanism to alleviate this problem. The value of these methods within SCIRun2 is the ability to compose and run simulations consisting of disparate components. This enables the creation of a simulation that may use, for example, EJB components to get a dataset from a remote database, use CCA components to perform a computation on the data, and leverage 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. The method that SCIRun2 uses provides efficiency via economies of scale in the translation process, enables the user to manipulate all the component instances from one framework, and provides a general and extensible way to add component models as needed.

6.2 Heat Distribution Problem

We present an example bridging between CORBA objects, the CCA component model and the VTK pipeline, which is among several other we have developed so far. A picture of this application is given in Figure 6.3. The goal of the application is to solve the heat distribution over a two-dimensional domain, given the boundary conditions. The heat distribution follows the Laplacian equation, which is a second order partial differential equation. We use the Finite Element Method (FEM) to reduce the Laplacian partial differential equation to a linear equation $Ax = b$, where A is the matrix obtained from the FEM, x is the solution (the temperatures over the sample points) and b is a vector that reflects the boundary conditions. The input to this problem is the boundary conditions (the temperatures at the boundary of the domain) and the location of the sample points. We use the A matrix we obtain through FEM and the solution vector b to solve for x. The output x is the temperature at each sample point. Temperature at other points is estimated with interpolation.

6.2.1 SCIM Implementation

The bridges in the example application have been developed between the three application paradigms after they were added as component models in the SCIRun2 framework. Several choices exist for handling the control between CORBA, CCA and VTK. For this example, we chose the control to move in the direction from CCA to CORBA and then



SCIRun2
Framework URL: socket://qwerty.sci.utah.edu:39164/141642176

Figure 6.3. An example depicting the heat distribution through an L-shaped domain, simulated within SCIRun2 by CORBA, CCA, and VTK components.

from CCA to VTK. That is, the execution of the resulting bridge application is started by CCA components that get the problem description from CORBA objects. The CCA components perform the computation and send the result to be visualized in 3D by VTK, within its renderer window.

One important challenge in this simulation is that VTK imposes a very rigid pipeline and predefined data types. On the other hand, the CCA component model is very flexible but does not include native support for the VTK visualization data. To facilitate the bridging between CCA and VTK we devised a way to translate certain two-dimensional and three-dimensional arrays into data that VTK is capable of handling, and developed routines to perform the data translation. These routines impose an overhead and also require that a single copy of the data is made for the bridging to take place accurately. Within SCIM, we provide an easy way to plug in data translation code that is emitted to the bridge when the data convention requires it.

For this example, we have defined several abstract SCIM templates that describe

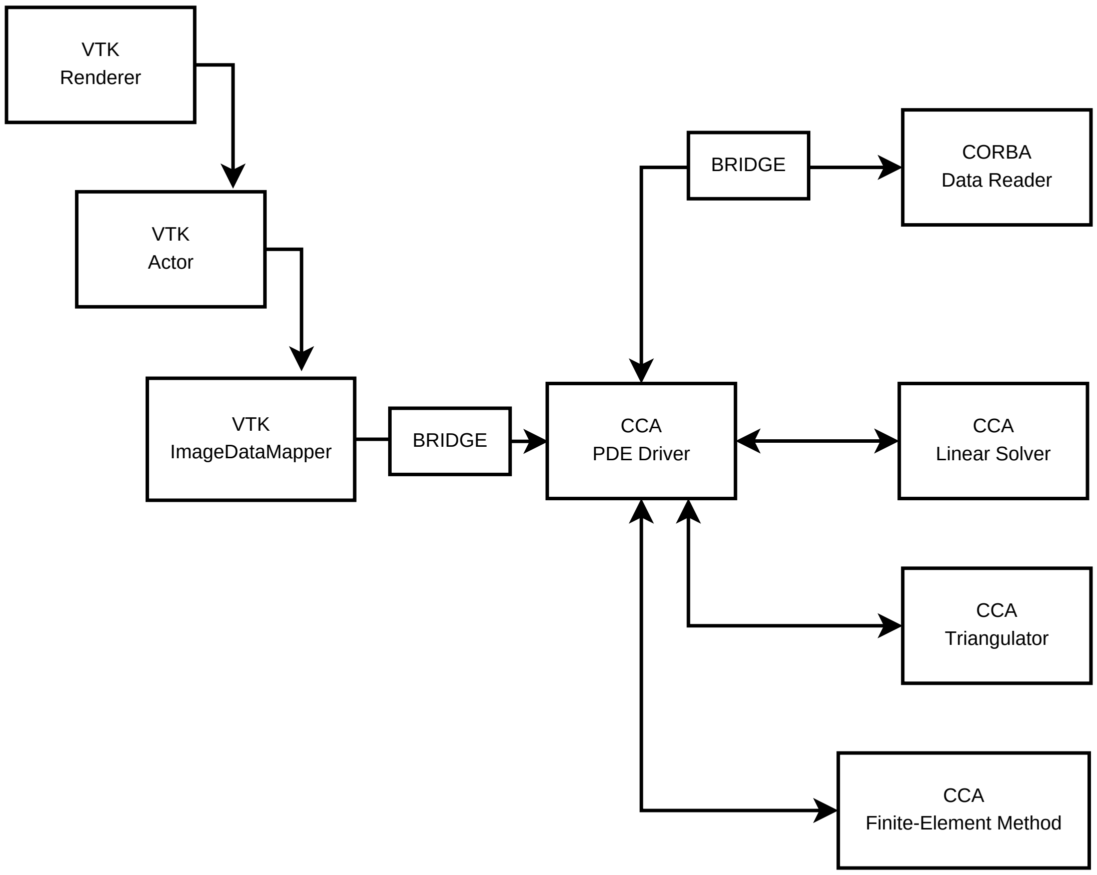


Figure 6.4. A graphical representation of the components in the heat distribution solver application.

the bridge code. For instance, one template generates bridge components that act as both a CCA component and a VTK object. The template consists of a method that receives invocations from CCA components, converts the data, and sets up the VTK pipeline (see Figure 6.4). When the VTK render window is opened or if it has already been open, this will result with an immediate rendering of the data computed by the CCA components. Using SCIM's capabilities, this template should naturally provide the generation of bridges for various components and interfaces.

The SCIM bridging templates have been shown to create bridges for several example scenarios. However, we are unable to claim complete generality and prove that a template works for all possible components. In spite of this, we feel that the simplicity of the template and its likeness to actual components can allow incremental improvement of the template to include the corner cases that may not work initially.

6.3 Linear Solver Interoperability Problem

Linear solver software libraries provide a computational abstraction of scientific computing algorithms. By using these libraries a chemist or physicist can focus on solving a problem in their domain without being impeded by implementing known numerical algorithms. Many solver libraries exist and and several are widely use in the several communities, such as PETSC [8, 7], HYPRE [12] and AZTEC [5]. Each numerical library provides its own Application Programming Interface (API), though most of the abstractions they provide are similar.

Our goal is to provide a user of one of these libraries with the ability to switch to a different library without major changes to the user code. The switch may be prompted by a new library's ability to provide a faster solution or a new functionality. We choose PETSC and AZTEC libraries for our task, as these are two numerical libraries that are very widely used. Enabling interoperability between these libraries involves designing an ontology between the API of PETSC and AZTEC and using SCIM to easily generate a bridge that can be used to translate between these libraries. This task of creating full interoperability is in progress at the time of writing; however, we are currently able to communicate between several examples.

We use SCIM's interface mapping directives to express the correspondence between linear solvers' arguments and methods, often connecting arguments produced and used in separate methods. The data models are easily leveraged to provide minor pointer dereferencing when necessary in argument code generation. They also declare global variables that save values that are produced in one method and used in another. These global variables are usually declared on a parameter basis, and would require some sort of locking if we use a multithreaded bridge. In a few cases we declare translated types and functions that generate translation code between complex data types. We find SCIM very useful in the task of generating a bridge for linear solver interoperability.

6.4 Bridge Performance Overhead

We analyzed the performance on a variety of generated bridges to determine the amount of overhead they impose on an application. The time spent using a bridge inflicts to the execution time of an application can be simply represented by this equation:

$$t_{bridge} = t_{caller} + t_{callee} + t_{datamanipulation}$$

, where t_{bridge} is the overall bridge time, t_{caller} is the time it takes to make an invocation with a given set of parameters by the caller component instance, t_{callee} is the time to make the same parameter invocation to the callee component and $t_{datamanipulation}$ is the time to translate or copy the data. To quantify the overhead that the bridge imposes we contrast it to a baseline situation where all the component instances belong to the caller's component model. By this we are comparing to a situation where a component instance has been converted from the callee component model in to the caller's. That approach would also likely require a significantly larger user effort than generating a bridge. However, we feel it gives a good overhead quantification from a user's perspective. The resulting overhead equation given our comparison application choice is:

$$t_{overhead} = t_{callee} + t_{datamanipulation}$$

We arrive to this result after subtracting the time it would take to invoke a component if all of the components belonged to the same component model. The overhead cost of a bridge is the time cost of an additional invocation plus the time it takes to convert the data between the two component models. In our experience, the callee invocation time is usually constant. While it may be somewhat significant in a distributed environment, in our experience it is significantly less than the time it takes to manipulate the data. This is because many component models are optimized to handle the inter-component communication part of their execution. Though we implemented the heat distribution problem as a nondistributed applications, it is clear by this application that the large overhead that is imposed by the transfer (and translation) of simulation data can rarely by matched by communication cost alone. The detailed overhead will be shown later by the timings we took from different bridge instances. We identify three data cases with regard to the performance of the bridge:

- the data need to be translated and copied
- the data require a copy
- the data are native and requires neither translation nor copying

If the bridge is able to pass the data from one model to the other natively, without any translation or copy of the data, the additional cost that a bridge imposes is negligible. This basically involves only passing the data by reference without making a copy within the bridge. Very few component models are able to communicate in such a straightforward way. In our experience, the type systems of heterogeneous component model prohibit easily converting a type belonging to one component model into a type belonging to another even if the memory structure of the data is similar. Often the only solution is to copy the data from one into another data structure. Mapped types are able to circumvent the copying but they also impose a terrible performance penalty for frequent access to data since the data are redirected using a proxy.

Due to these incompatibilities in the component models we often are required to copy the data (pass-by-value). This is the case in the CCA to CORBA bridge. Each CORBA component exists in its own process, which requires a copy from one address space to the other. This is no limitation of CORBA, but more a limitation of the way we have represented CORBA objects within our framework. It may be possible to improve this representation in order to move a greater the degree of CORBA applications that employ native data category.

This result indicates that if we converted the data to an encompassing protocol (e.g. SOAP) within our bridge, as component standardization that we discussed earlier, for most situations we would achieve similar performance characteristics as not doing so. This is because most situations are forced to make a simple copy of the data within the bridge. However, as we discussed before, our solution is more general and does not prohibit the use of a common protocol as an intermediary among all component models. Therefore the user of our system is free to follow that approach if he or she chooses to.

Situations exist like the one we have described in the CCA to VTK bridge that require complete data translation. This requires a more sophisticated bridge that imposes both a copy of the data and a numerical translation. This can affect performance in scenarios where the quantity of data is significant. The actual performance depends on the quantity of the translated data that is required, as well as the efficiency of that translation. We feel that the performance of the translation can be improved by using better numeric algorithms, however it will always significantly lag behind the copy-only bridge implementation. The copy-only implementation, on the other hand, will be significantly slower than being able to pass the data natively through the bridge.

Figure 6.5 shows performance benchmarks taken from the CCA to CORBA, CCA to VTK, and a no-copy (native) bridging example. The benchmarks were taken on a dual-processor Pentium4 Xeon machine running a Linux OS. The native example was modeled as a CCA to CORBA invocation while passing only a pointer, representing passing the data by reference. It resulted with an average time of 40 microseconds, which

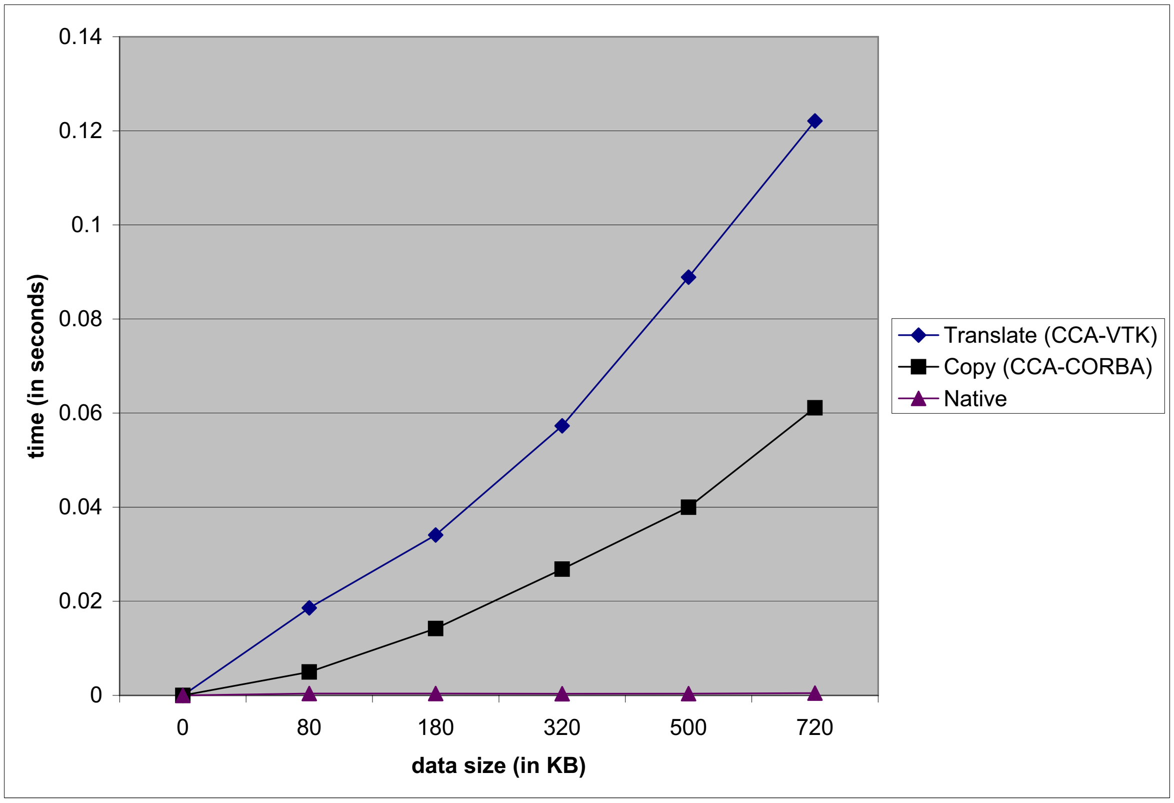


Figure 6.5. Performance of three types of bridges, differentiated by the type of data manipulation they perform.

we feel is not significant. The time grew at a constant rate as the quantity of data was increased. We see a clear difference in the growth of the CORBA to CCA bridge time compared to the CCA to VTK. The overhead imposed by translating the data to fit the VTK mold is significant. The data transfer penalty modeled in this graph follows our theory regarding the overall overhead that a bridge imposes.

Table 6.1 displays a separate bridge performance benchmark. It uses the PETSC to AZTEC bridging scenario to show the performance of two example applications using a hand-written bridge compared to using a SCIM generated bridge. Both example appli-

Table 6.1. Execution time of PETSC to AZTEC bridge in seconds, averaged over ten executions.

	SCIM generated PETSC-AZTEC	Hand-written PETSC-AZTEC
Example1	0.0131	0.0125
Example2	0.1712	0.1710

cation that were used for this instrumentation belong to the set of examples distributed with PETSC. The results show no large noticeable difference in the execution times of the applications. Although the hand-written bridge's average execution was faster by a small fraction of a second in each application, this fraction did not change from the smaller to the larger application. We attribute this fraction to minor differences in the saving a loading of variables transferred between methods in each bridge.

CHAPTER 7

THESIS EVALUATION

In this section we focus on evaluating the merit of this dissertation in supporting the claims made by the thesis statement. We begin by revisiting the thesis statement that guided our work:

A broad and practical method to express differences between software models into a specification can be leveraged for the creation of a bridge between two specific heterogeneous software applications belonging to separate models that supports efficient application execution.

This dissertation initially needs to show that we have expressed component model difference, and that these expressed differences can be used to create a bridge between two component models. This bridge ought to be produced by automatic and practical means to support efficient interoperation between the two component instances. In these sections we elaborate on each of these statement parts, finding proof for each of the claims in the design of SCIM and its associated group of bridging tools.

7.1 Expresses Software Model Differences

In the thesis statement we use the terms software model and software application instead of their component-ized versions: component model and component application. We used component software terms in our discussion thus far. However, component are not the only domain where we have applied bridge based interoperation. In this work we extend the common definition of what constitutes a component model or a component instance. Inspired by SCIRun2's handling of multiple component models through its meta-component model interface, our implementation considers all software that implements a firm interface and has a set of input parameters and output parameters as a component. Many of the bridges that we can create between two heterogeneous applications can perform their tasks for objects or even for operating system constructs such as pipes. UNIX pipes have a firm interface consisting of several system calls; they take some data and produce some other data. Therefore, for the purposes of our interoperability mechanism they behave as components do. Thus far, we have experimented with bridging the Visualization Toolkit (VTK) which utilizes objects as the unit of construction. We also demonstrated a simple example involving UNIX operating system constructs: pipes and signals.

The encompassing method to express differences between software models, noted in the thesis statement, is seen in two parts of our system: SCIM template and component meta-model with the bridging language. Our design enables several ways of expressing component model differences both closer and further away from SCIM and the actual bridge implementation. At the level closest to SCIM, the template expresses the differences among component models by showing how they interoperate. The template is closest to the lines of code of the actual implementation. The component meta-model expresses features that belong to an individual component model. In doing so, it provides a level ground upon which we can express model differences. Using the bridging language that is based on the component meta-model we combine two or more component meta-models expressing the differences among software models in an encompassing way.

The SCIM template mechanism can be used to express almost any bridge. Its closeness to the generated code enables it to create a bridge with logic to fit any situation. The SCIM template's power comes at the cost of software engineering effort by its designer. On the other hand, the component meta-model provides a significant improvement in the ease of expressing component models and is of greater software engineering value to a user. The component meta-model's value comes at the cost of expressibility; the meta-models are not able to express all component structures as a SCIM template does. Table 7.1 provides a list of component differences and the ability of expressing this difference using the component meta-model. This identifies the bound on the current version of component meta-models to serve new interoperability scenarios.

The SCIM template, component meta-model and bridging language can be leveraged to bridge two specific heterogeneous component (or software) models. This is apparent in the design of our system and through the examples we have demonstrated. The thesis statement claim that the method we defined to express differences between software models can be used to create a bridge is shown to be true by our design choices.

Table 7.1. Component differences and their expressibility by the component meta-model.

Type of Component Difference	Component Meta-Model Expressibility
Control	partial
Data	yes (through SCIM)
Interface	yes (through SCIM)
Component Model Structure	yes
Distributed	no
Lifecycle Control	no
Parallel	partial
Native DB Access	no

7.2 Automatic

Many aspects of the code generation of bridges can not be easily automated. The user is required to match interfaces, provide information about data and provide a SCIM template or a bridge language specification with underlying meta-models. A completely automatic solution within this problem domain is very challenging and likely impossible using current computer science approaches. This is due to several undecidable problems that present themselves upon considering automatic interoperation between a set of random software applications. The aim of this thesis is not complete automation of the task, but rather partial and practical user-guided automation in enabling interoperability between two autonomous applications. In doing so, we have already improved on the most common paradigm in this domain, which involves manually writing a bridge or converting applications from one component model into another by hand. We provide a structured and improved way of handling the tasks required for software interoperability. Apart from this, the benefits of automation that our approach provides can be found in two additional aspects: the ability to create new bridges using component meta-models and the economies of scale that SCIM provides.

A meta-model defines in a complete way the necessary information to generate a bridge involving a particular component model, instance or port. An already defined meta-model can be reused to create a new bridge by a new bridge language specification. In this fashion, the meta-model enables reuse and eases the creation of new types of bridges.

Once a general SCIM template is developed and available, it enables code generating bridges using SCIM for any two component instances. The user pays a significant cost up-front, by creating a general and robust SCIM template. However, economies of scale can be achieved and the initial cost can be amortized over a large number of generated bridges.

Table 7.2 displays lines of code measurements for three hand-written bridges compared to the sum of SCIM templates, interface mapping directives and data models. A SCIM template contains the same structure as the code generated bridge it produces, which is relatively similar to the bridges that we hand-wrote. However, the table indicates that the lines of code a user wrote for interfaces containing several methods were often significantly more than those needed to perform the same task using SCIM. This implies that an interface that contains more methods produces greater benefit for SCIM users in terms of lines of code compared to a hand-written bridge. It also follows that as the number of interfaces grows, we see an analogous growth in SCIM's utility. We observed in several scenarios that as an interface grows larger or interfaces grown in number this benefit becomes more pronounced. One example of this is the PETSC to AZTEC problem scenario and observed a total of 152 lines of code for SCIM (including all user written code) compared to 303 for the hand-written bridge for a seven method interface. In Figure 7.1 we see the difference in lines of code for the same PETSC-AZTIC example as we increase the number of bridged methods in the interface. The benefit of SCIM's use comes through in this lines of code measurement, but it is further pronounced by SCIM's division of the interoperability task as well as the ability to easily handle larger interfaces. In addition, SCIM templates can be reused to generate a bridge for any group of components that belong to the set of component models of the SCIM template.

To better illustrate the reuse of SCIM templates, we consider the use of the template we created between the CCA and VTK models for the finite element method problem scenario (details in Chapter 6). In the course of creating this example, we experimented with connecting several different CCA and VTK component instances using SCIM and the template we developed. In the course of this process, we used SCIM to generate

Table 7.2. Number of lines of user code for a hand-written and a SCIM generated bridge.

Bridge Type	Number of Methods	Hand-written Bridge	SCIM Overall
CCA to Babel	16	163	72
CCA to VTK	1	75	113
PETSC to AZTEC	7	303	152

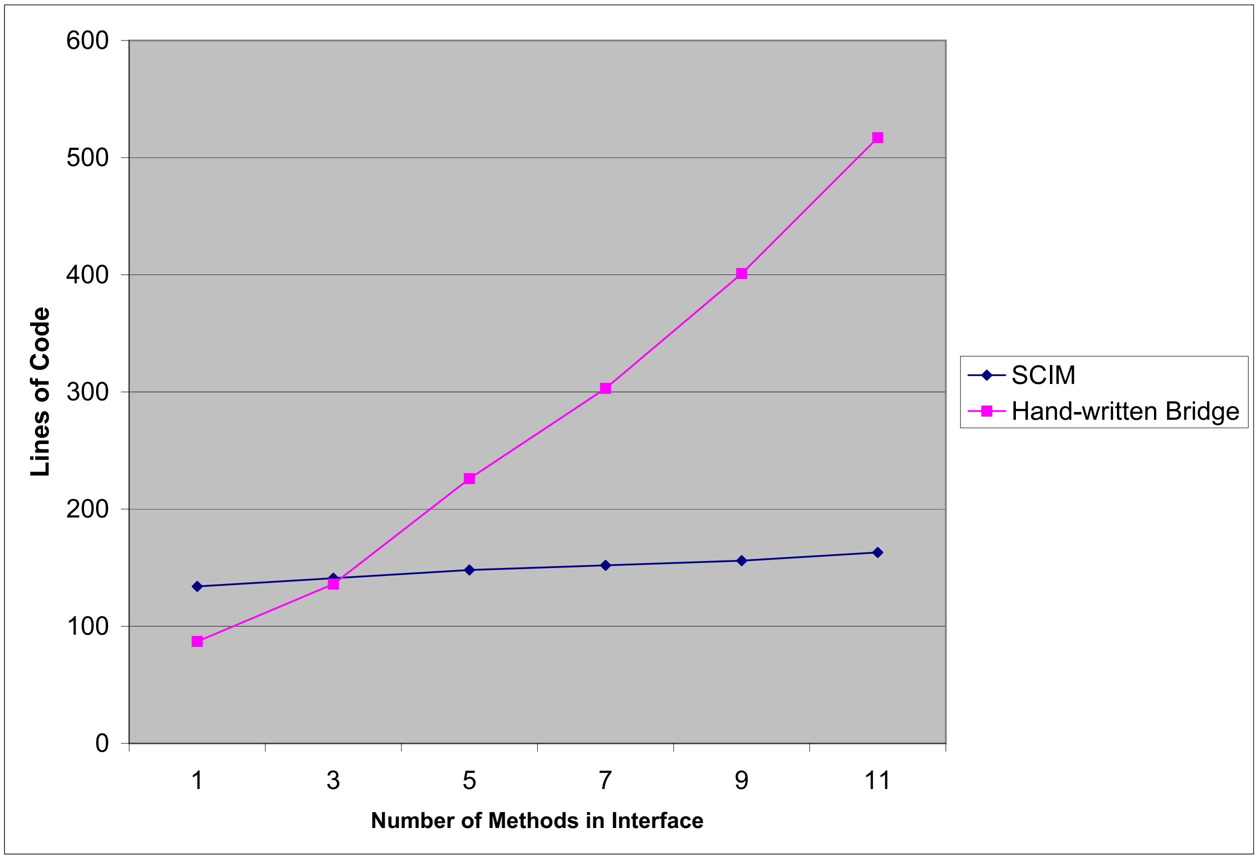


Figure 7.1. A comparison of the lines of user code (template, interface mapping directives and data models) needed for SCIM and the lines of code used for a user to write the same bridge by hand.

over 350 lines of code that accommodated four bridge connection scenarios. The CCA to VTK template we used consists of only 81 lines of code. This is an over 400% gain in productivity in terms of lines of code. This comparison illustrates that a SCIM template is significantly more useful than hand-written interoperability code for several connections. By SCIM's automation and economies of scale we can achieve a growing level of benefit to the user.

7.3 Practical

Practicality in this domain is defined as the user's ability to create a bridge as straighforwardly as possible. Our tools help the user avoid pitfalls and spend a minimum amount of effort to solve the interoperability task. They perform this in two ways:

- By providing a structured approach through SCIM where the user is prompted to provide the necessary commands or specifications to the system. The structured approach reminds the user of the list of necessary tasks while splitting the complex task into manageable pieces.

• By automatically generating a bridge based on knowledge of a component model and a bridge language specification the user avoids the intricacies of low level bridge design and instead focuses on expressing each component model and providing a high level specification of bridge behavior.

SCIM is a tool aimed at achieving interoperability between two component instances, once the general problem of interoperability between two component models has been solved through a SCIM template. The ability of SCIM to combine ways of handling data and interface differences adds to the number of interoperability problems it is able to handle. These aspects of SCIM that enable it to solve a greater degree of problems, also enable it to be more practical from a user's standpoint by providing a separation of concerns. However, writing a SCIM template can be a very difficult and intricate task. A bigger gain for the user is the accurate creation of bridges (or SCIM templates) based on component meta-models and using the bridge generation language. This approach provides the user with a general way to express each component model, then describe how these models interoperate on a high-level.

To quantify the improvement that the semantic tools provide to component interoperability, we compare the lines of code of a component meta-model and bridge language specification to lines of code in a SCIM template. Through this comparison we want to evaluate the ease of use of our approach to semantic bridge generation. The SCIM template's number of lines of code is very similar to the number of lines of a hand-written bridge. In fact, a SCIM template is essentially a hard written bridge made more general by inclusion of variables, code, and the help of data models and interface mapping directives. Therefore, the comparison is valuable in providing a benchmark for the effort required to construct a bridge (or SCIM template) by hand versus semiautomatically using our tools. One aspect that does not come through in the lines of code measure is that SCIM templates are much more difficult to express: containing both text and intertwined code.

Table 7.3 shows the number of lines of code that are used to model the framework and a component in the SCIRun2 component framework as well as the lines of code cost per each port. In Table 7.4 we display the lines of code cost for the bridge language specification and SCIM template for three interoperability scenarios. A bridge between

Table 7.3. Number of lines of code for meta-model parts.

Element	Lines of Code
SCIRun2 Framework	33
SCIRun2 Component	23
Base SCIRun2 Models Total	56
CCA Callee Port	29
Babel Caller Port	13
Babel Callee Port	49
VTK Out Port	29

Table 7.4. Number of lines of code for bridge language, SCIM template and semantic tools.

Bridge Type	Bridge Language	SCIM Template	Semantic Tools
CCA to Babel	13	63	111
CCA to VTK	19	81	133
Babel to VTK	20	98	154

component models **M1** and **M2** contains a callee port to receive **M1**'s invocation. It also requires a caller port to redirect the invocation to the implementation in the **M2** model. Therefore, the semantic approach to create a bridge between two component models CCA and VTK requires each of the following: base SCIRun2 elements (SCIRun2 Framework and SCIRun2 Component), a CCA Callee Port, a VTK Out Port, and the CCA to VTK bridge language specification.

To initially design a bridge using the semantic tools we require several meta-model elements connected through bridge language code. In all of our three interoperability scenarios the lines of code written using the semantic tools are greater than the ones written in a SCIM template (first two parts of Figure 7.2). Although this indicates that the cost of the semantic tools are greater than that of a hand-written bridge or a SCIM template, we argue that this is a small increase in lines of code to pay for the generality and modularity achieved. This generality will pay dividends in the ability to generate an even greater range of bridges and enable greater levels of interoperability.

Within the SCIRun2 component framework, where our examples where implemented, the cost of the component meta-model's implementation of the SCIRun2 framework and component elements is a static cost: the implementation of these elements is performed

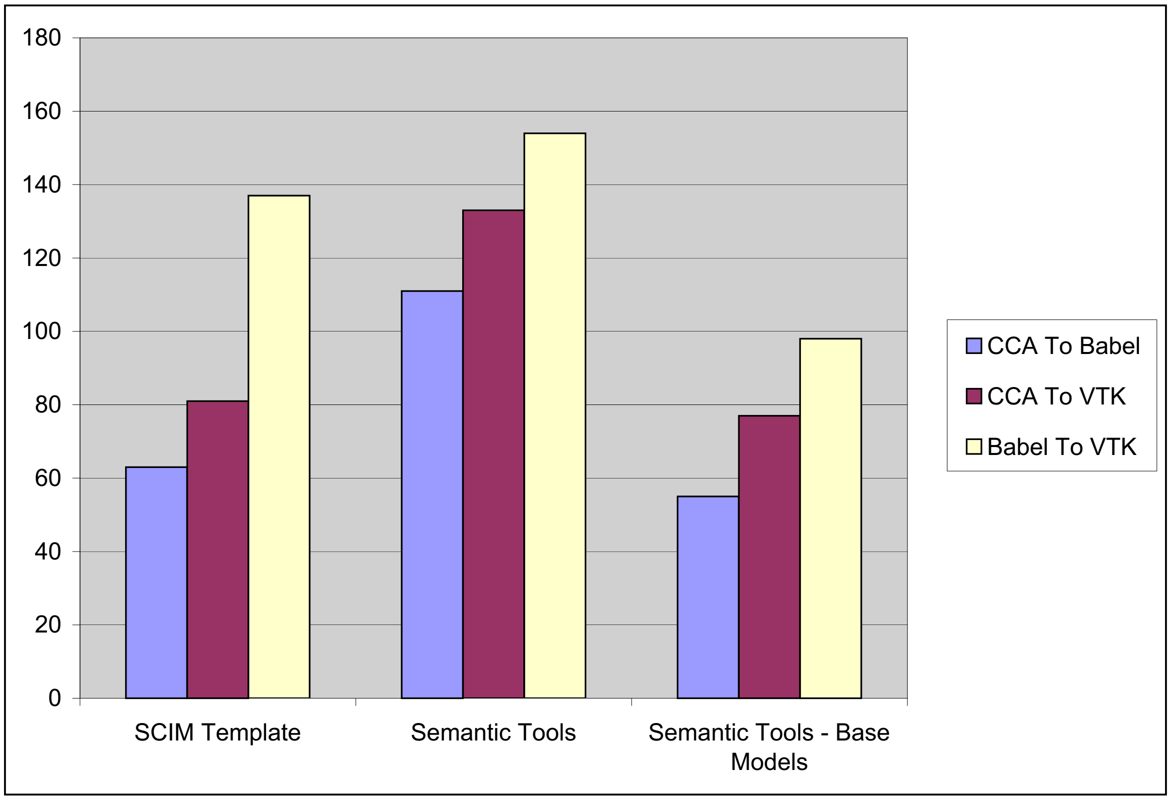


Figure 7.2. A comparison of the lines of code in three interoperability scenarios using: a SCIM template, a bridge language and component meta-models, and bridge language and component meta-models without some of the static cost of setting up the meta-models.

once and is valid for all interoperability scenarios involving SCIRun2. Many of the bridge instances designed by these tools reused existing implementations of these meta-model elements. In the third section of Figure 7.2 we plot the number of lines of code needed for each new component model cooperation within our SCIRun2 example. These numbers are less than the SCIM template. They numbers prove that the meta-model's modular design can be leveraged even further: combining two component models whose ports we have already modeled requires only a new bridge language specification, which we saw in Table 7.4 is on the order of two dozen lines of code.

To conclude, we feel that the semantic tools we implemented are useful and provide the user with a quick and practical approach to create a component model bridge. The lines of code we measured show that the cost of modeling and generating a bridge in this fashion is low and is amortized over many bridge specifications.

7.4 Supports Efficient Execution

The previous chapter elaborated on the efficiency of data transfer of a component bridge from an overhead standpoint. The resulting measurements showed that data transfer overhead is a significant portion of the overall overhead a bridge imposes on an application. This is due to incompatibilities in the data and types requiring a copy of the data within the bridge. This penalty is particularly costly in the domain that we have applied bridging to so far: scientific computing. Scientific computing usually manipulates large data structures that require massive memory copies when interoperating with other software.

However, here we argue that most of these effects are inherent in the problem. When dealing with interconnecting completely orthogonally designed component models' data, copying is often a penalty that can not be avoided. Our design choices allow that if a possibility exists not to copy the data, the opportunity can be capitalized upon within the bridge. The flexibility and generality of our bridge design allow any kind of code to be inserted in the bridge. This, in turn, allows creating the most optimal bridge given the constraints imposed by the participating component instances.

CHAPTER 8

CONCLUSION AND FUTURE WORK

This dissertation makes the case for automated component interoperability as a viable option in using scientific component software. It discusses potential scenarios where exists a need for disposable adapters and where large applications belonging to different component models need to intercommunicate. We analyze a few possibilities in component bridge design and chose a flexible path, allowing the inclusion of multiple component models. Our decision was exemplified through an implementation of a bridging framework that includes a bridge code generator (SCIM) and a tool that semantically creates bridges. Our system intends to leverage regularities in design of component instances and component models in order to ease and automate the interoperability task.

Multiple parts of the SCIM bridge code generator are discussed: interface mapping directives, data models and templates. Through these pieces SCIM encompasses the component bridge generation for component instances that may differ in interface expression, data definition, and underlying component model assumptions. The code generator is also able to read interfaces expressed in several programming languages and interface definition languages increasing its applicability. SCIM is valuable in generating bridges for two component instances belonging to two separate component models, where it is provided with a template that closely specifies the interoperability of component models. A template is tuned to each interoperability scenario, which usually involves two component models. SCIM easily achieves economies of scale in dealing with the same interoperability scenario when provided with the proper template as input. In an example involving a component solution to a 2D finite-element method problem, we experienced several fold reduction in lines of code using SCIM's ability to generate bridges based on a well-designed SCIM template. We also experienced a gain in productivity in terms of lines of code in a linear solver interoperability example. In addition, we analyzed the performance of several bridges during their execution. We concluded that the data transfer is most of time the largest overhead we have to deal with in component interoperability. Special effort ought to be made in the design of future bridges to create close coupling between the components and reduce data copying and manipulation.

We also presented a semantic approach to component interoperability. This approach can be used separately from SCIM to generate a bridge or it can be used in conjunction to provide SCIM with a template. Our semantic approach involves a bridge domain-specific language whose generative power is based on component meta-models. The component meta-models specify, in a generative way, the characteristics of each component model. Our bridging language is used to define the bridge between two or more component meta-models, which similarly specifies a bridge between the corresponding component models. These semantic tools add value by providing the user with a structured way to express a component model and a raised level of abstraction in expressing a bridge. We show that the meta-models can be reused to decrease the cost (in terms of lines of code) of creating the initial bridge between two component models. Once this initial bridge is created in the form of a SCIM template, we can leverage SCIM's generative power to produce bridges between the two component models with little or no effort.

Component interoperability in the scientific computing domain is a very useful capability. Several component models are often needed in this domain in order to easily produce a complete application. Scientific computing is usually dependent upon complex data and translating the data well from one format into another is very important for interoperability. The data models or a similar abstraction are necessary to make this task easier for the user.

While writing SCIM and the semantic tools we relied on strings to produce and emit generated code. This method of generating code works well, however other methods are also available. In retrospect, the use of macros to perform this task would have made this task significantly more straightforward.

One aspect of the future work of this project is to use the interoperability framework in a wide variety of applications belonging to several domains. This will prove its applicability as well as reveal any gaps or inconsistencies of our approach. The framework has been tested and used thus far in several concept-proving examples using a maximum of three to four component models. In the future, we want to expand its use in several real world problems. In addition, this set of tools can be used to generate code for tasks unrelated to interoperability such as code instrumentation or enforcement of simple invariants (contracts).

An additional goal of this project is to expand the handling of data through the data models. We believe that more work can be performed to automate the handling of component data types and to increase user-friendliness. One such issue is using the interface mapping directives to specify the order of allowed method invocations within an interface. We would enable SCIM to enforce the method ordering at bridge runtime by emitting additional code.

Another area of improvement would be enabling the faster creation of component meta-models by automatically inferring structure based on user tags placed within component code. This approach would increase the productivity and speed of expressing a new component meta-model.

APPENDIX A

INTERFACE MAPPING DIRECTIVES GRAMMAR

The following is the grammar of the interface mapping directives coupled with comments and explanations regarding their purpose and use. The grammar is expressed in Extended Backus-Naur Form (EBNF).

```
COMMAND:= /*
           * Maps one interface (and it's parts) into
           * another. The %map command is only useful to
           * combat name discrepancies in interfaces
           * where the underlying abstraction is the same.
           */
           '%map' IDENTIFIER '->' IDENTIFIER NAMEMAP* ;
           |
          /*
           * A map command that applies to all interfaces.
           */
           '%map' NAMEMAP* ;
           |
          /*
           * Map commands where one of the interfaces is
           * not defined by an IDL, so user defines it using
           * the 'NEW:' construct.
           '%map' 'NEW:' IDENTIFIER '->' IDENTIFIER ;
           |
           '%map' IDENTIFIER '->' 'NEW:' IDENTIFIER ;
```

```
 |
/*
 * Define a set of interfaces to be part of the
 * ``in''/''out'' component.
 */
 '%ininterface' IDL_STATEMENT* '\%ininterface' ;
 |
 '%outinterface' IDL_STATEMENT* '\%outinterface' ;
 |
/*
 * Excludes all currently defined ``inout'' interfaces
 * from code generation. These interfaces are ones
 * that do not require mapping and are the same in
 * both bridged components.
 */
 '%inoutomit' ;
 |
/*
 * Excludes a particular ``inout'' interface from code
 * generation.
 */
 '%inoutomit' IDENTIFIER ;
 |
/*
 * Remaps an interface that was erroneously excluded
 * using the '%inoutomit' command. Usually used after
 * the issue of the general version of that command.
 */
 '%inoutremap' IDENTIFIER ;
 |
/*
 * Declares a map for an object data type that is to
 * be transfered between two bridged components
```

```
           */
           '%datamap' IDENTIFIER '->' IDENTIFIER NAMEMAP* ;

/*
 * Second level of mapping, usually representing the map of methods
 * or data types.
 */
NAMEMAP:= /*
           * A map between one name and a set of one or more others, with
           * the possibility of defining a special function that translates
           * the differences between two data types (if the names are datatypes).
           */
           '>' IDENTIFIER '->' IDENTIFIER* [: IDENTIFIER] SUBNAMEMAP* ;

/*
 * A third layer of mapping. Similar to the second layer, it allows the
 * definition of a conversion function name for data type mapping.
 */
SUBNAMEMAP:= /*
              * A third layer of mapping. Similar to the second
              * layer, it allows the definition of a conversion
              * function name for data type mapping. A identifier
              * followed by a double colon can be used to represent
              * that an argument belongs to a method outside of this map.
              */
              '>>' [IDENTIFIER'::'] IDENTIFIER '->' [IDENTIFIER'::']
                   IDENTIFIER* [: IDENTIFIER] ;
              |
             /*
              * The ``in'' argument can be defined by its ordered
              * position in the method.
              */
              '>>' '#' INTEGER '->' [IDENTIFIER'::'] IDENTIFIER*
```

```
                    [: IDENTIFIER] ;

IDENTIFIER:= (A-Za-z)(A-Za-z0-9._)* ;
             |
            /*
             * The string ‘‘return’’ is a special identifier denoting
             * the return value of a method.
             */
             ’return’ ;

IDL_STATEMENT:= /*
                 * Some IDL text that we are able to parse
                 */
                 {*} ;
```

APPENDIX B

GLOSSARY OF SCIM TEMPLATE VARIABLES

A SCIM template uses variables and methods to adapt to the specific bridge we are trying to create. The variables and methods that are available are presented in three sections: global, interface, and method. The sections delineate which layer of the output the variables are representing. SCIM generates each layer hierarchically by processing the template as it reads IDL data. The variables can be subdivided in another way; in terms of whether they are part of the component making the invocation (“in”) or the component receiving this invocation through the bridge (“out”).

B.1 Global

•$inPackage

A package name, as defined by a specific IDL, defined by the “in” model.

•$outPackage

A package name, as defined by a specific IDL, defined by the “out” model.

•`<interface>interfaceCode</interface>`

A command that repeats `interfaceCode`, which includes raw text and variables, for every interface encountered in the IDL specification being read.

•$iDM

A reference to the data model object tbat describes the data of the component model of the ”in” component.

•$oDM

A reference to the data model object tbat describes the data of the component model of the ”out” component.

B.2 Interface

●$inInterfaceName

The name of the “in” interface SCIM is currently processing as defined by the IDL.

●$outInterfaceName

The name of the “out” interface SCIM is currently processing as defined by the IDL.

●`<method>methodCode</method>`

A command that repeats `methodCode`, which includes raw text and variables, for every method encountered within an interface. This command is used pervasively in templates in order to iterate and produce code for each interface method.

B.3 Method

●$inMethodName

The name of the “in” method SCIM is currently processing as defined by an IDL.

●$inMethodType

The return type of the currently processed “in” method.

●$outMethodName

The name of the “out” method SCIM is currently processing as defined by an IDL.

●$outMethodType

The return type of the currently processed “out” method.

●$args

Array of arguments that are part of the current method. The args array defines two methods `$args[n].type` and `$args[n].name` able to extract a string representing the type and name of the argument respectively.

APPENDIX C

DATA MODEL BASE CLASS

EmitModel is a base class defining all possible methods that concrete data models that extend this class provide to SCIM templates. The implementation is in the Ruby programming language, while most of the code implementing the behavior of this class is omitted. The current data model implementation is limited to C++ as the code emitting language.

```
class EmitModel

  #
  # Convert array of IDL args to a corresponding array
  # in the emit language (C++)
  #
  def createEmitArgs

  #
  # General function to return a string containing all
  # arguments' signatures. IDL arguments are stored in $args global
  # variable for each method.
  #
  # @first -- concatenate this to beginning of resulting string
  # @stride -- concatenate this after each argument signature
  # @last -- concatenate this to end of resulting string
  #
  def emitArgSignatures( first, stride, last )

  #
```

```
# General function to return a string containing all
# arguments' names
# (see emitArgSignatures for parameter description)
#
def emitArgNames( first, stride, last )

#
# Prints out a list of arguments of a method,
# formatted as if calling the method.
#
def outCallArgs

#
# Prints a comma delimited list of arguments'
# fullnames of a given method
#
def outDefArgs

#
# Same as outDefArgs but starts with a
# comma if args exist
#
def commaoutDefArgs

#
# Prints a semicolon+newline delimited list of
# arguments' fullnames of a given method
#
def semioutDefArgs

#
# [idl_type -> emit_type (c++)]
# Function has to be defined by a concrete data model. It defines
# the translation between idl types and programming language types,
# which is specific for every data model.
#
def idlToEmitType( arg )

#
# Defines an IDL to Emit Type mapping that comes from
# the interface mapping directives of SCIM
#
def addTypeMap( idl_T, emitT )

#
# Register a data conversion function for
# translated types
#
def registerConvertFunc( type1, type2, func )

#
# Determine the way this argument was passed: by value
# or by reference. (C/C++ specific)
#
def passedType( argT )

#
# Function that matches C++ types and emits translations
# between types that differ only in the way they are passed (val,ref)
#
def emitCppMatchT( arg, argTp, calleeArgTp )

#
# Data translation function. All data translation is emitted upon the
# invocation of this function.
# @emitTo -- EmitModel instance representing the model the
#            translation is to.
#
def emitDataTranslate( emitTo )

#
# Add a user specified translation type that won't be derived by
# calling the idlToEmitType function and isn't found in the IDL
# specifications.
# calleeT -- user specified translation type
#
def addCalleeTranslateT( calleeT )

#
# Limit the translation of data (produce an error otherwise) to
# the types declared. A type "Any" is allowed to denote no limitation.
#
def limitDataType( type )

#
# Adds the definition of a new mapped type into the local list, returns
# a pointer to that mapped type that allows further constructs to be
# added to this map.
#
def addMappedType( typeFrom, typeTo )

#
# Generates code that is automatically produced based on information about
# the method variables that the data model contains. This may generate code
# that saves a variable used in a different method into a global variable.
#
def emitProcessMethodVars

#
```

```
  # Generated code that declares global variables that are used as saving points
  # to transfer variables between methods.
  #
  def emitGlobals

  #
  # Produces code that handles the return from a ‘‘out’’ method invocation.
  #
  def emitProcessReturn

end     # of EmitModel class
```

APPENDIX D

GLOSSARY OF BRIDGE DOMAIN SPECIFIC LANGUAGE

Our implementation of the bridge languages is written in the Ruby programming language. The following is a glossary of all the methods we provide in our bridge language. The methods are subdivided by the subject on which they are issued; port methods are issued on a port instance, component methods on a component instance, etc. Global methods are issued without an instance.

D.1 Global Methods

•`selectFramework( frameworkName )`

Selects a framework meta-model with a given `frameworkName` and loads its symbols. A framework meta-model encapsulates a group of component meta-models and a single base bridge component. This command is usually the first one issued by bridge domain specific language programs.

•`setVar( var, value )`

Mutates a variable name `var` by assigning it the value in `value`. The variable does not need to be pre-defined, this method will define a new variable if none with the same name exist.

•`getVar( var )`

Returns the value stored in variable named `var`. If this variable name is undefined, the method returns `nil`.

D.2 Framework Methods

•`getBaseBridgeComponent()`

A framework retreives an instance of a base bridge component upon this method's invocation. The base bridge component is a placeholder that we use to construct a bridge. The meta-models may define certain features of a base bridge component that are appropriate for the deployment environment.

•`getNewPort( portName )`

Allocates and returns a new port `portName` based on a meta-model description of a port with the same name.

•`init`

This method signifies emitting the code to initialize the framework, as defined by the framework meta-model.

•`destroy`

Signifies emitting code corresponding to exiting the framework.

D.3 Component Methods

•`addPort( portInstance )`

Adds a port represented by `portInstance` into a component. The port instance needs to be allocated and passed into this method.

•`init`

Code emitter method used to signify the component's initialization.

•`destroy`

Emits code when this component is deallocated.

•`connect`

This method emits code that signifies a component connecting to another component.

•`onMethodInvoke( methodName ) { ...}`

Catch-all method for events not present in the language. This method sets up event whose bridge language commands get issued upon the execution of a `methodName` method.

•`onInit() { ...}`

This event is used to define a set of bridge language (usually code emitting) commands that correspond to the scenario of initializing this component.

•`onDestroy() { ...}`

This event is used to define a set of bridge language (usually code emitting) commands that correspond to the scenario of deallocating this component.

D.4 Port Methods

•`getDataModel()`

Retreives the data model defined for this port, returns `nil` if a data model was not defined. Data models are defined in the component meta-mode for a particular port.

•`init`

Code emitter method used to signify the port's initialization.

•`destroy`

Emits code when this port is deallocated.

•`connect`

This method emits code that signifies a port connecting to another component's port.

•`invoke`

Emits code corresponding to an invocation of this port.

•`sendReply`

A code emitter method invoked when sending a reply from an invocation of the port.

•`receiveReply`

A code emitter method invoked when receiving a reply from an invocation of the port.

•`onMethodInvoke( methodName ) { ...}`

Catch-all method for events not present in the language. This method sets up event whose bridge language commands get issued upon the execution of the port's `methodName` method.

•`onInit() { ...}`

This event is used to define a set of bridge language (usually code emitting) commands that correspond to the scenario of initializing this port.

•`onDestroy() { ...}`

This event is used to define a set of bridge language (usually code emitting) commands that correspond to the scenario of deallocating this port.

•`onInvoke() { ...}`

An event used to define bridge language methods executed upon the invocation of this port.

D.5 Other

D.5.1 Data Model

•`emitDataTranslate( emitTo )`

Data model method used to signify where the data translation code should be inserted in the bridge. The `emitTo` variable signifies the data model that we are traslating in to.

•`registerConvertFunc( typeFrom, toper, func )`

Registers a code emitting function `func` that is able to create a translation between `typeFrom` and `toper` variables.

•`limitDataType( type )`

Used to limit the execution of the data model to a type named `type`. All other types result in an error. The command can be issued several times to create an array of allowable types.

D.5.2 Locking

Component meta-model ports or components may include the `Waitable` mixin to provide some locking support for multithreaded bridges.

- `waitLock()`

Method that emits code to acquire the lock if it's free, wait otherwise until lock is freed.

- `releaseLock()`

Method that emits code to release the lock.

REFERENCES

[1] Allen, R., and Garlan, D. Formalizing architectural connection. In *ICSE '94: Proceedings of the 16th international conference on Software engineering* (1994), IEEE Computer Society Press, pp. 71–80.

[2] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S. Kepler: An extensible system for design and execution of scientific workflows. In *Proceeding of the 16th Intl. Conference on Scientific and Statistical Database Management* (June 2004).

[3] Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., and 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)* (August 1999).

[4] Auerbach, J. S., and Russell, J. R. The concert signature representation: IDL as intermediate language. In *Proceedings of the workshop on Interface definition languages* (1994), ACM Press, pp. 1–12.

[5] AZTEC OO. http://software.sandia.gov/trilinos/packages/aztecoo/index.html, 2006.

[6] Balaji, Da Silva, A., DeLuca, C., Hill, C., and Suarez, M. The architecture of the earth system modeling framework. *Computing in Science and Engineering 6* (2004).

[7] Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[8] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing* (1997), E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., Birkhäuser Press, pp. 163–202.

[9] Beazley, D. M. *A Wrapper Generation Tool for the Creation of Scriptable Scientific Applications.* PhD thesis, University of Utah, 1998.

[10] Beazley, D. M. An embedded error recovery and debugging mechanism for scripting language extensions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference* (2001), USENIX Association, pp. 147–160.

[11] Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. Ptolemy: a framework for simulating and prototyping heterogeneous systems. 527–543.

[12] Chow, E., Cleary, A. J., and Falgout, R. D. Design of the hypre preconditioner library. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing* (Philadelphia, PA, 1999).

[13] Component Object Model. http://www.microsoft.com/com, 2005.

[14] Czarnecki, K., and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[15] Czarnecki, K., and Eisenecker, U. W. Components and generative programming (invited paper). In *ESEC: Proceedings of the 7th European software engineering conference* (1999), Springer-Verlag, pp. 2–19.

[16] Dashofy, E. M., van der Hoek, A., and Taylor, R. N. An infrastructure for the rapid development of xml-based architecture description languages. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering* (2002), pp. 266–276.

[17] Eclipse Modeling Framework. http://www.eclipse.org/emf, 2005.

[18] Enterprise Java Beans. http://java.sun.com/products/ejb, 2005.

[19] Floch, J. Supporting evolution and maintenance by using a flexible automatic code generator. In *Proceedings of the 17th international conference on Software engineering* (New York, NY, USA, 1995), ACM Press, pp. 211–219.

[20] Forkert, T., Kloss, G. K., Krause, C., and Schreiber, A. Techniques for wrapping scientific applications to corba components. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments* (April 2004).

[21] Fromherz, M. P. J., and Saraswat, V. A. Model-based computing: Using concurrent constraint programming for modeling and model compilation. In *CP '95: Proceedings of the First International Conference on Principles and Practice of Constraint Programming* (1995), Springer-Verlag, pp. 629–635.

[22] Geraghty, R., Joyce, S., Moriarty, T., and Noone, G. *COM-CORBA Interoperability*. Prentice Hall, 1999.

[23] Giese, C. Practitioner's report: Practical experience with frame-based generators. In *GPCE 2003: Generative Programming and Component Engineering: Second International Conference* (2003), Springer-Verlag.

[24] Goodale, T., Allen, G., Lanfermann, G., Masso, J., Radke, T., Seidel, E., and Shalf, J. The cactus framework and toolkit: Design and applications. In *Proceedings of the 5th International Conference on Vector and Parallel Processing* (2002).

[25] Herrington, J. *Code Generation In Action*. Manning Publications, 2003.

[26] Johnson, C., and Parker, S. The SCIRun Parallel Scientific Compouting Problem Solving Enviroment. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing* (1999).

[27] Kohn, S., Kumfert, G., Painter, J., and Ribbens, C. Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing* (Portsmouth, VA, March 2001).

[28] Konstantas, D. Object oriented interoperability. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming* (London, UK, 1993), Springer-Verlag, pp. 80–102.

[29] Lee, E. A., and Sangiovanni-Vincentelli, A. L. Comparing models of computation. In *Proceedings of the International Conference on Computer-Aided Design* (1996), pp. 234–241.

[30] Liu, L., Pu, C., and Han, W. XWRAP: An xml-enabled wrapper construction system for web information sources. In *Proceedings of the 16th International Conference on Data Engineering* (2000), IEEE Computer Society, p. 611.

[31] Luckham, D. C. Rapide: A language and toolset for simulation of distributed systems by partial ordering of events. In *DIMACS Partial Order Methods Workshop IV* (1996), Princeton University.

[32] Magini, S., Pesciullesi, P., and Zoccolo, C. Parallel software interoperability by means of corba in the assist programming environment. In *Proceedings of the 10 International Euro-Par Conference* (August/September 2004), vol. 3149 of *Lecture Notes in Computer Science*, pp. 679–688.

[33] OMG. *The Common Object Request Broker: Architecture and Specification. Revision 2.0*, June 1995.

[34] Parker, S., and Johnson, C. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95* (1995), IEEE Press.

[35] Raje, R. R., Auguston, M., Bryant, B. R., Olson, A. M., and Burt, C. A unified approach for the integration of distributed heterogenous software components. In *Proceedings of the 2001 Monterey Workshop* (2001), pp. 109–119.

[36] Rastofer, U. Modeling with components - towards a unified component metamodel. In *Proceedings of the ECOOP Workshop on Model-Based Software Reuse* (2002).

[37] Sauer, L. D., Clay, R., and Armstrong, R. Meta-component architecture for software interoperability. In *Proceedings of the International Conference on Software Methods and Tools* (2000).

[38] Schroeder, W., Martin, K., and Lorensen, B. *The Visualization Toolkit, An Object-Oriented Approach to 3-D Graphics*, 2nd ed. Prentice Hall PTR, 2003.

[39] Simple Wrapper Interface Generator. http://www.swig.org, 2005.

[40] Sousa, J. P., and Garlan, D. Formal modeling of the enterprise javabeanstm component integration framework. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems-Volume II* (London, UK, 1999), Springer-Verlag, pp. 1281–1300.

[41] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, second ed. Addison-Wesley Publishing Company, 2002.

[42] Tolvanen, J.-P. Making model-based code generation work. *Embedded Systems Europe* (2004), 36–38.

[43] Unified Modeling Language. http://www.uml.org, 2005.

[44] Vanneschi, M. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Computing 28*, 12 (2002), 1709–1732.

[45] Wegner, P. Interoperability. *ACM Computing Surveys 28*, 1 (1996), 285–287.

[46] Zhang, K., Damevski, K., Venkatachalapathy, V., and 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* (April 2004).

[47] Zhao, W., Bryant, B. R., Burt, C. C., Raje, R. R., Olson, A. M., and Auguston, M. Automated glue/wrapper code generation in integration of distributed and heterogeneous software components. In *Proceedings of Enterprise Distributed Object Computing Conference, Eighth IEEE International (EDOC'04)* (2004), pp. 275–285.