Automated Synthesis of Connectors for Heterogeneous Deployment

Tomáš Bureš
Charles University
Faculty of Mathematics and Physics
Department of Software Engineering
Distributed Systems Research Group
Malostranske namesti 25
118 00 Prague 1, Czech Republic
bures@nenya.ms.mff.cuni.cz
http://nenya.ms.mff.cuni.cz/

August 16, 2005

Abstract

Although component based engineering has alredy become a widely accepted parading, easy combining components from different component system in one application is still beyond possibility. In our long-term project we are trying to address this problem by extending OMG D&C based deployment. We rely on software connectors as special entities modeling and realizing component interactions. However, in order to benefit from connectors, we have to generate them automatically at deployment time with respect to high-level connection requirements. In this paper we show how to create such a connector generator for heterogeneous deployment.

1 Introduction and motivation

In the recent years, the component based programming became a widely accepted paradigm for building large-scale applications. There are a number of business (e.g., EJB, CCM, .NET) and academic (e.g., SOFA[12], Fractal[11], C2[10]) component models varying in maturity and features they provide. Although the basic idea of component based programming (i.e., composing applications from encapsulated components with well defined interfaces) is the same for all component systems, combining components for different component systems in one application is still beyond possibility. The main problems hindering this free composition of components comprise different deployment processes (e.g., different deployment tools), different ways of component interconnections (e.g., different middleware), and compatibility problems (e.g., diffferect component lifecycle and type incompatibilities). To allow for freely and uniformly composing, deploying, and running applications composed from components of

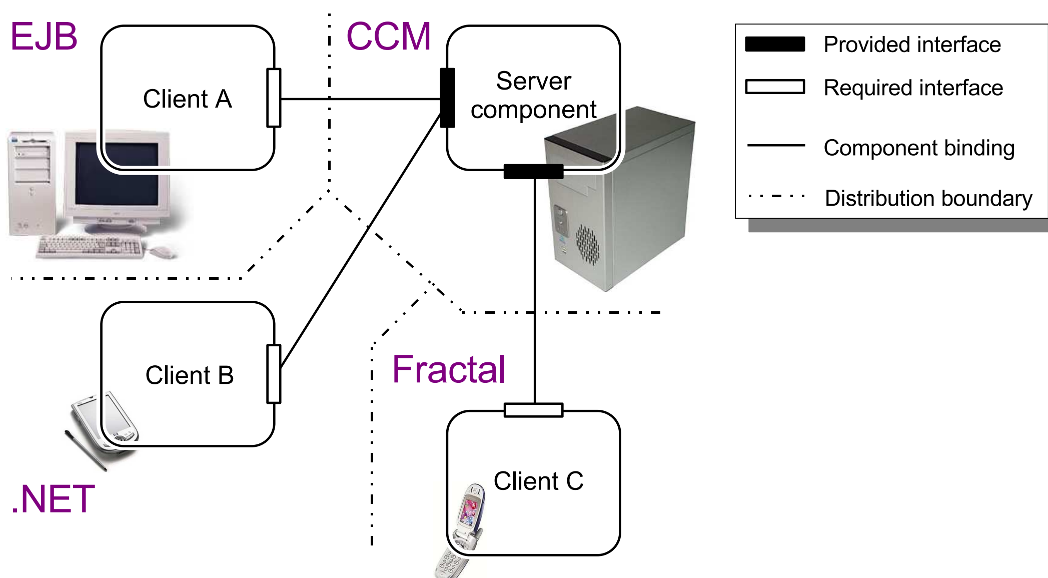

Figure 1: An example of heterogeneous component-based application

different component systems (as depicted in Figure 1) is the aim of heterogeneous deployment.

In our recent work we have used OMG D&C [13] as a basis for unified deployment. OMG D&C defines a deployment of homogeneous application in a platform independent way. It describes processes and artifacts used in the deployment. OMG D&C follows the MDA paradigm; thus, the platform independent description of deployment is transformed to a particular platform specific model (e.g., deployment for CCM). In order to allow for the heterogeneous deployment, we have extended the OMG D&C model to make it capable of simultaneously handling components from different component systems [9], and we have introduced the concept of software connectors [3] to the OMG D&C model to take responsibility for component interactions [5].

Connectors being first class entities capturing communication among components (see Figure 2) help us at design time, to formally model intercomponent communication, and at runtime, to implement the communication. Connectors seamlessly address distribution (e.g. using a particular middleware). They also provide a perfect place for adaptation (for overcoming incompatibilities resulting from different component systems) and value added services (e.g. monitoring). Important feature of connectors is that an implementation of a particular connector is prepared as late as at deployment time, which allows us to tailor the connector implementation to specifics of a particular deployment node.

The fact that connectors are prepared at deployment time, however, brings a problem of their generation. They cannot be provided in a final form in advance. Forcing a developer to be present at deployment stage to tailor connectors to a particular component application and deployment nodes is not feasible either. Our solution to this problem is to benefit from the domain specificity of connectors and to create a connector generator which would synthesize connectors automatically (i.e., without human assistance) with respect to a high-level

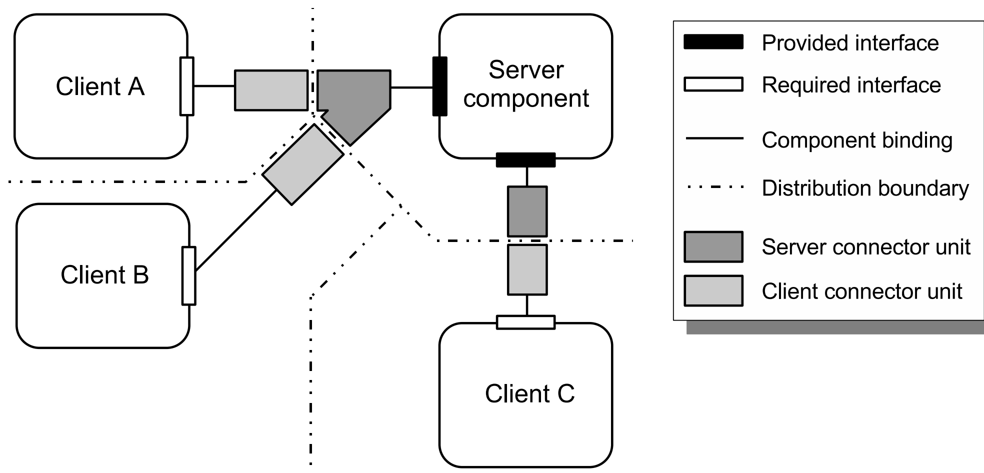

Figure 2: Using connector to interconnect components

connector specification. As the specification we take connection requirements, which are associated with bindings in OMG D&C component architecture (e.g., a requirement stating that a connection should be secure with a key no weaker than 128-bits). In our case the the connection requirements are expressed as a set of name-value pairs (e.g., $minimal_key_length \rightarrow 128$). Also, the connector synthesis reflects particular target deployment environment (e.g., if both the components connected by the binding required to be secure reside on one computer, then the security is assure even without encryption). The description of the environment is taken from OMG D&C target data model.

2 Goals and structure of the text

In this work we present our approach to automated connector synthesis for heterogeneous deployment. We base the work on our previous experiences with building a connector generator for SOFA component system [4]. However, the connector generation in SOFA was not fully automatic — a connector structure must have been precisely specified — and it did not allow for handling and combining different component systems (i.e., it was homogeneous). This report takes up the approach shown in [8] and improves it by modifying existing structures and techniques (e.g, the architecture of the generator, the type-system handling) and bringing in new features (e.g., as automatic architecture resolution).

In this paper, we show how to create a connector generator that allows for the heterogeneity and works fully automatically, generating connectors with respect to target environment and high-level connection requirements.

The paper is structured as follows. Section 3 shows the connector model we use. Section 4 describes the architecture of the connector generator and discusses certain parts of it in greater detail. Section 5 presents the related work and Section 6 concludes the paper.

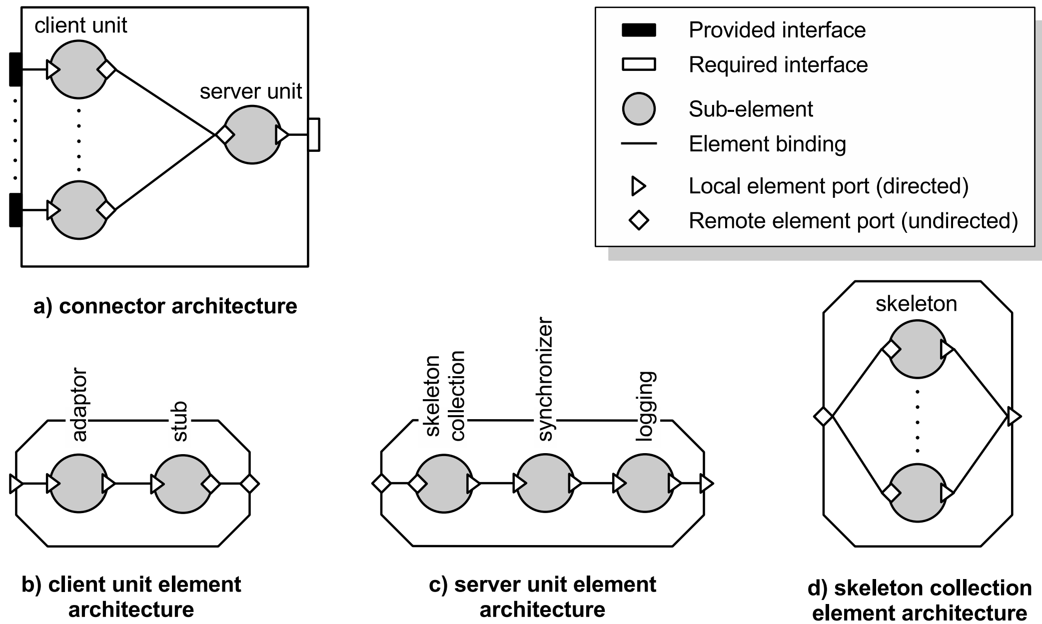

Figure 3: A sample connector architecture

3 Connector model

Software connectors are first class entities capturing communication among components. They are typically formalized by a connector model, which is a way of modeling their features and describing their structure. The connector model thus allows us to break the complexity of a connector to smaller and better manageable pieces, which is vital for automated connector generation. In our work we have adopted (with modifications) the connector model used in SOFA, since it has been specially designed to allow for connector feature modeling and code generation. In the rest of this section, we briefly describe this model along with the modification we have made to it.

Our connector model is based on component paradigm (see Figure 3. A connector is modeled as composed of *connector elements*. An element is specified by *element type*, which on the basic level expresses the purpose of the element and *ports*. Ports play the role of element interfaces. We distinguish three basic type of ports: a) provided ports, b) required ports, and c) remote ports. Elements in a connectors are connected via bindings between pairs of ports. Based on the type of ports connected we distinguish between a *local binding* (between required-provided or provided-required ports) and a *remote binding* (between two remote ports). Local bindings are realized via a local (i.e., inside one address space) call. Remote bindings are realized via a particular middleware (e.g., RMI, JMS, RTP, etc.). Since it is not possible to easily capture the direction of the data flow on a remote binding (i.e., who acts as a client and who acts as a server) we view these bindings as undirected (or bidirectional).

On the top-level a connector is captured by a *connector architecture*, which defines the first level of nesting. All inter-element bindings in a connector architecture are remote. Thus, elements in a connector architecture encapsulate

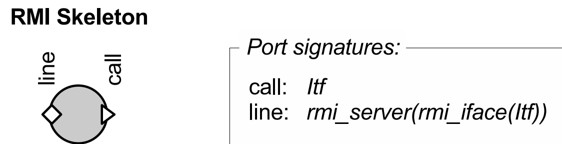

Figure 4: Specification of formal signatures of ports on an element.

all the code which runs in one particular address space. Alternatively, we call these elements on the first level of nesting *connector units*.

Elements in an architecture are implemented using *primitive elements* and *composite elements*. A primitive element is just code (or rather a code template as we will se later). Apart from code, a composite element consists also of an *element architecture*. Binding between sub-elements in an element architecture may be only local. Thus, an element cannot be split among more address spaces.

Figure 3 gives an example of a connector realizing a remote procedure call. The top-level connector architecture is in Figure 3a. It divides a connector to one server unit and a number of client units. An implementation of the client unit is given in Figure 3b. It implements the client unit as a composite element consisting of an adaptor (which is an element realizing simple adaptations in order to overcome minor incompatibility problems) and a stub. Notice that the stub exposes on the left-hand side a provided interface (i.e., local) and on the right hand side it exposes a remote interface, which is use to communicate with a skeleton on the other side using a particular middleware. An implementation of the server unit (in Figure 3c) comprises a logging element (which is responsible for call monitoring), a synchronizer (which is an element realizing a particular threading model), and element called skeleton collection. The architecture of the skeleton collection element is given in Figure 3d. Its purpose is to group a number of skeletons implementing different middleware protocols, and thus to allow the connector to support different middleware on the server side.

A concrete connector is thus built by selecting a connector architecture and recursively assigning element implementations (either primitive or composite). The resulting prescription stating what connector architecture and what element implementations is called a *connector configuration*.

Connectors are in a sense entities very similar to components, however, there are a few basic differences between components and connectors: a) Connectors are typically span different address spaces and deal with middleware, while component (at least the primitive ones) reside in one address space only. b) Connectors have a different lifecycle to components — remote links between connector units have to be established before a connector can be put between components, and connectors may appear at runtime as component references are passed around. c) Connectors are in fact templates, which are later adapted to a particular component interface (as opposed to components which have fixed business interfaces).

In our model we do not only view connectors as whole as templates, but we view also connector elements as templates. Thus, an adaptation of a connector implies adaptation of all elements inside a connector[1].

[1]Although this approach is more tedious, it allows us to perform static invocation inside the connector; as opposed to having elements with general interfaces and being forced to

As elements are templates, their signatures are variable, typically with some restrictions and relations to other ports of an element. To capture the restrictions and relations, we use interface variables and functions. An example is given if Figure 4. The skeleton element has two ports *call* and *line*. The RMI implementation of the skeleton element as depicted in the example prescribes no restriction on the *call*-port. The actual signature[2] of the *call*-port is assigned to the interface variable Itf. The formal signature of the *line*-port is more complicated. It expresses the fact the original interface Itf is accessible via RMI. It uses the variable Itf to refer to the actual signature of the *call*-port and two functions rmi_server and rmi_iface. The rmi_iface function changes the interface Itf by adding neccessary features so that the interface can be used by RMI — it modifies the interface to extend `java.rmi.Remote` and changes the signature of every method in the interface to throws `java.rmi.RemoteException`. The rmi_server function expresses that the interface it takes as a parameter is accessible remotely via RMI. Notice that we do not use the functions only to capture changes in the interface itself (e.g., rmi_iface), but also to assign a semantics (e.g., rmi_server).

To perform the adaptation of a connector to a particular component interface we need to know the actual signature of each element port of every element inside the connector. To realize this, we assign the adjoining components' interfaces to respective interfaces of the connector and let the interface propagate through the elements inside the connector. The result of this process is shown in Figure 5. The $'Service_v1'$ and $'Service_v2'$ in the example are names of a sample component interfaces; the $java_iface$ function returns an interface identified by a name. The adaptor element in the example solves incompatibilities between the two interfaces.

4 Connector generator

The connector model we have presented in the previous section very clearly outlines how connectors are generated — we synthesize a connector configuration and we assemble and adapt a connector accordingly. From the implementation point of view this is, however, more complicated.

An important fact to note is that we are not actually interested in building a connector as whole. Rather we want to build a server part and different client parts separately, as it better corresponds to the lifecycle and evolution of an application. Thus, in the rest of the text we focus on generation of connector units.

We have designed the connector generator from a few interoperating components (see Figure 6). The *generation manager* orchestrates the connector generation. The *architecture resolver* is responsible for creating a connector configuration with respect to high-level connection requirements and a target

perform dynamic invocation. The main benefit of static invocation over the dynamic one is better performance.

[2]When necessary we distinguish in this text between *formal port signature* and *actual port signature*. By formal signatures we mean the interface variables and restrictions as declared in an element implementation represented as a template. By actual signatures we mean the interfaces as values which are assigned to element ports and to which the element implementation is adapted. The terminology is in some sense similar to *formal* and *actual arguments* known from programming languages.

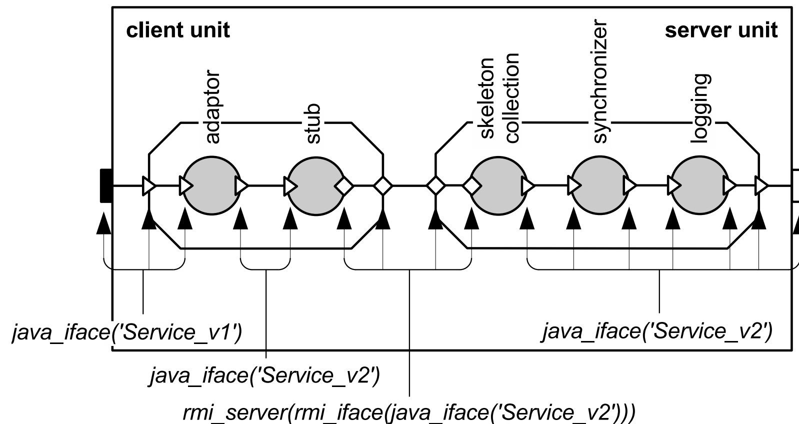

Figure 5: An example of interface propagation thorough a connector.

deployment environment (as described in Section 1). The *element generator* adapts element templates to particular interfaces and creates builder code for composite element architectures. Since we are interested only in connector units which are modeled also as elements, we can perform all the code generation only with help of the element generator. The *connector template repository* holds connector architectures and element implementations (i.e., code templates and composite element architectures). It is used by the architecture resolver to create a connector configuration and by the element generator to retrieve code templates to be adapted. The *connector artifact cache* is a repository where adapted elements are stored and from which they can be reused when needed next time. It addresses the problem that the code generation tends to be slow. Finally, the *type system manager* is a responsible for providing unified access to type-information originating from different sources and being in different format.

The synthesis of a connector configuration performed in the architecture resolver is discussed in more detail in Section 4.1. The element generator is elaborated on in Section 4.2 and the issue of handling different type-systems is explained and addressed in Section 4.3

4.1 Architecture Resolver

We have implemented the architecture resolver in Prolog. We fill the Prolog knowledge base with predicates capturing information about architectures and elements kept in the connector template repository. Then we use the inherent backtracking in Prolog to traverse the search tree of various connector configurations and to find the configuration satisfying our requirements.

For every element implementation in the connector template repository we introduce to the knowledge base a predicate `elem_desc` (see Figure 7). The purpose of the predicate is to build a structure representing an instance of the element in a connector configuration (we call the structure *resolved element*). The resolved element contains all information required by the element generator (i.e., the name of the element implementation, which is a reference to the element's code template, and actual signatures of ports). In the case of a

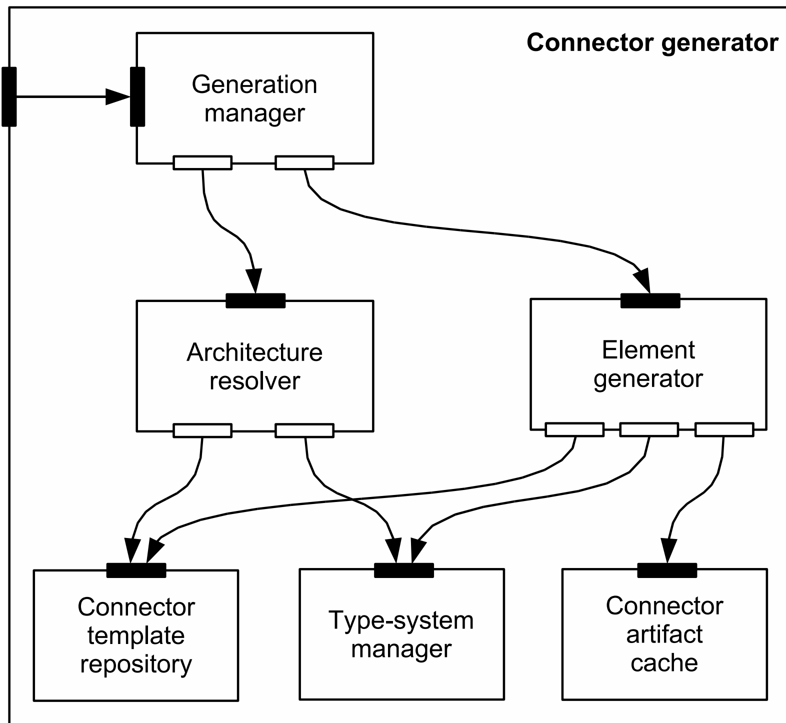

Figure 6: The architecture of the connector generator.

composite element, the resolved element includes also resolved elements of the sub-elements. Thus, the connector configuration in our approach has the form of a set of resolved elements for the units of the connector.

The auxiliary predicates used in the `elem_desc` predicate are in charge of constructing the resolved element. Predicate `elem_decl/3` creates its basic skeleton, predicate `elem_inst/7` chooses and constructs a sub-element, predicates `provided_port/2`, `remote_port/2` create the element ports, and predicates `binding/4` and `binding/5` create delegation and inter-element bindings respectively.

The `elem_desc` predicates are built based on the information kept in repository. To partially abstract from the Prolog language and to capture the element description in more comprehensible way, we use an XML notation which is during resolving transformed to the Prolog predicates. An example of the XML element description is shown in Figure 8. The XML description comprises also parts related to code generation which are use by the element generator. As these are not important for us now, we have omited them in the example (marked by the three dots).

Using the `elem_desc` predicates we are able to build a connector configuration, however, what remains to ensure is that the connector configuration specifies a working connector which respects the connection requirements. We have formulated these concerns as three consistency requirements — a) cooperation consistency, b) environment consistency, and c) connection requirements consistency. They are reflected in the following way.

```
elem_desc(logged_client_unit, rpc_client_unit, Dock, This,
      CurrentCost, NewCost) :-

   elem_decl(This, logged_client_unit, rpc_client_unit),
   cost_incr(CurrentCost, 0, CurrentCost0),
   elem_inst(This, Dock, SE_stub, stub, stub, CurrentCost0,
      CurrentCost1),
   elem_inst(This, Dock, SE_logger, logger, logger,
      CurrentCost1, NewCost),
   provided_port(This, call),
   remote_port(This, line),
   binding(This, call, logger, in),
   binding(This, logger, out, stub, call),
   binding(This, line, stub, line).
```

Figure 7: Prolog predicate describing a sample client_unit implementation

```
<element name="logged_client_unit" type="client_unit" ... >

   <architecture cost="0">
      <inst name="logger" type="logger"/>
      <inst name="stub" type="stub"/>
      <binding port1="call" element2="logger" port2="in"/>
      <binding element1="logger" port1="out" element2="stub"
         port2="call"/>
      <binding element1="stub" port1="line" port2="line"/>
   </architecture>

   ...

</element>
```

Figure 8: XML specification of a sample client_unit implementation — architectural part

Cooperation consistency. It assures that elements in a connector configuration can "understand" one another. We realize this by unifying signatures on two adjoining ports (in predicate `binding`). Recall that the signatures encode not only the actual type but also semantics (e.g., $rmi_server(rmi_iface('CompIface'))$).

Environment consistency. It assures that the element can work in a target environment (i.e., it requires no library which is not present there, etc.). We realize this by testing requirements on environment in `elem_desc` predicate (the description of environment capabilities is in the variable `Dock`).

Connection requirements consistency. It assures that the connector configuration reflects all connection requirements. Since some connection requirements require the whole connector configuration to be verifiable, we do not check them on-the-fly during building the connector configuration, but we verify them once the configuration is complete. We realize the checking by predicates able to test that a given connection requirement (in our case expressed as a name-value

```
nfp_mapping(This, logging, console) :-
   This = element(console_log, _, _, _, _).

nfp_mapping(This, logging, Value) :-
   This = element(logged_client_unit, _, _, _, _),
   get_elem(This, logger, SE_Logger),
   nfp_mapping(SE_Logger, logging, Value).
```

Figure 9: An example of predicates verifying connection requirements for a primitive and a composite element

property) is satisfied by a particular connector configuration. In order to not loose extensibility and maintainability, we compose the predicate from a number of predicates verifying the connection requirement for a particular element implementation. In the case of a primitive element the connection requirement is verified directly. In the case of a composite element, the verification is typically delegated to a sub-element responsible for the functionality related to the connection requirement. If there is no predicate for the connection requirement associated with the element, the verification automatically fails. The example in Figure 9 shows the predicates for checking a "logging" property for a primitive element (console_log in our example) and for a composite element (logged_client_unit in our example).

To select the best configuration we use a simple cost function. We assign a weight to every element implementation reflecting its complexity. The cost of a connector configuration is then computed as a sum of weights of all element implementations in a configuration. We then select the configuration that has the minimal cost. To prune too expensive configurations on-the-fly already during their building, we use the `cost_incr/2` predicate (as shown in Figure 7).

4.2 Code generation

The connector configuration (which is the result of the process described in the previous section) provides us with a selection of element implementations and prescription to which interfaces they should be adapted. The next step in the connector generation is the actual adaptation of elements and generation of so called *element builder* (i.e., a code artifact which instantiates and links elements according to an element architecture) for each composite element.

In our approach, we generate source code and compile it to binary form (e.g., Java bytecode). The exact process of element's code generation is captured by element's specification. An example of such specification is shown in Figure 10. The omited parts correspond to specification of element's architecture and signatures previously shown in Figure 8.

The code generation is specified as a script of actions that have to be performed. In our example the action *jimpl* calls the class `CompositeGenerator`, which creates the source code (i.e., a Java class `LoggedClientUnit`) based on the actual signature it accepts as input parameters. The `CompositeGenerator` works actually as a template expander providing content for tags used in a static part of element's code template (in our case stored in file `compound_default.template` — see Figure 11). By dividing the code template to the static and the

```
<element name="logged_client_unit" ...
      impl-class="LoggedClientUnit">

   ...

   <script>
      <command action="jimpl">
         <param name="generator" value="org.objectweb.dsrg.
            deployment.connector.generator.eadaptor.
            elements.generators.CompositeGenerator"/>
         <param name="class" value="LoggedClientUnit"/>
         <param name="template"
            value="compound_default.template" />
      </command>

      <command action="javac">
         <param name="class" value="LoggedClientUnit"/>
      </command>

      <command action="delete">
         <param name="source" value="LoggedClientUnit"/>
      </command>

   </script>

</element>
```

Figure 10: XML specification of a sample client_unit implementation — code generation part

dynamic part (the template expander), we can reuse one template expander for a number of elements. Moreover, by using inheritance to implement the template expanders and providing abstract base classes for common cases we can keep the amount of work needed to implement a new code template reasonably small.

4.3 Handling of different type-systems

In the context of heterogeneous deployment it is necessary to deal with multiple type-systems simultaneously. By a type-system in this text we mean the way of specifying types and interfaces (e.g. CORBA IDL, Java, etc.) as well as set of types native to a particular programming language (e.g. java.util.Hashmap in Java), component model (e.g. org.objectweb.fractal.Interface in Fractal component model) and middleware (e.g. java.rmi.Remote in RMI). Notice that a type-system also brings different features of interfaces and method signatures, as well as different means of acquiring type information (e.g., using reflection in Fractal, using TIR in SOFA, etc.).

In order to make the generator independent of different type-systems we have separated the functionality regarding types to a separate component (i.e., type system manager). We have identified actions the generator performs on types during connector generation and tailored the type system manager interface

```
package %PACKAGE%;

import org.objectweb.dsrg.deployment.connector.runtime.*;

public class %CLASS% implements ElementLocalServer,
      ElementLocalClient,   ElementRemoteServer,
      ElementRemoteClient {

   protected Element[] subElements;
   protected UnitReferenceBundle[] boundedToRemoteRef;

   public %CLASS%() {
   }

%INIT_METHODS%

}
```

Figure 11: A static part of element's code template — file `compound_default.template`

accordingly. Fortunatelly, all the actions concerning types are not numerous and are well-defined. In fact, apart from passing types around (for which just a super-class or an super-interface for all types is needed) only the following two actions are required from the type system manager.

1. We need to build a type based on a symbolic specification. In other words, we have to implement the type functions (e.g., $java_iface$, rmi_iface, etc.). This comprises the following actions:
   (a) creating a type (e.g., using reflection on a Java interface, getting a type from TIR)
   (b) mapping a type to a different type-system (e.g. CDL interface to Java interface)
   (c) modifying an interface (e.g., modifying the signature of each method in an interface to throw java.rmi.RemoteException)
2. We need to test whether an interface is a sub-type of another interface.

Thus, we have created a simple abstract type model to allow for passing accessing and passing the types in an uniform way (see Figure 12). Abstract types are reified using concrete classes brought by plug-ins for particular type-systems (see Figure 13 showing how Java-types were implemented).

Moreover to allow for item #1 (building types) we implement a type-factory which using type-system plug-ins builds required types based on the symbolic type specification builds the types. The item #2 (sub-type relation) is implemented as a method present in the `InterfaceDef` interface in the abstract type model.

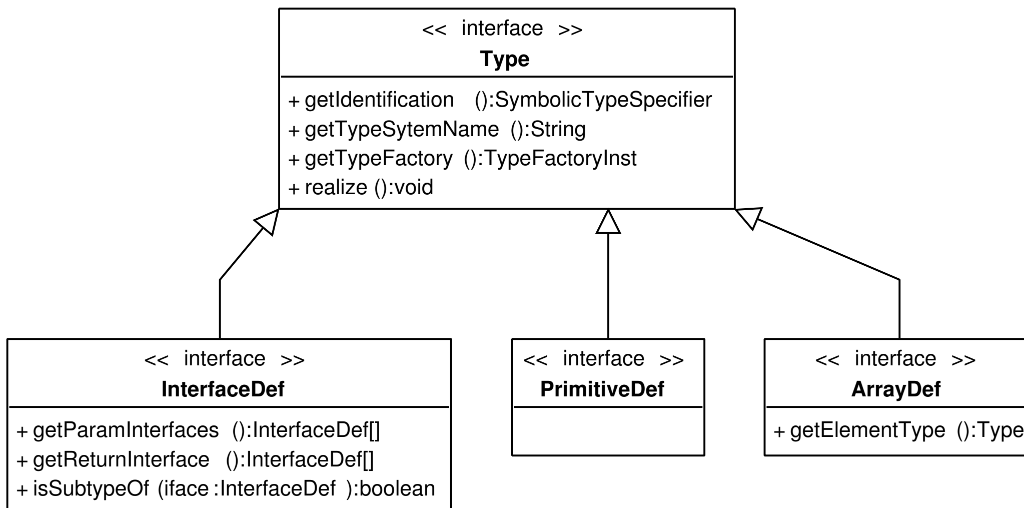

Figure 12: The abstract type model.

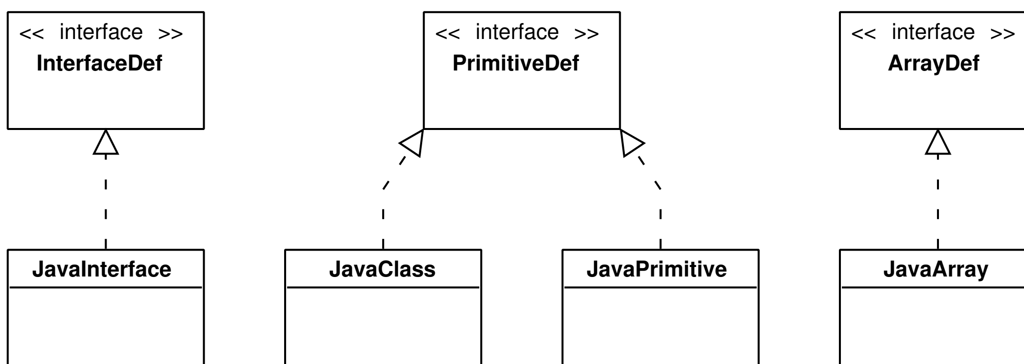

Figure 13: The Java reification of the abstract type model.

5 Related work

To our knowledge there is no work directly related to our approach, however, there are a number of projects partially related at least in some aspects.

Middleware bridging. An industry solution to connect different component models (or rather middleware they use) is to employ middleware bridges (e.g. BEA WebLogic, Borland Janeva, IONA Orbix and Artix, Intrinsyc J-Integra, ObjectWeb DotNetJ, etc.). A middleware bridge is usually a component delegating calls between different component models using different middleware. In this sense, it acts as a special kind of connector. However, middleware bridges are predominantly proprietary and closed solution, allowing to connect just two component models for which a particular bridge was developed and often working in one direction only (e.g. calling .NET from Java).

Modelling connectors. A vast amount of research has been done in this area. There are a number of approaches how to describe and model connectors (e.g., [1], [14], [7]). Save the [1] which is able to synthesize mediators assuring that a resulting system is dead-lock free, the mentioned approaches are not generally concerned with code generation. That is why we have used our own connector model, which has been specifically designed to allow for code generation. Also, we are rather interested in rich functionality than formal proving that a connector has specific properties; thus, at this point we do not associate any formal behavior with a connector.

Configurable middleware. In fact connectors provide configurable communication layer. From this point of view, the reflective middleware (e.g., OpenORB [2]) being built from components is a very similar approach to our. The main distinction between the reflective middleware and connectors is the level of abstraction. While the reflective middleware deals with low-level communication services and provides a middleware API, connectors stand above the middleware layer. They aim at interconnecting components in a unified way and for this task transparently use and configure a middleware (e.g., OpenORB).

6 Conclusion and future work

In this paper we have shown how to synthesize connectors for the OMG D&C-based heterogeneous deployment. We have used our experience from building the connector generator for SOFA. However, compared to SOFA, where the generator was only semi-automatic and homogeneous, we had to cope with the heterogeneity (which caused the co-existence of different type-systems) and fully automatic generation based on target environment and high-level connection requirements.

We have created a prototype implementation of the connector generator in Java with an embedded Prolog [6] for resolving connector configurations. Currently we are integrating the generator with other our tools for heterogeneous deployment. As for the future work we would like to concentrate on automatic handling of incompatibilities between different component systems (i.e., life-cycle incompatibilities and type incompatibilities).

References

[1] M. Autili, P. Inverardi, M. Tivoli, D. Garlan, "Synthesis of "correct" adaptors for protocol enhancement in component based systems", Proceedings of SAVCBS'04 Workshop at FSE 2004. Newport Beach, USA, Oct 2004

[2] G. S. Blair, G. Coulson, P. Grace, "Research directions in reflective middleware: the Lancaster experience" Proceedings of the 3rd Workshop on Adaptive and Reflective Middleware, Toronto, Ontario, Canada, Oct 2004.

[3] D. Bálek and F. Plášil, "Software Connectors and Their Role in Component Deployment", Proceedings of DAIS'01, Krakow, Kluwer, Sep 2001

[4] L. Bulej and T. Bureš, "A Connector Model Suitable for Automatic Generation of Connectors", Tech. Report No. 2003/1, Dep. of SW Engineering, Charles University, Prague, Jan 2003

[5] L. Bulej and T. Bureš, "Using Connectors for Deployment of Heterogeneous Applications in the Context of OMG D&C Specification", accepted for publication in proceedings of the INTEROP-ESA 2005 conference, Geneva, Switzerland, Feb 2005

[6] E. Denti, A. Omicini, A. Ricci, "tuProlog: A Light-weight Prolog for Internet Applications and Infrastructures", Practical Aspects of Declarative Languages, 3rd International Symposium (PADL'01), Las Vegas, NV, USA, Mar 2001, Proceedings. LNCS 1990, Springer-Verlag, 2001.

[7] J. L. Fiadeiro, A. Lopes, M. Wermelinger, "A Mathematical Semantics for Architectural Connectors", In Generic Programming, pp. 178-221, LNCS 2793, Springer-Verlag, 2003

[8] O. Gálik and T. Bureš, "Generating Connectors for Heterogeneous Deployment", accepted for publication in proceedings of the SEM 2005 workshop, Jul 2005

[9] P. Hnětynka, "Making Deployment of Distributed Component-based Software Unified", Proceedings of CSSE 2004 (part of ASE 2004), Linz, Austria, Austrian Computer Society, ISBN 3-85403-180-7, pp. 157-161, Sep 2004

[10] N. Medvidovic, N. Mehta, M. Mikic-Rakic, "A Family of Software Architecture Implementation Frameworks", Proceedings of the 3rd IFIP Working International Conference on Software Architectures, 2002

[11] ObjectWeb Consortium, Fractal Component Model, http://fractal.objectweb.org, 2004

[12] ObjectWeb Consortium, SOFA Component Model, http://sofa.objectweb.org, 2004

[13] Object Management Group, "Deployment and Configuration of Component-based Distributed Applications Specification", http://www.omg.org/docs/ptc/03-07-02.pdf, Jun 2003

[14] B. Spitznagel and D. Garlan, "A Compositional Formalization of Connector Wrappers", Proceedings of ICSE'03, Portland, USA, May 2003