Using DSL for Automatic Generation of Software Connectors

Tomáš Bureš[1,2], Michal Malohlava[2], Petr Hnětynka[2]

[1]Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod Vodarenskou vezi 2
Prague 8, 18207, Czech Republic

[2]Department of Software Engineering
Faculty of Mathematics and Physics
Charles University, Malostranske namesti 25
Prague 1, 11800, Czech Republic

{*bures,malohlava,hnetynka*}*@dsrg.mff.cuni.cz*

Abstract

Component-based engineering is a recognized paradigm, which models an application as a collection of reusable components. The key idea behind components is that they contain only the business logic and communicate with one another only via well-defined interfaces. The communication paths among components (so called bindings) are in modern component systems realized by software connectors, which allow explicit modeling of communication and also its implementation at runtime. An important aspect of using connectors is the possibility of their automatic generation, which saves a significant amount of development work. However, the generation itself is not a trivial task, since there is a big semantic gap between the abstract specification of a connector at design time and its implementation at runtime. In this paper, we present an approach to generating implementations of software connectors. The approach is based on a new domain specific language for describing templates of connector implementations and a transformation framework using the Stratego/XT term rewriting system for generating source code of connectors.

1. Introduction

In recent years, software components and component-based software engineering (CBSE) have been recognized as a solid abstraction and a methodology for reducing complexity of software applications. CBSE brings a higher level of order to software by introducing a tangible concept of software components, which act as reusable blocks providing a specified functionality and implementing the business logic of a component application.

A component is an encapsulated entity, which is allowed to communicate with other components only via its well defined interfaces (both *provided* and *required*). A composition of components is defined by an *architecture* [17], which specifies component instances and *bindings* among their interfaces.

With the growing complexity of component applications and with the need to run them distributively, there has appeared another significant aspect in CBSE, which is modeling the interaction among components. For this task, another first-class primitive called *software connector* has been introduced.

A connector is a reusable entity that models and implements a binding among component interfaces. It is inherently distributed; it consists of a number of *connector units*, with each unit connected to a particular component interface.

At design time, connectors provide a way to express high-level requirements on component interactions. This typically comprises the communication style (e.g. synchronous method invocation, asynchronous message delivery, shared memory, streaming, etc.) and a number of non-functional requirements (e.g. requirements on security, QoS, monitoring, distribution, etc.).

As each particular inter-component binding in a component application is realized by a specific instance of a connector, it is possible to model each binding separately. This allows for example for employing security only in a part of an application or for using different communication styles [19] choosing the most appropriate one for each inter-component link.

At runtime, connectors are used to actually implement the prescribed interaction. Here, middleware is typically employed, but its use is encapsulated by a connector so that components do not have to deal with the middleware directly. Since connectors are present in distinct instances at runtime, it is possible to use different middlewares in one application, and to easily implement non-functional features (e.g. to check that communication complies to a particular protocol [11], to collect performance statistics for each particular binding [13]).

In the context of the non-functional features, it is also important that connectors can be introspected and reconfigured at runtime. This allows for enabling and disabling a particular feature (e.g. collecting performance data) at runtime without causing static overhead when the particular feature is disabled [13].

1.1. Motivation example

In today's world, there are many cases where the demand for connecting multiple computers exists. A commonly used application architecture is shown on Figure 1. It displays a bank application distributed on 3 computers. On each computer one part of the application is running. The central part is a *bank server*, which stores bank clients data, and it is located in a secure environment of an intranet. Other two parts serve as bank operators, which manage data stored at the bank server. They are connected with the server via bindings *L1* and *L2*. The *bank operator 1* lies in the same safe environment as the bank server, thus there are no special requirements on a binding *L1*. On the other hand, *bank operator 2* is located in an external network somewhere in Internet. Therefore, the binding *L2* has to be secure to avoid eavesdropping. There could be other extra requirements for bindings — e.g. QoS or monitoring of operator transactions.

Several possible ways of building such parametrized bindings exist. One way is to create each binding separately by hand, but this solution is not scalable in the case when it is necessary to connect higher number of computers.

Another method of binding representation and implementation is employing software connectors. But the main problem with the current state of the art in the area of software connectors is that there is no comprehensive approach, which would combine freely designed user-defined connectors with the possibility of generating their implementation.

Without a seamless generation, the use of connectors brings an extra work connected with their implementation, which significantly effaces their potential benefits.

Additionally, it becomes evident that if the connector implementation can be generated fully automatically, it is possible to postpone the generation to the deployment time, which is the last stage of the development life-cycle before an application is actually launched. When creating connectors at deployment time, it is possible to have complete knowledge of an application architecture, its distribution to particular computers (*deployment docks*) and capabilities of the deployment docks. This information can be then utilized to create a highly optimized connector implementation that is tailored specifically to the particular deployment scenario.

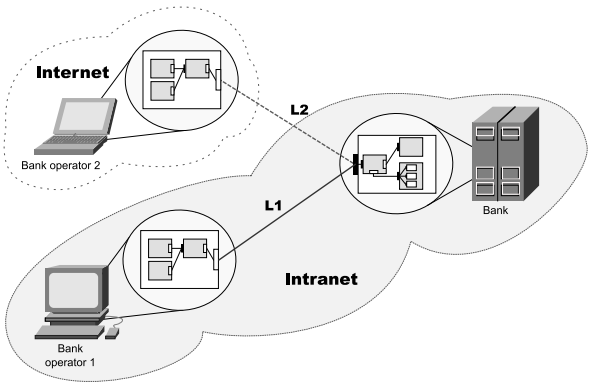

Figure 1. An example of real-life application

1.2. Goals and structure of the paper

With regard to the previously stated problems, we focus on a process of the generation connector implementation and put it to the context of the large project mainly presented in [19], which covers the problem of software connectors completely from their resolving to deployment. Concretely, our goal has been to create a comprehensive approach which would allow us to 1) specify a general implementation of connectors and 2) would automatically generate a connector implementation from its high-level specification and a general description of the implementation.

The paper is structured as follows. In Section 2, we present an overview of the connector building process. The following Section 3 focuses on a domain specific language ElLang for describing implementation of connectors and the generation of the connector code. The related work is discussed in Section 4, while Section 5 concludes the paper.

2. Building the connector code

This section describes the step-by-step process of building connector code (as proposed in [19]) for the example described in Section 1.1. For simplicity, we show a process of generation of the binding *L2*, because the other bindings are generated in the same way.

The first step of the connector generation is modeling application with help of components and connectors. The part of the application, which represents the bank operator 2 and the bank server, is shown on Figure 2. It shows two components and the binding between them represented by a connector. The first component is called a *bank server component* and lies in a *server deployment dock*. The second one represents the bank operator 2. Both components are connected with help of the connector standing for the binding *L2*.

The application designer specifies parameters of the connection and requirements laid on deployment docks. This

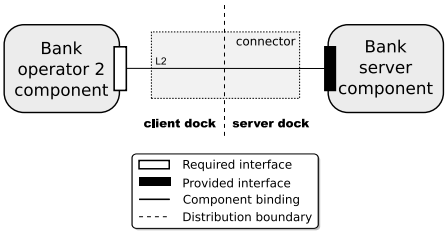

Figure 2. Real-life application from component developer point of view

information is important for a deployment of components onto described docks, connecting them and launching a whole application. It is specified at design time. For the purpose of preparing connectors a *connector generator* is called (see Figure 4) and a high-level connector specification (see Listing 1) is passed to it. The specification determines requirements that have to be satisfied by the generated connector. The specification is automatically generated from docks and binding parameters specified at design time, and additional requirements known at development or deployment time (e.g. restrictions of a run-time environment).

Listing 1 shows an example of a high-level connector specification used in our approach. It specifies a connector for an interface ServiceItf, which uses the method invocation as the communication style and enforces logging and the secure connection over SSL on the client side (i.e. *client connector unit*) and serialization of calls on the server side (i.e. *server connector unit*).

```
unit "client_unit" {
  dock "client_dock";

  provided port "call" signature "ServiceItf";

  nfp "communication_style"
      value "method_invocation";
  nfp "secure_protocol" value "SSL";
  nfp "logging" value "console";
};

unit "server_unit" {
  dock "server_dock";

  required port "call" signature "ServiceItf";

  nfp "communication_style"
      value "method_invocation";
  nfp determines "call_serialization";
};
```

Listing 1. An example of a high-level specification

The high-level connector specification is not directly used for generating connector code, because there is a big semantic gap [19] between the high-level feature-based specification (in Listing 1) and the executable connector code. It is rather difficult to cross this gap directly. Thus, we have narrowed the gap by introducing an intermediary connector description called the *low-level connector configuration* (see Figure 3). It views a connector in a component-based way as a composition of *connector elements* which are basic building blocks of connectors. Elements may be either *primitive* or *nested*. Primitive elements are associated with code that implements them. Nested elements have an internal structure consisting of sub-elements and bindings among them. A low-level connector configuration can be thus seen as a tree with inner nodes formed by nested elements and with leaves represented by primitive elements. The elements located on the first level of nesting are called *connector units* and each of them lies in a different deployment dock. Therefore, remote bindings are used only between connector units. Elements inside one unit are connected with local bindings.

Elements communicate only via designated *ports*, which can be either *local* or *remote*. The local ports are further divided to *provided* and *required* (to modeling control flow). In the case of remote ports, the control flow is not captured on the level of the low-level connector configuration.

Each port has an associated signature, which is a term describing the element's interface. This is because some of the features introduced by the connector may internally change the format of transmitted data, change method signatures, order of calls, etc. The signatures attached to an element are then used for its adaptation. An example of the signatures can be found in Figure 3. There is a signature assigned to each element port — java_iface indicates a Java interface, rmi denotes accessibility via RMI (technically it modifies the Java interface by adding java.rmi.Remote to its parents), etc.

The low-level connector configuration is automatically generated by a *connector architecture resolver*. The generation is based on an input high-level connector specification and additional information about a connector model. The process creates a connector architecture with a nested hierarchy of elements and assigns resolved signatures to all ports in the architecture in accordance with prescribed communication requirements. The goal is that the connector modeled by the low-level configuration must be able to run on a target deployment dock.

Once the low-level configuration is ready, what remains from the generation is actual building of the connector code. In our approach, we build the connector implementation element-wise. This means that each of the elements in the configuration is built separately and the elements are connected together when instantiating the connector at runtime.

We have chosen this approach for simplicity, because it keeps the relation $1 : n$ between elements and files containing their implementation. However, in the future we would like to support also the relation $n : m$ (i.e. a number of ele-

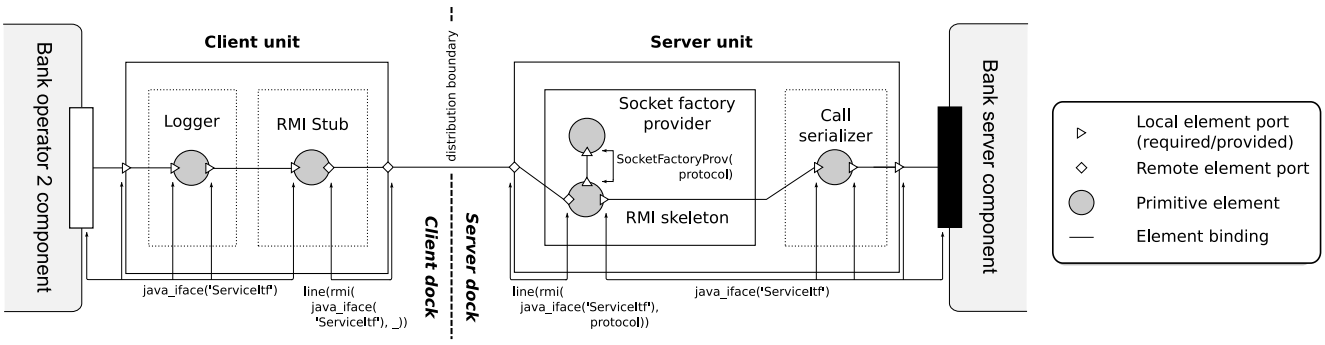

Figure 3. An example of a graphical representation of low-level connector configuration

ments may result, for example, into one Java class), which may obviously yield more optimized code.

Building of the element implementation is done in a number of steps. They typically include creation of the source code (i.e. the adaptation of the source code template associated with the element), building interfaces used for communication between elements, compiling the code, and packaging the code. In the case of building middleware stubs and skeletons, there is typically another step presented, in which a particular generator of stubs and skeletons is called (e.g. RMI compiler, IDL compiler, etc.).

Since this list of steps is obviously not complete and differs from an element to an element, we associate with each element not only a source code template, but also a build-script which specifies what steps to perform when creating the element implementation. These steps we call *actions* in our generator and they can be introduced to the generator in the form of plugins.

An example of a build-script is given in Listing 2. The script performs three steps. It starts with the generation of source code (the action *jimpl*). Then it compiles the source code (the action *javac*) and deletes generated source files (the action *delete*). The run of the script is concluded by packaging the code, which is common for all connector elements, and thus it is not stated explicitly in the script.

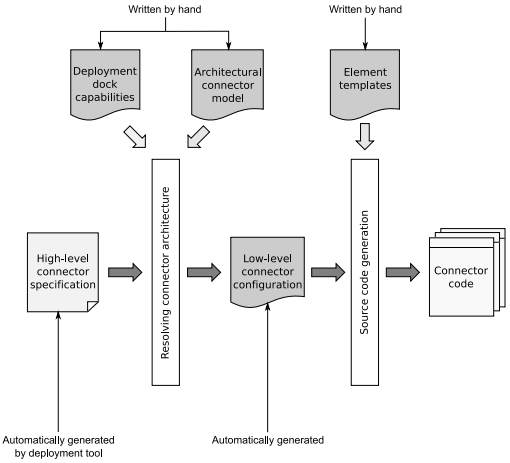

Figure 4. Generation process with inputs

```
<script>
  <command action="jimpl">
    <param name="generator"
        value="org.....StrategoGenerator"/>
    <param name="class"
      value="LoggedClientUnit"/>
    <param name="template"
        value="compound_default.ellang" />
  </command>
  <command action="javac">
    <param name="class"
      value="LoggedClientUnit"/>
  </command>
  <command action="delete">
    <param name="source"
        value="LoggedClientUnit"/>
  </command>
</script>
```

Listing 2. An example of build script

Probably the most important action present almost in every script is the generation of source code. This action belongs to those parts of the generation that have been significantly evolving since we have started with the problem of the automatic connector generation. It is by itself a non-trivial task that contains issues such as how to specify the source code templates, how to actually perform the adaptation, etc. To deal with the issues we have designed a special domain specific language (ElLang) for describing element code templates. The templates are processed by a term rewriting engine and source code is generated. Section 3 describes the ElLang language and its processing in detail.

3. Generation of code

As mentioned in the previous section, the structure of a connector (or rather a connector element) is described by a low-level connector configuration. The configuration specifies sub-elements, signatures of their ports, and bindings, however, it does not include the connector code. The code

is defined separately in the form of element code templates. Each template contains target code and by combining the templates the connector implementation is assembled.

The templates are generic with regard to business interfaces a connector mediates. This means that the templates have to be adapted to particular sub-element signatures. Moreover, the combination and instantiation of templates has to comply to the connector structure as specified by the low-level connector configuration. In more general terms, there are issues related to the format of the templates and the way they should be processed to get the resulting connector implementation.

There are several possible ways of addressing the issues. One extreme is to realize the template as a program in a general programming language (e.g. as a Java class) which outputs target code depending on the low-level configuration. Obviously, in this case it is relatively straightforward how to process the low-level configuration and output the code accordingly. But since the target code is in this approach encoded in the constructs of the general programming languages, it is very difficult to develop and debug the templates (especially when they are more complex).

Another extreme is to use of an existing template language (like JET, Velocity or PHP). These languages allow putting special tags into a textual document. When processing the document they evaluate the tags and leave the rest of the document unchanged. This brings quite good readability and comprehensibility of the templates. However, there are problems with adapting templates according to the low-level configuration. Especially, simpler template languages (e.g. JET and Velocity) lack the expressive power to reasonably describe the adaptations of templates to business interfaces. Moreover, as the languages do not know the grammar of the target code, they can easily produce syntactically incorrect code.

In our approach, we have chosen a trade-off between these two extremes. We have designed a dedicated domain specific language (DSL) for defining element code templates. The language contains special constructs for describing the kind of code generality that is found in connectors (i.e. generality in respect to component business interfaces). Thus, we have effectively drawn the advantages from both the previously discussed extremes. Templates are at least as well readable and comprehensible as in the case of other template languages (such as JET, Velocity), and they support the adaptation, which is in this case performed by the DSL processor.

After having decided to use a dedicated DSL and a DSL processor there still remains an issue how to address the support of different target languages (e.g. Java, C++, C#) for connector implementation, which is necessary to achieve the heterogeneity of the generator. There are actually several ways to achieve the task. The key difference lies in what parts of the DSL and the DSL processor remain independent of the target language and what are dependent.

The approach where DSL processor is kept independent of the target language and only the template language differs corresponds to the general purpose template languages (e.g. JET, Velocity, PHP, etc.). They use special groups of characters (e.g. tags <?php and ?> in the case of PHP) to delimit constructs that the processor can handle. The rest of the template (i.e. target language code) is used verbatim as it is not understood by the processor. The main problem of this approach is that it allows the generation of syntactically incorrect code, which in our experience happens quite often.

The opposite approach where the whole DSL is kept independent of a target language, basically leads to creating a new abstract programming language that is translated to a particular target language. Apart from meta-programming constructs, this new language has to incorporate also programming constructs from a target language. The dependence on a target language is concentrated in the DSL processor, which as a result gets very complex and different for every target programming language. This approach is even more complicated in the area of software connectors. There, low-level system dependent features are often utilized and they would have to be included in the abstract language as well.

A reasonable trade-off, however, lies in creating a hybrid solution where a part of the DSL and a part of the DSL processors are target language dependent while the rest remains independent. In our solution we have taken this way.

Generally, we have followed the MDA paradigm and designed the template DSL (called ElLang) as independent of the target language (it corresponds to PIM in MDA). It contains basic meta-programming constructs as described in Section 3.1. This language is embedded into a particular target language by merging their grammars using MetaBorg method [14]. This way a platform specific template language arises (e.g. ElLang-J in the case of Java as the target language), which corresponds to PSM in MDA.

The DSL processor in this approach is basically independent of the target language. The only dependent parts are a grammar definition and a definition for a pretty-printer. However, the middle part of the DSL processor (which performs the transformation) is mainly target language independent. An important benefit of this approach is also that it ensures syntactical correctness of the generated code (thanks to the knowledge of the target language grammar).

3.1. Overview of the ElLang language

In our approach we have introduced a domain specific language as a mixture of an abstract meta-language ElLang and a target language for the connector implementation (e.g. Java, C++, C#). The base language ElLang provides a set

of basic meta-statements (if, set, foreach, etc.) and a few special constructs which are specific for purpose of the connector element generation.

The basic meta-statements operate with meta-variables and they are used for code modification and generation. The meta-variables are important for storing useful generation information — e.g. indexes or names of generated elements. The notation used for working with meta-variables is similar to JSP or Velocity (i.e. ${a}).

A meta-variable can store either a primitive type (a number or a string) or it can be an associative array: ${elIndex[ELEMENT.name]}. The latter case is essential when it is necessary to remember an index of a generated sub-element based on the sub-element's name.

ElLang also provides a special group of read-only meta-variables called *queries*. These meta-variables are used for accessing information about the low-level connector configuration. The configuration is a hierarchical structure and thus we use a dot notation to navigate in it. For example, for reading value stored in sub-node name, which is a child of the root node class, the query ${class.name} is used.

These *queries* also provide a counting operator and restrictions. The counting operator is useful in cases when it is needed to know the number of nodes with given name (e.g. for allocating an array).

For example, the query ${elements.element#count} counts the number of sub-nodes with the name element of the parent node elements. The restriction operator limits a result set of a query in according to a defined condition — e.g. ${elements.element(name=stub)} returns all elements, which have the sub-node name with the value stub. All these meta-variables and queries can be used with basic meta-commands.

The assignment is realized using the meta-command set, which modifies value of the meta-variables: $set a = 1$.

The control flow of the generation is controlled by meta-commands if −else, foreach, rforeach.

Statements if and foreach operate in the same way like in others programming languages. The meta-command rforeach (i.e. *recursive foreach*) serves for creating nested grammar constructs. For example:

```
$rforeach(PORT in ${ports.port(type=PROVIDED)} )$
  if ("${PORT.name}".equals(portName)) {
    /* ... */
  } else $recpoint$
$final$
  /* ... */
$end$
```

The command repeats the body of the cycle (statements between meta-commands rforeach and final), but instead of concatenating bodies it expands the next body at the place of the meta-statement recpoint. The final section enclosed by meta-statements final and end substitutes the command recpoint in the body of the last cycle. This construct is especially useful when creating nested if-then-else blocks.

The last command of basic meta-commands is the import statement. This command allows dividing the element template code into multiple files and sharing these partial templates among several element templates.

The template language also provides a set of special meta-statements which have been specially designed for the connector generation. The first group of special meta-commands creates support for a template inheritance. A template has a possibility to extend another template and reuse its code.

In the example below, the template console_log extends the template primitive_default . It means that the child template inherits all the definitions of the parent template.

```
element console_log
  extends primitive_default {
   /* ... */
}
```

The template inheritance simplifies writing of connector templates and supports code reuse, because most templates typically share a common base and differ only in small parts.

Another extension mechanism closely tied with the template inheritance are extension points. They define named locations in the parent template, which may be overridden in the child template. This allows inserting new code or overriding existing code at specified places.

```
element primitive_default {
  public ${classname}() {
    $extPoint(beforeCtor)$
      /* default code, may be empty */
    $end$
  }
}

element console_log
  extends primitive_default {
  $defineExtPoint(beforeCtor)$
  /*
   * new code to override
   * the default one
   */
  $end$
}
```

The template inheritance addresses in a sense the same problem as the import statement. However, these two methods of code reuse are complementary, each having its own use cases as well as issues which it cannot easily address.

Another group of meta-commands specially designed for the connector generation are statements for defining method templates. A method template is generic code that is instantiated for every method, which is defined in an element port signature. This creates a direct support for the main sort of generality found in connector elements, which is the generality in the business interfaces a connector mediates.

The example below shows a method template for methods of element's in port. Technically, this means that the DSL processor will produce code implementing each method of the in port and the implementation will result from instantiating the method template. To get to the signature of the method being processed, the template uses a special meta-variable method.

```
implements interface
  ${ports.port(name=in).signature} {

  method template {
    ${method.declareReturnValue}
    /* ... */
    $if (method.returnVar) $
    ${method.returnVar} =
      this.target.${method.name}(
      ${method.variables});
    $else$
      this.target.${method.name}(
      ${method.variables});
    $end$
    // generates return statement
  // if it is needed
   ${method.returnStm}
 }
}
```

Listing 3 shows code of a compound connector element. The element provides initialization of subelements and connects them (method $initializeArchitecture$). It also detects remote ports and prepares an array for storing remote references. Then the element has to implement set of interfaces for local/remote server/client. Each of interfaces provides a set of method which manipulates with subelements and their ports.

```
package ${package};

element compound_default {
  protected Element[] subElements;
  protected RemoteRefBundle[] remoteTargetRefs;
  protected final ConnectorUnit parentUnit;
  /* Constructor */
  public ${classname}(ConnectorUnit parentUnit,
                      boolean isTopLevel)
              initializeArchitecture ();
  }

  void initializeArchitecture() {
   subElements =
     new Element[${elements.element#count}];

   try {
    $set i = 0$
    $foreach(ELEMENT in ${elements.element})$
     subElements[${i}] =
    new ${ELEMENT.class}(parentUnit, false);
     $set el[ELEMENT.name] = i$
     $set i = i + 1$
    $end$

    /* create bindings */
    $foreach(BINDING in ${bindings.binding})$
     $if (BINDING.type == "BINDING") $
      ((ElementLocalClient)
      subElements[${el[BINDING.from.element.name]}])
          .bindElPort( "${BINDING.from.port}",
           ((ElementLocalServer)
            subElements[${el[BINDING.to.element.name]}])
           .lookupElPort("${BINDING.to.port}"));
     $end$
    $end$
   } catch (Exception e) {
     throw new ElementLinkException(e);
   }

   $set i = 0$
   $foreach(REMOTE_PORT in ${ports.port(type=REMOTE)})$
    remoteTargetRefs[${i}] = null;
    $set ref[REMOTE_PORT.name] = i$
    $set i = i + 1$
   $end$
  }

  /* ElementLocalServer interface implementation */
  implements interface ElementLocalServer {

    public Object lookupElPort(String portName) {
     $rforeach(PORT in ${ports.port(type=PROVIDED)} )$
       if ("${PORT.name}".equals(portName)) {
         /* ... */
       } else $recpoint$
       $final$
        throw new ElementLinkException();
      $end$
    }
  }

  implements interface ElementLocalClient {
    /* ... */
  }

  /* ... */
}
```

Listing 3. An example of compound element for client and server unit

3.2. Creating connector sources from ElLang

In this section, we describe the DSL processor we have created for processing the ElLang language. Its task is to produce element source code in a target programming language from the low-level configuration of an element and related code templates.

We have implemented the DSL processor in Stratego/XT [8], which is a tool set containing programs for the generation of parsers and pretty printers, for abstract syntax tree transformations, etc. The entire Stratego/XT package is based on the Syntax Definition Formalism (SDF), the Generic Pretty-Printing (GPP) and the Stratego language. The Stratego language is a special purpose transformation language based on term rewriting and strategic programming paradigm.

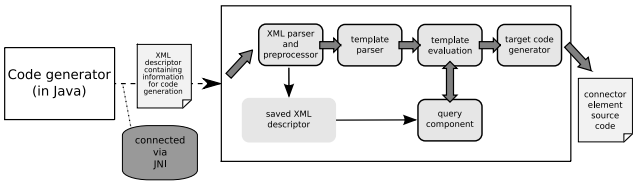

Figure 5. The architecture of the proposed generator

We have defined the grammar of ElLang in SDF and we have used MetaBorg method [14] to join the grammar of ElLang with the grammar of a particular programming language. Here, we have benefited from the fact that SDF grammars and even GPP pretty printer definitions are available for major programming languages (including Java).

The DSL processor has been formed as a chain of processes as depicted in Figure 5. It is divided into two parts. The first part is implemented in Java and provides bridging between the architecture resolver and the Stratego part of the generator. The second part is a code generator implemented in the Stratego language and it is responsible for generating target code of a connector element based on the low-level configuration and element templates.

Communication between the Java and the Stratego part is performed via Java Native Interface (JNI), which is a way to access native libraries from Java and vice versa.

There are a lot of information being exchanged between the Stratego and the Java part. In our implementation, we have used XML to encode them. The information comprise:

- low-level configuration of an element to be generated along with description of sub-elements (in the case of a composite element);
- name of a template providing the element implementation (the template may additionally import or extend other templates);
- target package name, class name, working directory, etc.
- user-defined parameters customizing the generated implementation (e.g. a default file name to which a *logger* element is to log collected data).

When looking on the Stratego part in more detail. The chain starts with an XML parser that processes XML containing the information listed above and stores it in memory for later use. Subsequently, it invokes a template parser to parse the file containing the element code template.

The parsed template is transformed by the template evaluation process. During this process all meta-variables and meta-commands are gradually evaluated and replaced by constructs of the target programming language.

The resulting abstract syntax tree containing only constructs of the target language is then passed to the pretty printer. The pretty printer gives a concrete syntax to the abstract syntax tree and writes out the resulting element implementation to a file.

The template evaluation process often needs to access data in the low-level configuration. It references them using the queries explained in Section 3.1. The queries are evaluates by the query component, which traverses the stored low-level configuration and evaluates the query.

The fine-grained division of the DSL processor to a chain of processes, which we have shown above, lowers its overall complexity. Moreover, it has allowed making some of the processes completely independent of the target language (the XML parser and preprocessor, the query module or the best part of the template evaluation module).

4. Related work

There are different approaches to connectors (e.g. Wright [3], Unicon [15], C2 [16], Fractal [6], etc.). However, none of those strives to combine sufficient freedom in a connector design together with the automatic connector generation. Thus, we list in this section the most related works in other domains and in the code generation in general.

4.1. Methods of program synthesis

Our code generator implemented in Stratego takes as an input an abstract descriptor of a connector element. The description is transformed into source code with help of templates. The idea of a program synthesis from an abstract description is not new and it helps reduce development time, errors and maintenance.

One representative is *AutoBayes* [2] — a fully automatic high level generator system developed in NASA which produces data analysis programs from statistical models. The model specifies statistical properties, their conditions and probabilities. From this description, AutoBayes generates optimized C++ code which can be then used in Matlab programs. A generation process is guided by a general algorithm schemes which include program fragments with open slots and constraints checked against a statistical model during the code generation. The open slots are filled in according to the input model and described schema constraints. The generator can be easily extended with new control schema without modifying the kernel of the system.

Another program synthesis tool is *AutoFilter* [12] as well implemented by Robust Software Engineering group at NASA Ames. It deals with the generation of programs that solves estimation problems (e.g. computing an approximate attitude or a velocity vector) with special methods generally called Kalman filters. A given high-level model of a problem described in a form of linear or differential equations is automatically transformed into C/C++ (or Modula II) code, which can compute estimations in according to the described model. The code generation is also schema based, where each schema represents some well-known algorithm which solves an input problem. The algorithm is described by a simple template programming language. An advantage of the generation system is pluggable *support modules* that produce target code.

Both systems mentioned above represent an idea of program synthesis from a high level description, which is close

to our system. They contain well-designed part of the target code generation that supports pluggable modules. On the other hand, each of them implements a solution of a domain specific problem that cannot be simply adapted to the general connector generation.

Another approach of synthesizing programs is to reuse existing code fragments and build them together in accordance with a high-level description. The solution proposed by the paper is based on this idea. It adapts code templates for different elements and brings them together. Our solution is not generic and is close to the connector generation. However, the general applications of this principle exist:

Feature oriented programming (FOP) is a paradigm for applications synthesis, analysis and optimization. The main idea is to build programs (and also classes) incrementally by composing features. Features are seen as basic building blocks of applications and feature characteristics are used to distinguishing programs within a family of related programs. These families are called software product lines [5]. The model of a product line architecture has three basic ideas — (a) identifying the similar set of features in a family, (b) implementing each feature in one or more ways and (c) defining specific applications of product lines by the set of features that it supports and their implementations.

There are several implementations of feature refinements — one of them is *AHEAD (Algebraic Hierarchical Equations for Application Design)* [4] which is based on stepwise refinement. This system considers the feature as the primary unit of software modularity. In this concept the feature is not only source code (e.g. class), but also another hierarchical program representation (e.g. makefile, documentation, UML model, etc.). From AHEAD's theoretical point of view features are algebraic operators and programs can be created by composition of such operators.

The idea of composing features is related to building connector from elements. Each feature would be associated with some type of element (e.g. log element has logging feature) and the whole connector would be built as a product line of specified elements. But currently FOP is only academic concept which is not widely accepted.

Aspect oriented programming [9] is a method which helps programmers to separate concerns. The AOP is primary focused on cross-cutting concerns. These concerns appear across many modules in a program (typical example is logging which is tangled and scattered in code). This is the contrast [18] with FOP which separates program features which are composing the resulting application. With AOP programmers instead of implementing functionality into an object they write a point-cut which *weaves* the aspects into objects. Hence, objects can contain the basic logic which are not tangled with uninteresting code.

AOP identifies well-defined points in program flow (e.g. method execution, method call, field get, etc.), which are called *join-points*. Execution at a join-point can be modified by an *aspect*. The aspect encapsulates information which join-points affects it and how. Thus, it introduces a *point-cut* — a set of join-points in which the given aspect is interested. And in addition the aspect has to specify what happens at the join-points. The piece of the code which is executed at the specified point-cut is called an *advice*. This code can access variables which are visible at given point-cut (e.g. class variables).

The leading Java implementation of this paradigm is *AspectJ* [10]. It provides all features of AOP described above. Although AOP is currently widely used even in a commercial environment, it is not really usable for the connector element generation based on the concept of the existing generator [19]. This is because AOP just affects predefined parts of code, but we need to adapt and modify whole element template as a solid unit.

4.2. Transformation languages

The proposed generator transforms a template into a target language with help of the Stratego language. Transformations are implemented by term rewriting strategies, but it is not only the sole approach, which can be chosen.

One favorite method of a transformation is using extensible stylesheet language transformations (XSLT [22]). It belongs among XML-based languages used for the transformation of XML documents. It is most often used for converting XML documents into web pages or PDF documents but it can be also used for the code generation. An input XML document serves as an input template which is processed by a XSLT processor. The processor is driven by a XSLT stylesheet file which describes transformation over the input document (represented as XML tree). Because an input template is in an XML language there is no need to define a grammar of the template. The main disadvantage of this approach is not-well readable language for defining template transformations (because it is based on XML) which is not really suited for developing larger programs.

Another special group of transformation languages is composed of template languages. Each provides a set of meta-commands which control the generation of output code (e.g. HTML, XML, RTF). These languages are often designed as general purpose, but favorite target domain, where they are used, is the generation of web pages. The most known are Velocity [1] or FreeMarker [20], which offer accessing Java variables from a simple template language. Next representative is JET project, which template language is Java-based and is similar to proposed language in this thesis. Java Emitter Templates (JET) [7] is a part of *Eclipse Modeling Framework*. It realizes implementation of the code generation from an abstract model. The syntax of *JET* templates is based on JSP (Java Server Pages).

Hence, the template provides the Java based meta language to access Java variables. Variables can be passed into the template via a template expander class and they can be easy used inside template code. The template itself is translated into a Java class during the evaluation stage. The class has a method generate which produces a result string (containing evaluated template). The advantage of this approach is easily comprehensible template language based on Java.

The main problem of the JET and generally of all template engines is, however, the grammar of the template. The engines do not know the target language, so they cannot define a complete grammar of a template language (they can only define the grammar of a meta-language). This is the main difference between our solution and the general template engines, because the proposed template language has the grammar based on the grammar of the target language (Java) and the meta-language (*ElLang*). Thus all syntax errors in a template can be found during parsing stage, which makes writing templates more comfortable and error proof.

5. Conclusion

In this paper, we have presented an approach to the automatic generation of connector implementation. The approach is based on the newly designed DSL (called ElLang) for writing templates of connector implementations. The DSL is accompanied by the transformation system which produces connector source code based on an automatically generated low-level configuration of a connector and its code template written in ElLang.

We have developed the ElLang language specifically for defining connector code templates and we have equipped it with special constructs addressing the connector generation. We have designed the language as independent of a target programming language, which has been an important requirement brought by the goal of supporting different component systems. Specialization of the ElLang language for a particular programming language is done in our approach by joining grammars of the two respective languages, which we have demonstrated on ElLang-J (a specialization of ElLang language for Java).

We have implemented the main part of the connector generator in Java, however we have used specialized framework Stratego/XT for processing the ElLang language and producing source code in a particular target language. Currently we are integrating the generator in the SOFA 2 component system [21]. The implementation is available at `http://dsrg.mff.cuni.cz/congen/`.

Acknowledgments

This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and partially supported by the ITEA/EUREKA project OSIRIS Σ!2023.

References

[1] The Apache Velocity Project. http://velocity.apache.org/

[2] B. Fischer, J. Schumann. AutoBayes: a system for generating data analysis programs from statistical models. J. Funct. Program. 13, 3 (May. 2003)

[3] B. Spitznagel, D. Garlan. A Compositional Formalization of Connector Wrappers. In *Proc. of ICSE'03*, Portland, Oregon, USA, May 2003

[4] D. Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *Proc. of ICSE'04*. IEEE CS, Washington, DC, May 2004

[5] D. Batory. Product-line architectures, aspects, and reuse (tutorial session). International Conference on Software Engineering, 2000

[6] E. Bruneton, T. Coupaye, J. B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proc. of WCOP'02*, Malaga, Spain, June 2002

[7] Eclipse Foundation, Inc. Eclipse JET (Java Emitter Templates). http://www.eclipse.org/emft/projects/jet/

[8] E. Visser. Stratego/XT. http://www.stratego-language.org/

[9] G. Kiczales, E. Hilsdale. Aspect-oriented programming. In *Proc. of EFSA'01*, Vienna, Austria, Sept 2001

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of ECOOP'01*, Budapest, Hungary, June 2001

[11] J. Kofron, J. Adamek, T. Bures, P. Jezek, V. Mencl, P. Parizek, F. Plasil. Checking Fractal Component Behavior Using Behavior Protocols. Presented at the 5th Fractal Workshop (part of ECOOP'06), Nantes, France, Jule 2006

[12] J. Whittle, J. Schumann. Automating the implementation of Kalman filter algorithms. ACM Trans. Math. Softw. 30, Dec 2004

[13] L. Bulej, T. Bures. Eliminating Execution Overhead of Disabled Optional Features in Connectors. In *Proc. of EWSA'06*, Nantes, France, LNCS 4344, Sept 2006

[14] M. Bravenboer, E. Visser. Concrete Syntax for Objects — Domain-Specific Language Embedding and Assimilation without Restrictions In *Proc. of OOPSLA'04*, Vancouver, Canada, Oct 2004

[15] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. IEEE Trans. on Software Engineering, vol. 21, no. 4, Apr 1995

[16] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. IEEE Trans. on Software Engineering, vol. 22, no. 6, Jun 1996

[17] N. Medvidovic,R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages IEEE Trans. Softw. Eng. 26, no. 1, Jan 2000

[18] S. Apel, T. Leich, G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proc. of International Conference on Software Engineering*, 2006

[19] T. Bures. Generating Connectors for Homogeneous and Heterogeneous Deployment. Ph.D. Thesis, Department of Software Engineering, Mathematical and Physical Faculty, Charles University, Prague, Sept 2006

[20] Visigoth Software Society. FreeMarker. http://www.freemarker.org/

[21] T. Bures, P. Hnetynka, F. Plasil. SOFA 2.0: Balancing advanced features in a hierarchival component model. In *Proc. of SERA 2006*, Seattle, USA, Aug 2006

[22] W3C. The extensible stylesheet language transformation, version 1.0, W3C Recommendation, Nov 1999