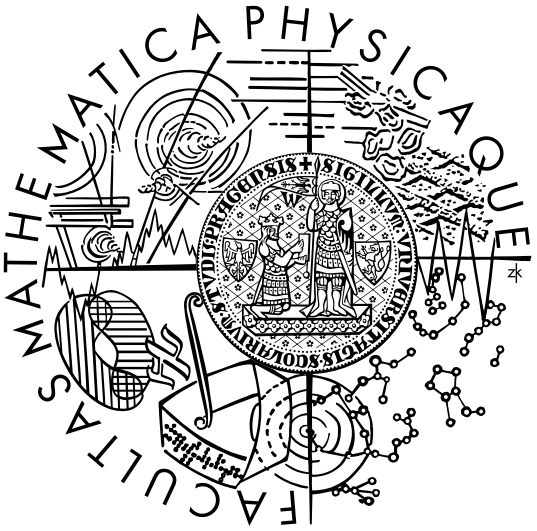
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Pavel Petřek

Optimizing of Software Connectors Code Generator's Performance

Department of Software Engineering
Supervisor: RNDr. Tomáš Bureš, Ph.D.
Study Program: Computer Science, Software Systems

This thesis has been produced by drawing upon the effort of a number of people. I wish to thank all of them, especially my advisor Tomáš Bureš for his ideas that led me to the successful completion of this project and for the time that he has spent advising me. I would also like to thank my friend Michal Malohlava, whose master thesis has been a starting resource for creating this work, for all his significant advice. I also wish to thank my family and all my interested friends for their strong moral support. A very special thanks also to my friends Pavla Kolářová and Kassidy Clark for their help with the proper grammar and wording of this document.

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

In Prague on August 8, 2007

Pavel Petřek

Název práce: Optimizing of Software Connectors Code Generator's Performance

Autor: Pavel Petřek

Katedra: Katedra Softwarového Inženýřství

Vedoucí diplomové práce: RNDr. Tomáš Bureš, Ph.D.

e-mail vedoucího: bures@dsrg.ms.mff.cuni.cz

Abstrakt: *Softwarové konektory jsou zprostředkovatelské entity používané v komponentových systémech k modelování a realizaci komunikace. Navíc mohou konektory poskytovat extra funkcionalitu, jako je logování nebo monitoring. Tato variabilita vyžaduje generování kódu konektoru na základě platných funkčních a ostatních požadavků. Ovšem některé požadavky nemohou být specifikovány dříve než v okamžiku nasazení. Prostředí při nasazování může být ale poměrně restriktivní.*

Existující generátor konektorů [32] používá ke generování tříd konektorů ze šablon sadu komplexních nástrojů. V této práci nabízíme optimalizaci generování na bázi předkompilování. Šablony jsou ještě v době návrhu předkompilovány do podoby, kterou lze posléze snadno zkompilovat použitím manipulace bytekódu.

Klíčová slova: Optimalizace, Bytecode, Stratego/XT, softwarové konektory, generování kódu, transformace kódu

Title: Optimizing of Software Connectors Code Generator's Performance

Author: Pavel Petřek

Department: Department of Software Engineering

Supervisor: RNDr. Tomáš Bureš, Ph.D.

Supervisor's e-mail address: bures@dsrg.ms.mff.cuni.cz

Abstract: *Software connectors are intermediary entities used to model and realize communication in component systems. In addition to the communication, connectors can also provide extra functionality, such as logging or monitoring. This variability requires generation of the connector's code according to valid functional and non-functional requirements. Some requirements cannot be specified sooner than at deployment-time. However, the deployment environment can be restrictive.*

The existing connector generator [32] utilizes complex external tools for generation of the connector's classes from templates. In this thesis, we propose an optimization based on a precompilation approach. Templates are precompiled at design-time into a form that can be later compiled easily using a bytecode manipulation technique.

Keywords: Optimization, Bytecode, Stratego/XT, software connectors, code generation, program transformations

Contents

Chapter 1

Introduction

During the past few decades, computer systems became an essential part of almost every business in the world. In the beginning, computers with available software solved only simple tasks, but more complex business requirements and larger amounts of input data have increased the complexity of software. To handle this complexity, several software engineering approaches have grown up. Component-based software engineering (CBSE) is one of them. CBSE presents a way of building large-scale applications using decomposition to smaller parts called components. A component is an independent piece of code that is responsible only for solving an exactly specified sub-task. Components are connected by using well-defined interfaces to realize mutual collaboration. In addition, due to its simplicity, components can be reused effectively.

There are many component systems available. The first one is a group of proprietary business oriented solutions that have widely-known code names, such as .NET component model from Microsoft [3], Enterprise Java Bean (EJB) from Sun Microsystems [5] or Corba Component Model (CCM) created by Object Management Group [4]. The second one is a group of academic solutions that typically offer models that are much more deeply thought-out. As an example, we mention SOFA [15] or Fractal [6].

1.1 Motivation

As we described in the text above, components are interconnected to achieve expected functionality. But each of the existing systems has a connection designed and implemented in its own way based on a different native middleware (e.g., .NET is based on .NET Remoting, EJB is based on RMI, CCM is based on CORBA). As a result, the final application works well in a homogenous environment. But if it is required to build a heterogenous application with more than one type of component, a problem appears. This problem can be solved by using available middleware bridges,

but these will work only for a particular architecture (e.g., JNBridge [12], J-Integra [8] are used for mediating a connection between .NET and EJB components). A more eligible solution for these heterogenous purposes are software connectors.

Another issue is a general ability of component connections monitoring at runtime. Existing component systems offer logging tools or single-purpose monitoring services for their supported middleware, e.g., [17]. However, as we switch to a heterogenous environment, we cannot guarantee unified monitoring over all involved systems. Once again, the software connectors can be a suitable solution.

The third issue is a limitation given by an underlying layer. A common situation in a business environment is that two remote instances of a component system are needed to cooperate, but their mutual direct network visibility is not possible at the moment (e.g., according to a requirement of low costs expended on their infrastructure). A solution could be a complicated adjustment of the middleware parts. But it can bring a significant increase of project time and extra costs for the given implementation. Software connectors utilizing parts of already existing architecture and infrastructure also seem to be a better solution in this case.

Now we will say what exactly are these software connectors? Software connectors as first-class entities realize an interaction-specific task in a component architecture. They utilize various communication styles based on various middlewares (e.g., remote procedure call used in CORBA [22] or Java RMI [19], messaging in JMS [20], CORBA Message Service [23] or JORAM [13], streaming in Helix DNA [7] or distributed shared memory in JavaSpaces [21]). Software connectors also bypass incompatibilities among particular component systems. Because a communication layer is a part of a connector, the connector separates a business layer of the architecture from the communication layer very well. A connector's design also allows to put some extra functionality into its implementation, such as measurement, logging or adaptation.

There are many existing connector generator implementations. The simplest solutions, such as CORBA or Java RMI, generate stubs and skeletons for a mutual interconnection of objects. A similar solution is used in .NET Remoting where proxies are generated to join remote objects. A more sophisticated solution is implemented in the Openwings project [14]. This project realizes a connector as a set of Java classes which are responsible for the binding of given components. In this case, connector classes are prepared at design-time. An example of an advanced solution is the Connector generator project (ConGen) [32]. It is designed for connector generation at deployment-time. The generation is based on a template system. It produces a bundle of connector classes based on the best evaluated architecture for a given components specification. An original version of ConGen [27] has been producing connector classes using a specialized set of classes instead of the class templates. The templates have been introduced to achieve better usability of the

solution.

In the previous text we used categories as design-time, deployment-time or run-time. From connector system's point of view we recognize three stages. First, there is the design-time stage where a connector framework and all of its parts including connector model are designed and prepared for deployment. Then, there is the deployment-time stage where the system is deployed, e.g., as a part of the connector, to a run-time environment. Lastly, there is the run-time stage where the framework is on and running the connectors.

We wrote several examples saying that a connector itself in various connector systems can be generated at each of these stages. During the design-time generation, all possible connector types and their bindings to all possible component types are to be prepared. This option does not allow any later adaptations of the connector at deployment-time, e.g., based on security settings. An example is the Openwings project. The second option is the generation at deployment-time. At this stage, only connector parts which are required by the current architecture are generated. An example is the ConGenproject. The third option is the generation of the connector or connector's parts at run-time which, for example, can be useful when new specific references are created in components at run-time. A simple example of this approach are EJB components where the stub and the skeletons are generated on-the-fly.

1.1.1 Goals

We already mentioned that the generation of the connectors at deployment-time or run-time is much more flexible than preparing them at design-time. However, a deployment environment is more restrictive than a design environment. Moving the generation to the deployment stage makes the deployment process more expensive than without the generation. Moreover, if we talk about the generation that takes place at run-time, its environment is even more restrictive.

The most significant requirement in the current ConGen is that the *Java* SDK is needed to generate code. A compilation using *javac* was also pointed out as a performance drawback in [32]. Typically, the generation of *Java* classes is optimized by using a technique called bytecode manipulation. It profits from the omission of the *Java* compiler tool and its parts from the generation process.

Thus, the goal of this thesis is to try to propose an optimization of the existing ConGen project to decrease its resources and tools needed at deployment-time and to speed up the compilation of the connector's code. The proposed optimization should preserve the current project's benefits, such as a highly usable template system and the architecture evaluation process which makes the project very robust.

Summarized goals of the thesis are:

- To propose a set of optimizations of the element generation in the existing connector generator solution.
  - To focus on the time needed to generate the class files of the element.
  - To focus on the additional environmental resources needed for the generation process.
  - To try to keep a concept of the optimization reusable for platforms other than Java.
- To develop appropriate supporting tools for the optimization.
- To measure and determine benefits of the proposed optimizations.

1.2 Structure of the thesis

First, in the following four chapters, the background of the connectors generation and bytecode manipulation will be introduced. A part of the introduction is also *Chapter 4*, which describes a framework used for the processing of a template content in the current version of ConGen and also to support proposed optimizations. Second, in *Chapter 6*, we will find an outline of the proposed optimization. The outline will consist of the particular steps which will be discussed and extended more in the following three chapters. Next, in *Chapter 10*, it will be evaluated how the optimized solution achieves the stated goals of this work. In this chapter, measurements of the original and the optimized solution will be compared. In *Chapter 11*, the summary of the works related to this thesis will be discussed and in *Chapter 12*, the entire thesis will be concluded. The appendix will introduce several code examples demonstrating the process of the generation using the optimized solution.

Chapter 2

Connectors

Software connectors as first-class entities act as a communication channel between components in component architectures. Furthermore, in distributed environments connectors span different address spaces to hide the distribution from the component's view. This is typically realized by using middleware.

2.1 Connector model

A connector model is a tool for the designers of the connectors. It describes a connector's fashion and its features. A connector model is typically a result of its primary focus.

An example of the connector model is the model introduced in [27], [26]. This model is based on a model from [25]. It is a core model for the ConGen solution discussed in this thesis. It follows the component paradigm (see *Figure 2.1*). A connector is modeled by a *connector architecture* defining the first level of nesting a set of *connector units*. Connector units will be discussed later.

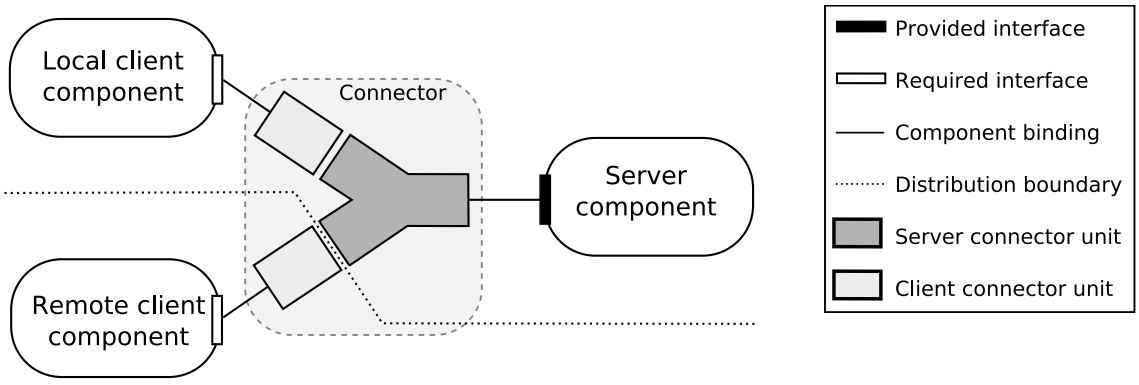


Figure 2.1: Connector architecture sample

2.2 Generation process

The generation process based on our connector model consists of two steps (see *Figure 2.2*). One of the model's views is used in each step. First, a human oriented *high-level connector specification* given by the deployment tool for connectors or by the connector designer is used to evaluate which architecture and layout are most appropriate. Second, a machine oriented *low-level connector configuration* is used to generate the code of the connector.

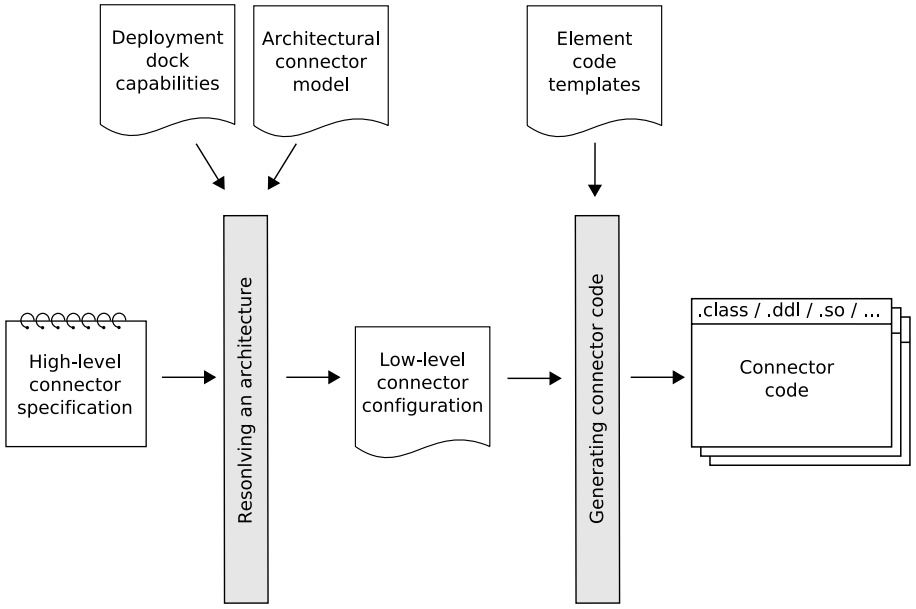


Figure 2.2: Overview of the generation process

2.2.1 High-level specification

A high-level specification of the connector is the starting input for the connector generator. It is used by a connector's deployment tool or a designer to specify the connector in an abstract way that is still convenient for a human. The specification contains a description of a communication style and a set of non-functional properties (NFP).

The description of the communication style is per-interface and specifies what type of communication an interface is associated with. Non-functional properties are named attributes in the dot notation and are written in forms of restrictions of attributes' values.

A component designer can prescribe some NFPs at design-time by associating them with the interface. These properties can comprise, for example, security requirements specifying that the interface provides sensitive data, or a threading policy specifying that parallel calls to the interface are supported. At the deploy-

ment phase, additional NFPs can be associated with the component interface or directly with the connector by a component deployer. These properties can comprise features such as the monitoring of the connector traffic by logging it to a file or by sending it to a monitoring database.

2.2.2 Low-level configuration

A low-level connector configuration defines a structure inside the concrete connector implementation. It consists of a description of the inner elements of the connector called *connector elements* and *element ports* and all the bindings among these ports. Analogous to the high-level connector specification, connector elements can be nested, so they create a bound entity from several connector elements.

The idea of binding makes us to see the configuration as a tree. Because the connector itself can be distributed over the network, the first level of the nesting is used for *connector units*. A connector unit is a direct descendant of the connector configuration tree's root describing a part of the connector code which is intended for a particular deployment dock. Because there are no additional specific requirements needed for the connector unit, it is, in fact, considered as one of the connector elements.

The connector elements and their subelements contain the element ports. An element port is an entity that has been introduced for use at the bottom level of the connector configuration tree. This port, together with its signatures, acts as a description of an element interface. Our connector model recognizes three types of the element ports:

- A *provided port* describes a locally provided interface.
- A *required port* describes a locally required interface.
- A *remote port* describes a remote interface.

As we mentioned in the previous text, a part of the connector configuration is a binding among the element ports. The binding is based on signatures of the ports and our connector model recognizes two types of bindings. The first one is a local binding between a provided and a required port (and vice versa) and the second is a remote binding among multiple remote ports.

The local binding is used for two connector elements inside one connector unit. These elements are in one address space and therefore it is possible to perform a call of the interface locally. Local binding is also used for a connection between a connector element and a component.

The remote binding is used for at least two connector elements located in more than one connector unit. The remote binding is undirected and is typically

realized by the underlaying middleware (e.g., RMI, JMS, CORBA). To support more complex communication styles than only a point-to-point style, our model defines the remote binding as a hyper-edge. Therefore, for example, a broadcasting communication style can be used. Moreover, one binding only will be needed to describe this communication.

The bindings follow a specific order in connections among connector units from different layers of the configuration hierarchy. If a nested connector element is provided with the remote port, it has to be delegated to a port of its bound element.

From the generation point of view, there are two architectures of the connector elements with their implementations, primitive and composite. The primitive architecture is only a code template and it realizes particular functionality such as an RMI stub, an RMI skeleton or a logger. This architecture corresponds to a leaf element of the configuration tree. The composite architecture corresponds to a bound element of the tree. It comprises all nested sub-elements and the bindings among them.

Figure 2.3 shows an example of the connector configuration. It consists of a server unit and two corresponding client units. The server unit is implemented by a server unit element. It is a composite element comprising multiple nested elements. Two of them are skeletons. One for local calls and the second for compressed RMI calls with minimized network traffic. The compressing RMI skeleton consists of an RMI skeleton element itself and an intercepting element that decompresses the parameters compressed on a remote client's side. The server unit also contains an example of two standalone intercepting elements. The first one is a logger which logs an incoming traffic and the second is a call serializer ensuring that all incoming calls will be processed in sequence. The first of the two client units is a local client unit and the second is a remote client unit. Both client units are implemented by a client unit element. The client unit element is a composite element and, in both cases, it contains a stub part for the corresponding server skeletons. For the local skeleton element, it is a local stub which is only an interceptor that forwards the passed parameters. For the compressed RMI skeleton, it is a pair of nested elements. The first of them is a compressor for the passed parameters and the second is an RMI stub itself. Multiple skeleton elements enable the server unit to serve multiple clients concurrently.

2.2.3 Resolving connector architecture

Connector architecture resolution is a process for transforming the high-level specification described in *Subsection 2.2.1* into the low-level configuration described in *Subsection 2.2.2*. In addition to information about the locations of the components in a deployment environment, the resolution process considers also communication

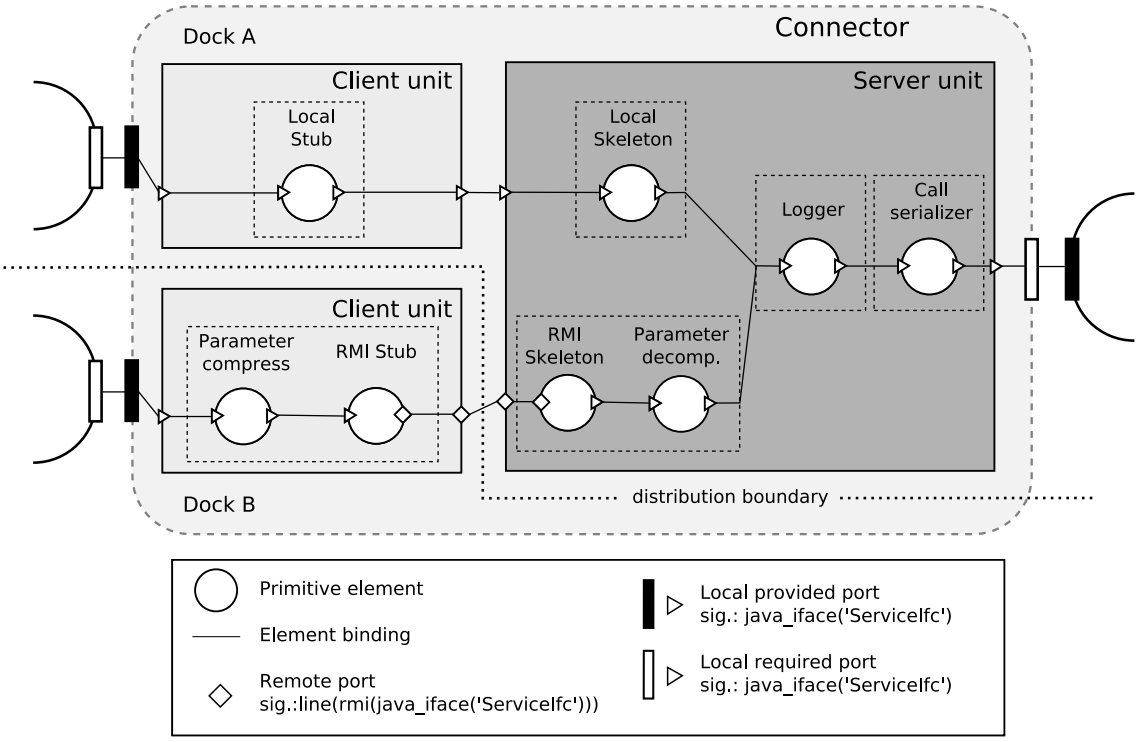


Figure 2.3: An example of low-level connector configuration

styles of the components and specified non-functional properties.

The resolution traverses the tree of all possible configurations known to the generator and determines which of them comply with the parameters from the high-level specification. First, during the resolution, an overall architecture is evaluated and then an architecture of the units and their subelements is recursively evaluated. From the list of the appropriate architectures, the "best" one is chosen. The evaluation of the best configuration could generally be a very complicated process based on various criteria to judge what is the "best" configuration (e.g., the lowest memory consumption, CPU utilization, latency, reliability, etc.). However, our model leaves the issue of choosing open and uses a quite simple solution which chooses the architecture with the lowest cost of the configuration. The costs are strictly assigned to all elements in the tree of known architectures.

2.2.4 Connector code generation

The second step of our connector generation approach is a process synthesizing the executable code of the connector from the low-level connector configuration obtained as an output of the first step.

As we described in *Subsection 2.2.2*, in terms of the connector configuration there is no specific need to distinguish the connector unit. Thus, the connector unit can be seen as a common connector element in the configuration hierarchy. There-

fore, the process of generation can be simplified to the generation of the connector elements only.

The generation of the connector element proceeds in two ways. For a primitive element, only its code based on a template is generated. For the composite element, both the wrapping code responsible for the creation of the nested elements and the bindings among them are generated. In both ways, additional element-type-specific actions can also be performed (e.g., optimizer on the code, *rmic* called on the generated code, etc.).

2.3 Existing generator solution

In this thesis, we discuss the connector model and the ConGen implementation of this model presented in [27]. The implementation was later extended in [32] to its current version. Inner architecture of the implementation is shown in *Figure 2.4*.

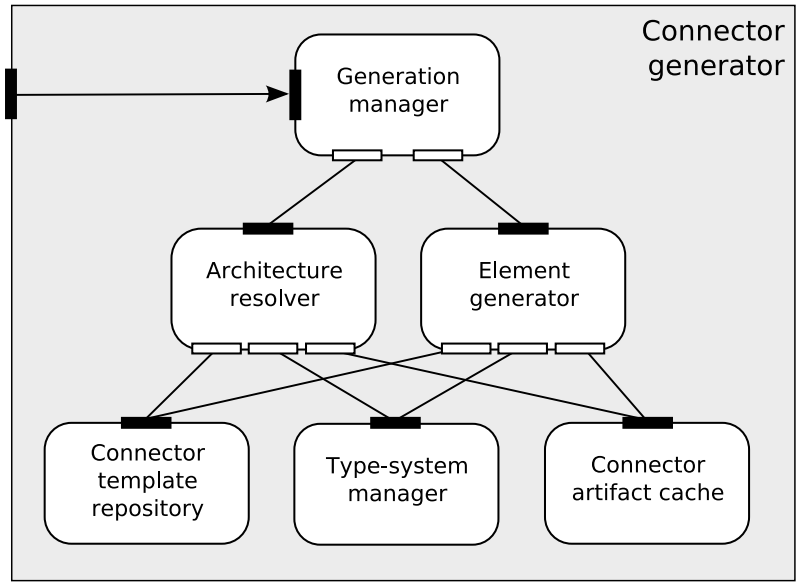


Figure 2.4: The architecture of the connector generator

The implementation satisfies the two steps described in the text above. The first part is an architecture resolver and the second one is the element generator. Collaboration of the parts in the processing is controlled by the generation manager.

2.3.1 Architecture resolver

A core of the architecture resolver is adjusted to searching for the "best" suitable architecture. The core of the searching is implemented in Prolog. Terms in Prolog are used to represent the low-level connector configuration. The terms are constructed from the database of all known architectures and connector units descrip-

tions specified in the high-level specification. NFPs are added as additional terms. Architectures matching to the terms cover the space of all suitable architectures. The matching architectures are evaluated in sequence and the one with the lowest costs is returned as the "best" architecture.

The input database of the element architectures known to the generator is stored as XML documents. An example of the descriptor of an RMI skeleton element is shown in *Listing 2.1.* The description contains hard-coded cost of the element usage, exposed ports of the element and NFPs related to the element. The second part of the descriptor includes a list of actions performed to generate the code of the element. This part is discussed later.

```
<element name="rmi_skeleton" type="skeleton"
impl-class="RMISkeleton">
  <architecture cost="4">
    <port name="line">
      <signature-entry ref-name="rmi" type="server" signature="rmi(I)"/>
    </port>
    <port name="call" signature="I"/>
  </architecture>

  <nfp-declarations>
    <nfp-mapping name="rpc(required_type)" value="rmi"/>
  </nfp-declarations>

  ... part describing generation actions ...

</element>
```

Listing 2.1: An example of an element architecture descriptor

The input high-level specification is also represented as an XML document. The specification is built by a deployment tool that has knowledge in a deployment dock. The tool put the dock requirements together with NFPs into the resulting XML document. The first level of the contained information specifies all *connector units* figuring in the solution. The unit descriptor contains nested information about exposed ports and NFPs to be satisfied. An example of the XML document is shown in *Listing 2.2.*

```
<specification>
  <unit name="server_unit" dock="host1">
    <port name="call" type="required" signature="java_interface('mypackage.Ifc')"/>
    <nfp-requirement
      predicate="nfp_mapping(Unit,'communication_style','method_invocation')"/>
    <nfp-requirement
      predicate="nfp_mapping(Unit,rpc(required_type),'axis')"/>
  </unit>

  <unit name="client_unit" dock="host1">
    <port name="call" type="provided" signature="java_interface('mypackage.Ifc')"/>
    <nfp-requirement
      predicate="nfp_mapping(Unit,'communication_style','method_invocation')"/>
  </unit>
```

```
  <unit name="client_unit" dock="host2">
    <port name="call" type="provided" signature="java_interface('mypackage.Ifc')"/>
    <nfp-requirement
      predicate="nfp_mapping(Unit,'communication_style','method_invocation')"/>
  </unit>
</specification>
```

Listing 2.2: An example of an element architecture descriptor

2.3.2 Element generator

In the low-level configuration obtained from the architecture resolver, we have full information about what source code for elements and what bindings shall be generated.

For each element of the configuration, we have a descriptor defining the element itself and the list of all actions that should be performed in sequence in order to generate the element's code. The range of the actions goes from producing the temporary target source code file, through a compilation of the file or a *rmic* call, up to a deletion of the temporary file. A name of each action corresponds to a *Java* implementation of the action.

The most important action is called *jimpl*. It is responsible for the *Java* source code file generation. In the ConGen implementation preceding [32] the *jimpl* action was implemented through a tree of generators. Each of the generators was based on the *JimplGeneratorInterface* interface. The generators were responsible for the hierarchical processing of simple templates to the target language. In the recent implementation, all the generators were replaced only by one consolidated generator which uses more complex template language for the generation of *Java* source code files. The template language as well as a platform used to define the language is described in following two chapters. The previous version of the generator is still available and can be used to generate target languages different from *Java*, but for purpose of this thesis it is not necessary to discuss it in detail.

If *Java* is the target language of the source code file, the *javac* action calling a native compiler is used to compile it. The *rmic* action is used to generate a stub part for the particular remote interface implementation. This is used especially for RMI skeleton and stub element generation. The *delete* action is used to delete temporary files that are no longer needed after the generation.

```
<element name="rmi_skeleton" type="skeleton" impl-class="RMISkeleton">

  ... part describing architecture ...

  <script>
    <command action="jimpl">
        <param name="generator"
          value="org.objectweb.dsrg.congen.elemgen.generators.stratego.StrategoGenerator"/>
```

```
        <param name="class" value="RMISkeleton"/>
        <param name="template" value="ellang/rmi_skeleton.template.ellang" />
    </command>

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

    <command action="rmic">
        <param name="class" value="RMISkeleton"/>
    </command>

    <command action="delete">
        <param name="source" value="RMISkeleton"/>
    </command>
  </script>
</element>
```

Listing 2.3: An example of an element architecture descriptor

Chapter 3

Template language

The current release of ConGen [32] discussed in *Chapter 2* uses a specially designed template language ElLang-J. This language is based on a combination of the target language (currently *Java*) with a meta language ElLang.

3.1 Template language ElLang

As we wrote above, ElLang is a part of the template language ElLang-J. It is designed from scratch as a meta language, so it can be easily encapsulated into the target programming language statements. The scope of the language contains simple *if*, *foreach*, *rforeach* or *set* and several more specific statements as, for example, an extension point or an import. All of these statements will be discussed later in a description of the merged language. Because ElLang is aimed at being embedded, it has no expressional power without being a part of the target language.

3.2 Template description

The ElLang-J template consists of target language statements and ElLang meta statements. The base of the template structure is a declaration of an *element*. The element defines the future class and can either be a pure element or can *extend* another element. The element declaration consists of included *interfaces* with their implemented methods and a number of methods without an explicit link to any interface. Furthermore, the element can contain any target language declaration valid for the level of class members and methods declarations. One of the important benefits of ElLang-J is the existence of *method templates* allowing the user to write a simple code template for the implementation of particular interface. The template code is then expanded for each method of the interface. *Listing 3.1* shows an example of an ElLang-J template.

```
package ${package};

import javax.net...runtime.*;

element Example extends "primitive_default.ellang" {

  /** Default constructor */
  public ${classname} {}

  implements interface PrimitiveInterface {
    /** Primitive method */
    public void primitive_method(int var1) {
      // ...
    }
  }

  implements interface ${InterfaceNameVariable} {
    method template {
      // ...
    }
  }
}
```

Listing 3.1: An example of an ElLang-J template

3.2.1 Meta variables

A meta-variable is a part of the ElLang-J language. It is a way of meta-value's projection into the target language. The meta-variable has two possible forms based on its type:

```
${var}
${v1.v2(v3=VALUE).v4}
```

where *var* is a name of the variable and *v1.v2(v3=VALUE).v4* is a sample query. The first type is a scalar (i.e., `${x}`) or array variable (i.e., `${y[2]}`) which can be declared in *set* or cycle meta-statements. The array variable can be indexed by either another meta-variable or by a valid literal (integer or string). The second type is a query variable. An expression contains names in a dot notation. Depending on the names, a value from the additional XML element descriptor or from a meta-variable containing a XML structure is selected. For more details on queries see [32].

3.2.2 Meta expressions

A meta-expression is used whenever further processing of the meta-value is required. It supports `+`, `!=` and `==` operators and as operands any meta-variable or literal (integer or string) can be used. Since the meta-expression is used as a part of some

meta-statement, there is no explicit need to bind the meta-variable in `${...}`. An example of the expression is shown in *Listing 3.2*.

```
$set y = 2$
$set x = y + 1$
$if (x == BINDINGS.BINDING(type=SRC).portnr)$
  ...
$end$
```

Listing 3.2: An example of meta-expression

3.2.3 Basic meta statements

Analogous to other procedural languages, ElLang-J offers meta-statements for setting meta-variables, cycles or conditioning.

3.2.3.1 Set statement

Set is a meta-statement for creating or changing values of meta-variables. It has the form:

```
$set v = e$
```

where v is a name of the variable, which is being set and e is a meta-expression whose resulting value is assigned into v. The scope of the created meta-variable is not restricted and its value can be used subsequently after its creation within the code of the template.

```
$set i = 1$
System.out.println("Available remote bindings:");
$foreach(BIND in ${BINDINGS.BINDING(type=REMOTE)})$
  System.out.println("  ${i} - ${BIND}");
  $set i = i + 1$
$end$
```

Listing 3.3: An example of set and foreach meta-statements usage

3.2.3.2 Foreach cycle

Foreach is a meta-statement that results in multiple copies of the included statements in the output code. It has the form:

```
$foreach(v1 in ${v2})$
  ... target language and/or ElLang statements ...
$end$
```

where *v1* is a simple scalar meta-variable name. *V2* is any meta-variable resulting in one or multiple values. In the case of the first type of the meta-variable,

v2 contains a single value. In the case of the query type, *v2* contains multiple values. An example usage is shown in *Listing 3.3*.

3.2.3.3 Recursive foreach cycle

Rforeach meta-statement is a special instrument to make a recursive version of the *foreach* cycle introduced in the previous text. It should be used if an expansion of the included statements is needed in a recursive way and the basic *foreach* statement cannot be used due to the grammar restrictions. *Rforeach* has the following form:

```
$rforeach(v1 in $[v2])$
  ... target language and/or ElLang statements ...
  $recpoint$
  ... target language and/or ElLang statements ...
$final$
  ... target language and/or ElLang statements ...
$end$
```

where *v1* and *v2* have the same meaning as in the *foreach* statement. The difference is that there are two sections of the statements. The first has the same meaning as the *foreach* section, i.e., to be repeated. The second set of statements beginning with the `$final$` keyword, is used as the last level of recursion if there are more than zero values in *v2*, or is the only resulting list of statements if there are no values in *v2*. The point of recursion is marked in the first block statement by the special `$recpoint$` meta-statement.

The recursive version of *foreach* evaluates the final code as follows. As we said in the previous paragraph, if there are no values given by the *v2* variable, only the *final* block of statements will be written to the output. Otherwise, the first value of *v2* is taken and the *v1* meta-variable is evaluated in the first block of statements. The evaluated block is added to the output. If there are still some values in *v2*, we process an expansion for the second value and the `$recpoint$` meta-statement is replaced with the expansion output. Accordingly, we process the other values from *v2*. As soon as there are no more values in *v2*, the `$recpoint$` is replaced with the *final* part of the statements. *Listing 3.4* shows a sample usage of *rforeach*.

```
$rforeach(BIND in ${BINDINGS.BINDING(type=REMOTE)})$
  if ("${BIND.port1.id}".equals(givenPortID)) {
    ...
  } else
    $recpoint$
  $final$
    throws new Exception("Requested port '" + givenPortID + "' not found.");
$end$
```

Listing 3.4: An example of rforeach meta-statement usage

3.2.3.4 If statement

If is a meta-statement for conditional output of its contained code. There are two types of this statements and these have the following forms:

```
$if(e)$
  ... target language and/or ElLang statements ...
$end$

$if(e)$
  ... target language and/or ElLang statements ...
$else$
  ... target language and/or ElLang statements ...
$end$
```

where e is a meta-expression. In the first type, the statements are generated to an output if and only if a value of the meta-expression is not equal to 0. In the second type, the first branch of statements is generated to the output under the same condition, otherwise the second branch of statements is generated.

```
$if (BINDING.from.element.name == "this")$
  $if (BINDING.to.element.name == "this")$
    System.out.println("Fully local binding");
  $else$
    System.out.println("One way local binding");
  $end$
$else$
  $if (BINDING.to.element.name == "this")$
    System.out.println("One way local binding");
  $else$
    System.out.println("Fully remote binding");
  $end$
$end$
```

Listing 3.5: An example of if meta-statement usage

3.2.3.5 Import statement

Import is a meta-statement for importing another template file. The name of the imported file is given as a parameter. This meta-statement is used for importing either full interface definitions or just certain methods.

```
$import("definitions/defaults/version_methods.ellang")$
```

Listing 3.6: An example of import meta-statement usage

3.2.4 Template hierarchy

As we wrote at the beginning of this section, a template can easily extend another template. If extended, all content of the extending element is taken and added as

part of the extended element body. The idea of this construction is to get a hierarchical structure of the templates with simple inheritance. Cycles are not allowed. The problem of method names collision is left to a template designer's responsibility.

3.2.5 Template extension points

Another feature provided by the ElLang-J is the concept of extension points. The basic idea consists of defining a location inside the code or a block of the code that can be replaced in a descendant template. The two forms of the extension point look as follows:

```
/* first type */
$extPoint(name)$

/* second type */
$extPoint(name)$
  ... target language and/or ElLang statements ...
$end$
```

where *name* is an identifier of the extension point. As soon as the extension point is defined, we can easily substitute it into an element's descendant by using the code:

```
$defExtPoint(name)$
  ... target language and/or ElLang statements ...
$end$
```

The new given block of statements will fully replace the original location or the marked block of the extension point's statements.

3.2.6 Method templates

One of the advanced ideas in the ElLang-J template concept is a method template. Many element templates in the component system need only to be adjusted, for example, for ports available in a connector configuration or other similar modifications of some method's code. However, there are situations when we need to make a template for the entire method of a particular interface. For this purpose, ElLang-J provides the method templates.

A method template is declared in an interface definition. It has the form:

```
implements interface ${ifc.name} {
  method template {
    ... target language and/or ElLang statements ...
  }
}
```

In contrast to the code of common template methods, the code of the method template can use several special meta-variables beginning with the prefix `method.`. The meta-variables are related to the method of the interface that is generated. More specifically, there are meta-variables of the following names defined for the scope of the method template:

- method.name - name of the generated method.
- method.variables - contains list of the method parameter names.
- method.declareReturnValue - declares a variable for a return value if the method has a return type.
- method.returnVar - name of the variable declared in the previous declaration. If the method does not return any value, this meta-variable contains 0.
- method.returnStm - generates a return statement. If there is a value to be returned, the variable declared above is used.

Listing 3.7 shows a typical usage of the method template in an interface given by a meta-variable. In this case, it is obvious that methods cannot be defined one by one, because an interface name is not available while designing the template. The only activity in the code of the method template is the forwarding of the call.

```
implements interface ${ports.port(name=call).signature} {
  method template {
    ${method.declareReturnValue}
    try {
      $if (method.returnVar)$
        ${method.returnVar} = this.target.${method.name}(${method.variables});
      $else$
        this.target.${method.name}(${method.variables});
      $end$
    } catch (java.rmi.Exception e) {
      throw new org.objectweb.dsrg.connector.ConnectorTransportException(e);
    }
    ${method.returnStm}
  }
}
```

Listing 3.7: An example of method template usage

3.3 Template evaluation

A process of the template evaluation, split into individual tasks, is shown in *Figure 3.1.* It includes all of the steps which take place to get a final ATerm. The final ATerm is almost ready to be pretty-printed as the *Java* code. To see more details about the ATerm see *Section 4.3.*

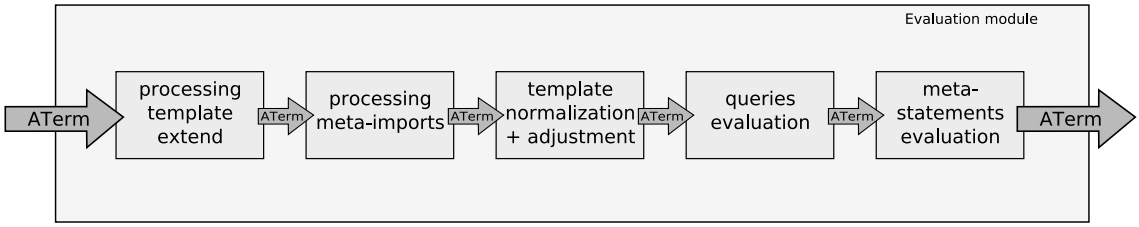


Figure 3.1: Template evaluation process

An evaluation module is called from the main STRATEGO program *elgenerator*. *Elgenerator* expects an XML descriptor as an input. The descriptor contains a path to a template and all settings of an element.

The evaluation begins with the processing of *extends*. A path of the extended template file, which is a parameter of the extension, is required to be relative to the launching directory. The launching directory is equivalent to the root template path from the XML descriptor. The extended element is firstly checked for another extension and the body is added as part of the evaluated template. The module also checks for cyclic dependencies, that are not allowed, but it does not check for method name collisions.

The next step of the evaluation is the processing of imports. The path of an imported file, which is given as a parameter, is also relative to the launching directory. The content of the file is parsed using the *sglri* XT tool and the definition table of the grammar. A possibility of using import for several levels, i.e., interface, method or field, is given by multiple starting symbols set to the parser.

The next step is the last part of the elaboration. The template is normalized, which means that statements with multiple versions of the constructor are transformed into a unified form. An example of the normalization is the *if* meta-statement, which has two forms, one with and one without an else branch. In this step, all occurrences of the term `IF(expression,statements*)` denoting the version without else branch are replaced with `IF(expression,statements*,[])` having an empty statements list as the else branch. After the normalization, all extension points are processed. This means that for each `$defExtPoint(name)$` occurrence, the original point `$extPoint(name)$` is found and replaced with the new version of the code.

The fourth step moves towards meta-variables and meta-expressions, where queries are evaluated. Each query is mapped to information from an element's XML descriptor structure and the new value replaces the original location of the meta-variable. For more details on queries see [32].

In the last step, meta-statements and remaining meta-variables are processed. This step proceeds hierarchically because of the control meta-variable's validity in the cycle meta-statements. Moreover, the hierarchical processing ensures that values

of the meta-variables are used in the same order as they are set. The described way of processing is simply realizable in STRATEGO using a proper traversal strategy, in this case, the one traversing an ATerm in top-down direction.

At this point, the template needs only a few structural adjustments and the final java code can be produced. These adjustments are also handled by the *Elgenerator* program after applying the evaluation.

Chapter 4

Overview of Stratego/XT

The current version of ConGen uses a system of templates to produce element code. A program producing the connector element's code is developed using Stratego/XT. Part of the optimization proposed later in this thesis is a program developed using the same platform. Thus, we will briefly introduce what Stratego/XT is.

Stratego/XT [16] is a framework combining a programming language called Stratego and a collection of transformation tools based on the Stratego language called XT.

Stratego is a transformation language designed for term rewriting. It provides various features such as transformation and traversal strategies, simplified transformation rules, variables and dynamic rules. It can be used for simple pipe transformation as well as for complex program transformation systems.

The XT bundle is a supporting set of tools making it much easier to generate a parser, a pretty-printer or an abstract syntax tree transformation for a particular language. It also contains a set of already generated tools for common languages, such as XML. XT is based on the Stratego language, Syntax Definition Formalism (SDF) designed for the language syntax definition and Generic Pretty-Printing (GPP) package using the Box language for production of readable outputs.

4.1 Architecture

XT is a collection of components used to implement the transformation system. Each component is a single executable that can handle standard input and output as well as to read and write from / to a file. Using a shell pipe, multiple tools can be connected together, so it makes a full transformation system. An example of a typical sequence of tools is shown in *Figure 4.1.* Based on the SDF of the language, a parser, all transformations and a pretty-printer are generated. First, a program is

transformed using the parser into an abstract syntax tree (AST) form. Second, the transformations (e.g., a desugarer, a simplifier, an optimizer) are used to produce another AST. Finally, the pretty-printer is used to produce the original language from the transformed AST.

A definition of the grammar plays a basic role in the transformation process. It uses the SDF language described in *Section 4.2*.

Data between the XT tools are exchanged in the form of an abstract syntax tree (AST). This tree form is equivalent to the form of prefix terms. By term, we mean a constructor with zero or more subterms. String and Integer constants are also valid terms. The tree is described from the top level down to its leafs. For example, an expression `(3 + a) * 4` is represented as `Times(Plus(Int(3),Var("a")), Int(4))`. Because many parts are repeatedly used in a notation, the Stratego/XT tools uses an optimized internal representation. The form is called Annotated Term (ATerm) and it will be discussed later in *Section 4.3*.

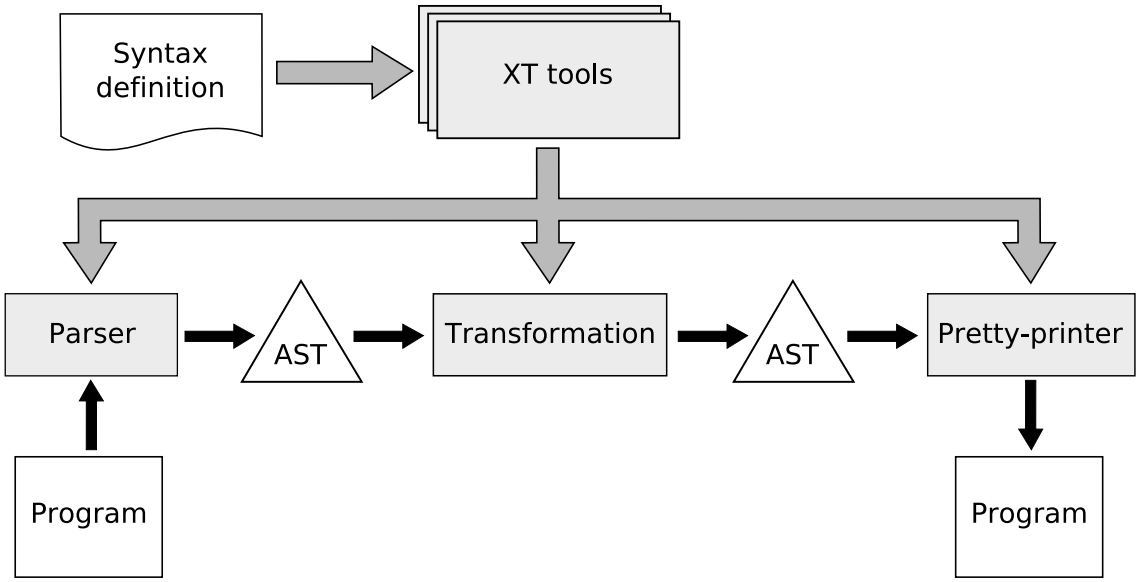


Figure 4.1: An example of a typical Stratego/XT transformation system

4.2 Syntax Definition Formalism SDF

Stratego/XT uses the SDF language for defining the syntax of a programming language. It consists of lexical and context-free definition of the syntax. The lexical definition is used to define lexical tokens in the program and the context-free one defines a grammar of lexical elements. To be a well-arranged structure, the syntax definition can be split into multiple modules. SDF also includes a disambiguation construct to declaratively define certain kinds of derivations, which are not allowed or are prioritized, i.e., priorities or lexical restrictions.

A syntax definition in SDF contains enough information for the generation of various tools and descriptors of the defined language. A parser, data type definitions and a basic pretty-printer can be easily generated from the syntax definition.

The automatically generated parser is based on *Scannerless Generalized-LR parsing* [28] and it directly produces an abstract syntax tree of an input program in the ATerm representation.

4.2.1 Abstract syntax tree

As we mentioned in the previous text, the XT tools use a tree representation of a program for exchanging information. The representation is called *Abstract syntax tree* (AST). It is based on a parsed tree obtained from an input program, but it does not contain any unnecessary information, such as whitespace or comments. To skip the unnecessary information, a constructor annotation extends a production defined in SDF. The constructors create the AST output.

Stratego/XT stores the AST in a term format called ATerm described in *Section 4.3*.

4.3 ATerm

The Annotated Terms (ATerms) are heavily used in Stratego/XT for storing and exchanging an internal program representation among tools. An automatically generated parser discussed in the previous section produces an ATerm which can be an input for another Stratego/XT programs. The ATerms are also used internally in programs written in the Stratego language. The format uses the principle of sharing, so whenever the same information occurs in the tree, the second occurrence is stored as a pointer to the the first definition. An example of the relationship between an input program and the AST form is shown in *Figure 4.2*.

ATerms are constructed from the following elements:

- Constructor application - a notation `c(t1,...,tn)` creates a term by applying a constructor to the list of zero or more terms.
- Tuple - a notation `(t1,...,tn)` creates a constructor application without constructor.
- List - a notation `[t1,...,tn]` creates an ordered list of zero or more terms.
- String - a constant written in double quotes is a String term. Special characters such as double quotes, newlines or backslashes should be escaped using a backslash character.

- Integer - a constant written as a set of the decimal digits is an Integer term.
- Annotation - previous elements create the structural part of terms. Besides, a notation `t{t1,...,tn}` can be used to annotate the term with a list of other terms. Typically, these terms carry additional semantic information.

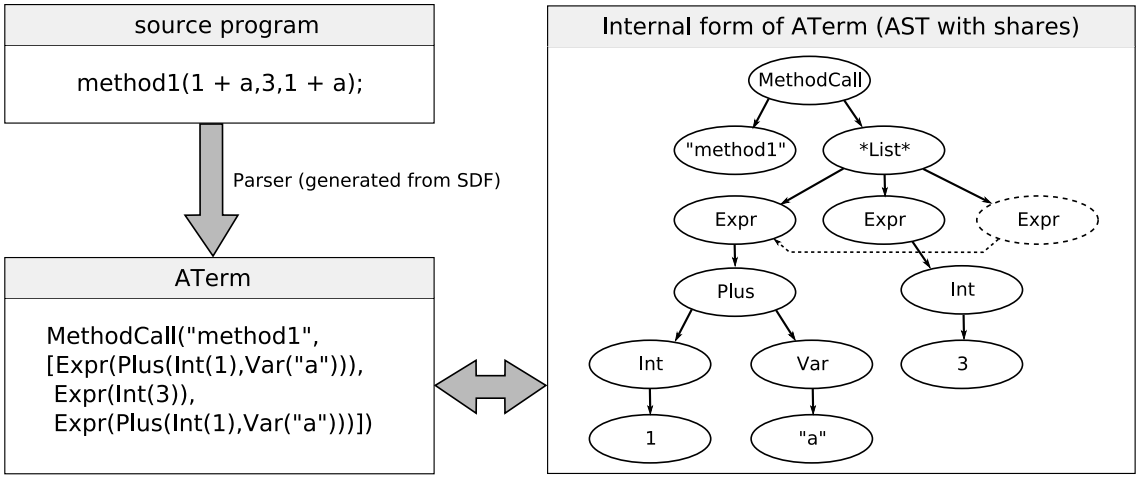


Figure 4.2: Steps to get AST from an input program

4.4 Stratego programming language

Stratego is a programming language designed for program transformations. It is based on the paradigm of rewriting using programmable rewriting strategies and dynamic rules. To represent a program it uses ATerm form discussed in *Section 4.3*.

4.4.1 Program representation

Analogous to the SDF language, Stratego programs are divided into modules for better organization and reusability. The modules are identified by a unique hierarchical name and can be mutually imported. The Stratego programs consist of three types of sections introduced by the following keywords:

- signature - begins a section containing term declarations. It contains information about sorts and their constructors. Mostly, it is automatically generated from the SDF grammar definition, but additional declarations can be manually added.
- strategies - begins a section of rewriting strategies' definitions.
- rules - begins a section of rewriting rules' definitions. In fact, the rewriting rule is only a syntactic sugar for the rewriting strategy.

Listing 4.1 shows an example of a STRATEGO program aimed at pretty-printing of an XML file. First, it includes the common STRATEGO library `liblib`, the XT library `strategoxt-xtc-tools` for handling a standard set of XT tools command line options and the latter two libraries for handling an XML document. Second, there is a main strategy defined in the section of strategies. It calls a wrapping strategy `xtc-io-wrap` for handling input and output based on the standard XT parameters and options. Parsing and pretty-printing strategies are passed as parameters to the wrapper.

```
/**
 * Stratego example.
 */
module MyPPXML

imports
  liblib
  strategoxt-xtc-tools
  xml-xtc-tools
  xml-info

strategies
    io-MyPPXML = xtc-io-wrap(xtc-parse-xml-info; xtc-pp-xml-info)
```

Listing 4.1: An example of a Stratego program

These strategies use terms and matching variables put in a sequential order. They handle the failed calls of other strategies and use plenty of instruments predefined in the common libraries. For more detailed information about the STRATEGO syntax, see [16].

Chapter 5

Bytecode manipulation

At the beginning of the thesis, we mentioned that the typical way of optimizing generation of *Java* code is bytecode manipulation. Because later we will propose an approach that utilizes this technique, this chapter introduces some aspects of the manipulation.

5.1 Java architecture

Before we get to the manipulation itself, we give a brief summary of the *Java* architecture.

In past years, the *Java* technology has entered almost all fields of the IT world. As it was designed for networks, almost any device able to connect to a network is also capable of running certain forms of Java program.

Java architecture consists of four parts [33]. The Java programming language, the Java class file, the Java API and the Java Virtual Machine. When a developer writes a program in Java, he certainly touches all of these four technologies. First, he types the program in the Java language. Second, he compiles it into the class file form. Last, he runs it in a Java virtual machine environment and calls the Java API instruments. *Figure 5.1* shows the relationship among the four parts of the Java architecture.

Together, the Java virtual machine and the Java API make up the *platform* for which all programs are compiled. The *Java platform* is a part of the architecture that can be implemented on different kinds of devices so it can run the compiled Java program.

The heart of Java's orientation to networks is the Java virtual machine. It supports all three requirements of the network orientation: platform independency, security and mobility over the network.

The Java virtual machine (JVM) is an abstract computer. Its specification

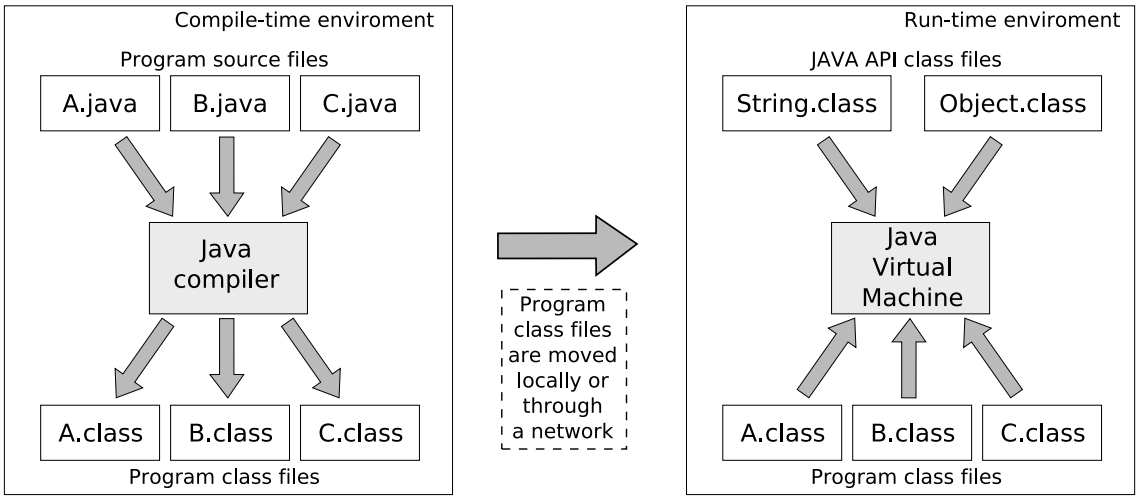


Figure 5.1: Java programming environment

sets up only certain required features of every JVMs, but many decisions are still left to the particular implementation designers. An example of the requirement is that JVM must be able to run Java bytecode, but they are free in choosing the technique used to execute it. It can be run completely in a software environment or can be supported, at least partially, in a hardware level. Flexible specification allows JVM to be implemented on wide field of devices and computers.

A primary task of JVM is to load and run code program code. JVM consists of a class loader and an execution engine. The class loader reads the class files from various resources and passes them to the engine responsible for interpreting the code. There are four kinds of execution engines. The first kind is the simplest one. It only interprets the code, one at a time. Another kind of execution engine is a just-in-time compilation. The speed benefit obtained from translating bytecode of class's methods to the much faster native code of the target platform is paid by a higher memory requirement. The just-in-time compilation was the first significant optimization that the Sun Microsystems company brought to the original Sun JVM. The third kind of execution engine is an adaptive optimizer. In this approach, JVM starts to interpret the bytecode, but then it monitors the activity of the running code and identifies often used locations in the program. The most often running parts are concurrently compiled to the native code and strongly optimized. The rest of the bytecode remains being interpreted by JVM. A typical example of this approach is the Hotspot technology that was firstly released in Sun Java 1.3 [18]. The last kind of the execution, which stands outside of the first three software oriented executions, is a hardware JVM. This JVM is built on top of a chip that is capable of running Java bytecode natively.

From the previous list of execution engine types, it is obvious that if we do not want to prioritize any target platform, we cannot perform any platform-specific

optimizations. Thus, to produce general code it is more convenient for compilers to leave the optimizations to JVM. As a benefit, the class file code is easy to be further processed.

In the previous text, we mentioned the Java bytecode multiple times. The bytecode is a machine language of JVM. Compared to the source code from which the bytecode is compiled, the bytecode is less abstract, more compact and rather computer-centric, which makes it less human-readable. Although, when being accessed by one of the available bytecode frameworks, particular operations are easily distinguishable.

5.1.1 Java class file

For a better understanding of the following introduction of bytecode manipulation, it is useful to have some general knowledge of the structure of a Java class file as specified in [31].

Generally, a class file consists of:

- Header containing the version information of the file
- Constant pool containing the list of constants identified by their indexes in this list
- Access rights descriptor containing information about modifiers used for this class
- Superclass of this class or interface (if the class file realizes the interface, this information must always be *java.lang.Object*)
- Implemented interfaces list specifying all directly implemented interfaces (not those implemented by transitive closure)
- Field list containing a record consisting of the name, descriptor, access rights for each field and a set of attributes (for instance, the initial value of the static field)
- Method (plus constructors and optional static method) list containing records consisting of the name, descriptor, access rights for each method and set of attributes (for instance, exceptions thrown by the method or the code of the method)
- Class attribute list that contains general information about the class or interface (for instance, the source file)

All textual descriptors and names are realized by indexes of the constant pool items, which contain the actual values. This is also used for instructions that use textual descriptors. These instructions contain the index of the required constant pool item.

Internal descriptors of the fully qualified class or interface names use a slash (‘/’) symbol instead of ASCII periods (‘.’). For example `java.lang.String` will be described as `"java/lang/String"`.

The internal descriptor of a field distinguishes three types:

- Basic type where the type is represented by the respective letter (*B*, *C*, *D*, *F*, *I*, *J*, *S*, *Z* for *byte*, *char*, *double*, *float*, *integer*, *long*, *short*, *boolean*)
- Object type in the form `"L<classname>;"` where `<classname>` is an internal description of the fully qualified class or interface name
- Array type in the form `"[<array component>"` where `<array component>` can be any basic, object or array type

The internal descriptors of methods have the following syntax:

```
(<parameter type>*)<return type>
```

where `<parameter type>*` denotes zero or more internal field descriptors and `<return type>` denotes either an internal field descriptor or *V* letter if the method returns *void*. An example is shown in *Listing 5.1*.

```
double[][] FooMethod(String s, long l) {}

/* described as */

(Ljava/lang/String;J)[[D
```

Listing 5.1: An example of internal java method descriptor

5.2 Manipulation objectives

There are many techniques used to implement dynamically adaptable systems. Most of them use a certain form of interposition, e.g., interposition objects, meta objects. This technique includes, for example, modification of semantics of components by modifying functional or non functional properties. It can also dynamically add or remove additional properties.

Most of the interposition techniques utilize some tools for code generation or for modifying existing code. Apparently, the interposition object must be of the

same type as the object it hides, which makes it possible to install the interposition object to the original location. Another situation is when we implement a meta object protocol. Code manipulation tools must be used to inline the interposition code directly at a meta element's location in the base class or at least to generate the class linked from the location.

Thus, the bytecode manipulation and generation tools are useful for implementing adaptable systems. There are two ways of their typical usage. They can be used either statically or dynamically. Although the static solution is simpler to develop, the dynamic way is more practical for the user. An example of the dynamic approach is on-demand *rmic* compiler call at run-time. In open systems, the dynamic solution is even required, because new inestimable object types can be required on the fly. Generated interposition code is used to load the type.

One of the dynamic solutions is to generate source code and then dynamically compile it. But this approach significantly increases the time and space overhead of the application, because Java SDK (including for example *javac* or *rmic* compilers) has to be included in the distribution package of an application. Therefore, to have an optimal solution it is necessary to use one of the existing bytecode manipulation tools that allow a user to perform operations on bytecode without any additional requirements (e.g., Java SDK).

5.3 Major manipulation frameworks

In the previous section, we explained the reason for using manipulation frameworks. The manipulation tools are generally aimed at the higher level of accessing the class file structure, its contained fields and methods and the bytecode of the methods.

In the present situation, there are several frameworks available. Mostly two of them are used. These are the BCEL project [2] and the ASM project [29]. BCEL started up sooner than ASM. In fact, ASM was a reaction to BCEL's robustness and its poor performance.

Byte Code Engineering Library (BCEL) is a toolkit for static analysis or dynamic creation and transformation of Java class files. It enables a programmer to implement bytecode manipulation code at a high level of abstraction without dealing with the internal details of the class file structure. A part of the project is also its own repository, where classes reside parsed, and which takes responsibility for looking for and parsing another class file if it is needed, e.g., when being linked from another class file. An integral part of the framework is the project's bytecode verifier called *JustIce*, which is a powerful tool to reveal any problems in the code.

Like BCEL, ASM is a toolkit designed to dynamically manipulate and transform Java class files. It is aimed at performance. Compared to BCEL, it is more

than ten times smaller (21kB compared to 350kB) and processing overhead adds about 60% to the system class loading time compared to 700% in BCEL [29]. ASM brings a new technique into the class files manipulation. While BCEL parses a class file and represents various structures of the class as a tree class structure, ASM uses the visitor approach which simply traverses all items from the class file and makes a given visitor visit them. The visits proceed as the information is fetched from the file and it is not kept in memory. BCEL also supports the visitor approach, but it is implemented on top of the complete memory representation of the class file, so there is no performance benefit.

Since BCEL is designed as a robust solution, it provides class files parsed in a well-formed tree structure, where all information can be accessed at any time. On the other hand, this approach requires a lot of memory to keep the structure and even special overhead needed to build it. ASM uses the visitor approach which optimizes both mentioned drawbacks, but also makes the solution less abstract. For a better understanding, both approaches are shown in *Figure 5.3* and *Figure 5.2*.

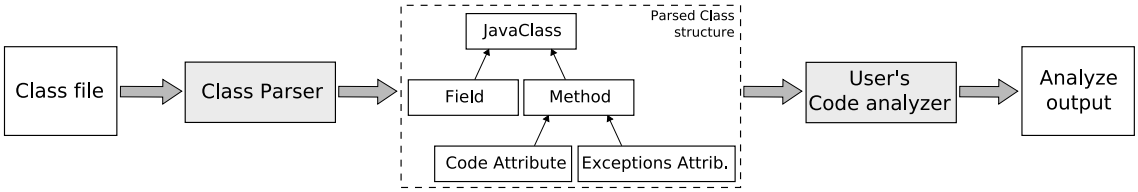


Figure 5.2: An example of class parsing using BCEL

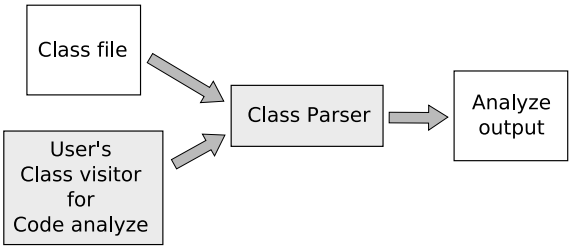


Figure 5.3: An example of class parsing using ASM

Chapter 6

Solution outline

In the previous text, we presented the existing connector element generator together with its language for templates ElLang-J. The ElLang-J language is a mixture of a target language (in the current solution it is *Java*) and a meta language enabling the adjustability of code. The connector elements are designed in this solution in the form of template files. To produce the code of the element, there are two steps at connector's deployment-time - processing the ElLang-J template using a Stratego program, which outputs *Java* source code, and compiling the *Java* file using *javac*. This approach offers a very comfortable tool for the designer of the element, because it primarily provides a full view of the code. On the other hand, the approach brings non-trivial requirements to the deployment stage, namely the Stratego program for producing the code and the Java SDK needed for its compilation. Using the required external parts, especially the SDK, makes the solution too expensive.

As we stated at the beginning of the thesis, we want to optimize the solution to decrease its resource requirements. We also require, that the particular steps of the optimization will preserve the usability benefits of the template approach as well as the robust concept of the overall connector generation.

There are two possible places to focus on, the Stratego program *elgenerator* and the *Java* compiler *javac*. As we explained in *Subsection 1.1.1*, we have decided to focus on the *Java* compiler that is bringing the whole *Java* SDK and is therefore, a more significant burden for overall generation.

The basic idea of entire the optimization is to precompile all elements' templates at design-time, when it has already been designed. The reason to do the precompilation, is to process the code at deployment-time in a much simpler way without using *javac*. However, the template evaluation at deployment-time can only evaluate the ElLang-J form of the template. Therefore, the precompiled form must match the ElLang-J syntax. Moreover, it has to keep the most important information obtained from the *Java* compiler, which is the bytecode of the class methods.

The precompilation takes place at design-time. First, it has to transforms the template so it can be compiled by *javac*. Second, it has to decompile the produced class file into an original ElLang-J shape with all of its meta-information. The only difference will be that the actual *Java* code will be replaced with its bytecode form.

As an alternative approach to the precompilation, we have also considered modifying one of the available open source *Java* compilers to replace the *javac* (e.g., Janino compiler [9] or Jikes [11] projects). But each of the compilers was either too limited (e.g., supporting only *Java* 1.4) or too big, so the replacement would not optimize the environment requirements at all.

Because each of the summarized steps of the precompilation is non-trivial, we will develop a prototype set of tools as a part of the proposal to support the optimizations. The tools should fit into the existing ConGen framework written in Java and if possible, it should not extend the current requirements of the framework. Other requirements would decrease the meaning of this work.

The following three chapters will describe, in detail, the architectural changes introduced in the proposed solution and each of the steps needed for operating with the precompiled templates. In *Chapter 10*, we will present a set of the optimized solution's measurements in comparison with the existing ConGen solution. The measurements should emphasize the benefits obtained by the avoidance of the *javac* compilation at deployment-time.

Chapter 7

Architecture changes

The connector element generator is currently implemented for the target language of *Java*. A generation process consists of two steps. The first one is producing Java source code based on an element descriptor and a template written in ElLang-J language. This part is processed by the Stratego program *Elgenerator*. The second one is the compilation of the generated files using the *javac* compiler.

As we said, there are two possible parts to be optimized. However, one of the requirements for the proposed changes is to preserve the comfort of the template language. Therefore, there is no option of removing the Stratego part. It could be replaced, for example, by a more independent part written in *Java* and doing the same work. But the other program processing the template would have to deal with a task of the same complexity and would save at maximum one native program call. Moreover, we have decided to focus on *Java* SDK which seems to be a more significant performance drawback. Let us have a closer look at the compilation part changes described in the previous chapter.

The proposed idea is to precompile ElLang-J templates at design-time, so *javac* could be replaced by some simpler and less demanding instrument. For a better understanding of the approach, *Figure 7.1* shows the existing element code generation and *Figure 7.2* shows the proposed solution using the precompilation of the templates.

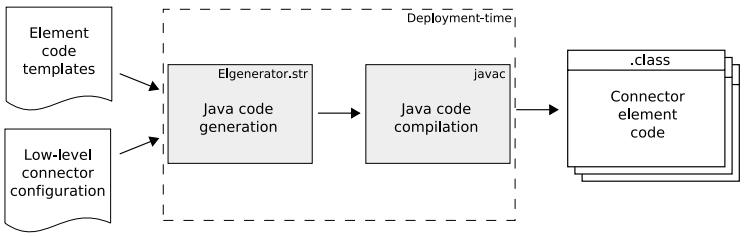


Figure 7.1: Original element code generation

The way of the precompilation is to precompile at design-time the designed

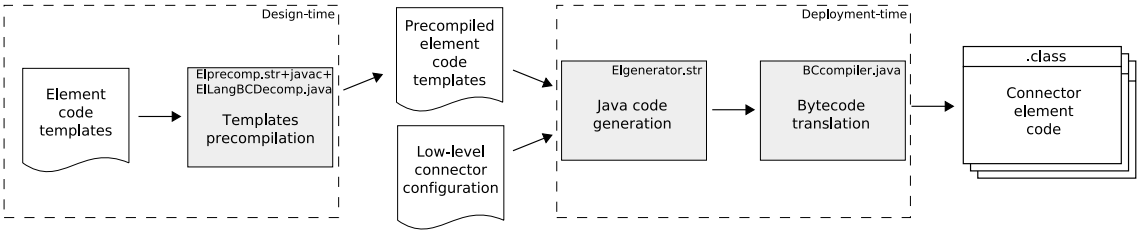


Figure 7.2: Proposed element code generation

element template written in the ElLang-J language into a form which does not require a full java compilation at the deployment stage. Instead, it is already compiled and stored in a way syntactically identical to the ElLang-J form and is only translated to a proper class file. Because the essential information for the Java class file format is the bytecode of the involved methods, it is crucial to store the bytecode information in the precompiled form. For this form, the name ElLang-J-BC was selected and it is described in the following subsection.

7.0.1 Ellang-J-BC variant of the Ellang-J language

The basic idea of the ElLang-J-BC form of the ElLang-J language is that we want to use the current template evaluation process without any changes, but we want to record a bytecode in method bodies. To fulfill the first part, ElLang-J-BC must use the completely same structure of a file as its ancestor. For the second part, we will use the syntax of the Java method body for recording the bytecode. This means that bytecode instructions will be written as *Java* function calls. Parameters of the call will be written in a numeric form or as a string literals.

Each bytecode instruction has its binary code that identifies it in a method body in an actual class file. Each bytecode instruction also has a name created by using a mnemonic code expressing its semantics. This name is used to identify the instruction in ElLang-J-BC. Because all labels consist of letter characters only and there are no special symbols used, the prefixed form of the syntax used in *Java* is very convenient. Moreover, because all variable information (e.g., field names, method names or used classes) are put in bytecode as parameters to the bytecode instruction, it allows us to use meta-variables at any place as it was in ElLang-J code.

However, it is not as direct as it seems. As we described in *Chapter 5*, the notation of class type descriptors and field type descriptors in a class file differ from their forms in *Java* code. Because we need to be able to insert this information using the meta-variable, the descriptors must be recorded in a *Java* source code fashion. This means that, for example, the instructions *NEW* and *PUTFIELD* will appear as

```
NEW("java.lang.String");
PUTFIELD("package.MyClass","myField","java.io.String");
```

instead of

```
NEW("java/lang/String");
PUTFIELD("package/MyClass","myField","Ljava/io/String;");
```

Another problem to be solved is parameter usage in method templates. In *Java* bytecode, there are no named local variables. The local variables, among which the method parameters belong, are identified by a local variable integer id. Therefore, during the precompilation step, we have to keep track of the method template parameters usage in order to be able to identify them in the precompiled template using the original meta-variable for the method template parameters. This way is syntactically correct, because numeric and textual parameters in the method call are exchangeable in the Java grammar.

In *Listing 7.1*, an example of a precompile ElLang-J-BC template is shown. An original ElLang-J form of the same template is shown in *Listing 7.2*.

```
package ${package};

element ElementExample {
  public java.lang.String getElementInfo(java.lang.String arg0) {
    LABEL(0);
    NEW("java.lang.StringBuilder");
    DUP();
    INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
    ASTORE(2);
    LABEL(1);
    ALOAD(2);
    NEW("java.lang.StringBuilder");
    DUP();
    INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.StringBuilder","append",
      "java.lang.StringBuilder(java.lang.String)");
    LDC(S("Implementation: ${implementation}\n"));
    INVOKEVIRTUAL("java.lang.StringBuilder","append",
      "java.lang.StringBuilder(java.lang.String)");
    INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
    INVOKEVIRTUAL("java.lang.StringBuilder","append",
      "java.lang.StringBuilder(java.lang.String)");
    POP();
    LABEL(2);
    LINE(119);
    ALOAD(2);
    INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
    ARETURN();
  }
}
```

Listing 7.1: An example of a precompiled ElLang-J-BC template

```
package ${package};

element ElementExample {
  public String getElementInfo(String indent) {
    StringBuilder result = new StringBuilder();
    result.append(indent + "Implementation: ${STR_implementation}\n");
    return result.toString();
  }
}
```

Listing 7.2: An example of an original ElLang-J template

Chapter 8

Design-time template precompilation

In this chapter, we will clarify how to get the precompiled form of the ElLang-J templates.

The precompilation basically consists of four parts. The first one is the transformation of the template into java code which is acceptable by the Java compiler. The second part is the compilation step. The third one is the decompilation of the class file to an ElLang-J-BC form with all original meta-information included. And the last one is the redirecting of the element's descriptor to the precompiled template. The entire process is shown in *Figure 8.1.*

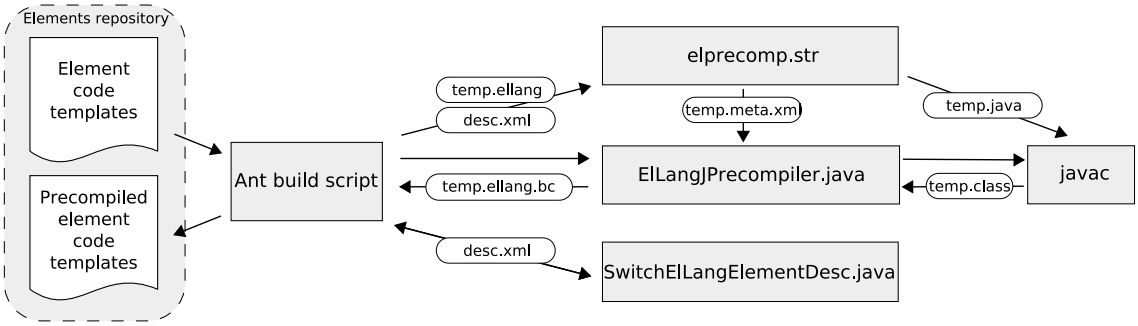


Figure 8.1: Precompilation process

The precompilation of ElLang-J templates takes place at the design-time of the connector generator. This step proceeds along with the compilation of the entire framework written in Java. Therefore, it is not a problem that an essential requirement for its second substep is the Java compiler.

8.0.2 Preparation of Ellang-J template using Stratego

The preparation of an ElLang-J template into a compilable form is the first step of the precompilation. The main idea of this step rests in replacing the meta-statements and meta-variables with a unique pattern of code. This pattern allows us to recognize the original location of the meta-information later in the decompiling part. The original information is stored separately and indexed by a unique identifier, which is then used as part of the code pattern. Particular code patterns are discussed in the following subsection.

The most convenient way of modifying and replacing ElLang-J grammar's elements is to use Stratego. Thus, for this step we have developed a new Stratego program called *elprecomp*. This program fetches an element descriptor from a connector template repository and prepares all ElLang-J templates that the descriptor references. The prepared templates are stored as *.java* files. Together with the prepared templates, the related meta-information is also stored as XML files in the same directory. The *elprecomp* program finally outputs names of all the prepared files.

Another Stratego utility was developed to format the meta-information in the output XML file. It is called *pp-ellang* and its responsibility is to transform an ElLang-J ATerm into the original ElLang-J language fashion. The utility is launched from the *elprecomp* program.

8.0.2.1 Tagging meta-artifacts

The *elprecomp* program transforms templates into compilable Java source code. It tags a location of every occurrence of meta-information with appropriate java code that is recognizable later in the decompilation part. Meta-statements are replaced with a special method call and meta-variables with a special constant or a reference. Meta-expressions used in all meta-statements and queries used in meta-variables are not kept in the code. Instead, they are stored in a separate XML file and indexed by a unique identifier. This identifier is used as a parameter of the replacement method call or as a part of the replacement constant or reference. We can use a method replacement without any limitations, because the meta-statement is mapped in ElLang-J grammar to a Java statement.

Two steps are performed before *elprecomp* starts tagging the meta-information. First, all extends, includes and extension points are evaluated. All of these instruments do not need to be tagged for later reconstruction. This is because, in terms of the element generation performed at deployment-time, the set of template files is fixed. Second, the method template declarations are expanded.

The expansion of the method template proceeds as follows. The basic question with regard to the meta-variables related to the method template is whether the

method returns a value or not (see *Subsection 3.2.6*). Thus, the body of the method template is copied twice into the final code. Once as a method returning a value and a second time as a void method. There is only one integer value declared as a parameter of each of the methods. All occurrences of meta-variables related to the method template are evaluated according to the particular methods in the bodies ofboth added methods. These values do not need to be tagged, because its usage has the meaning in a specific location only where they are easily recognizable. For example, a location where values of the method parameters are used is denoted by the `method.variables` meta-variable. This place is recognized based on the number of the internal local variable, which is fixed for method parameters.

During the processing, all meta-information is processed sequentially. This means that after the decompilation step all *set* meta-statements will still be in the original order.

Every occurrence of the *set* meta-statement is tagged in the code by a method call:

```
org.objectweb.dsrg.congen.meta.MetaClass.replace_set(ID);
```

where *ID* is a unique identifier of the original meta-expression in the output XML. The counting of IDs starts at 987651234 and increases by 1. The reason of choosing the threshold is discussed later in the text.

Every occurrence of the *if* meta-statement is tagged in the code by one of the two following sets of method calls according to an *else* branch presence:

```
org.objectweb.dsrg.congen.meta.MetaClass.replace_if(ID);
...
org.objectweb.dsrg.congen.meta.MetaClass.replace_end();

/* or */

org.objectweb.dsrg.congen.meta.MetaClass.replace_if(ID);
{
  ...
}
org.objectweb.dsrg.congen.meta.MetaClass.replace_else();
{
  ...
}
org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
```

where *ID* is a unique identifier of the condition's meta-expression in the output XML. As you can see, the version with the *else* branch encloses both sets of statements into the block of statements. This ensures that two local variables of the same name will not cause a conflict during compilation. Although, this variable should also not be used in the code after the *if* meta-statement.

Every occurrence of the *foreach* meta-statement is tagged in the code by the

following method calls:

```
org.objectweb.dsrg.congen.meta.MetaClass.replace_foreach(ID);
...
org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
```

where *ID* is a unique identifier of the inner meta-expression of the *foreach* meta-statement.

Every occurrence of the *rforeach* meta-statement is tagged in the code by the following method calls:

```
org.objectweb.dsrg.congen.meta.MetaClass.replace_rforeach(ID);
...
org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
...
org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
...
org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
```

where *ID* is a unique identifier of the inner meta-expression of the *rforach* meta-statement. Moreover, we have to deal with the unreachable code problem. According to the *Java* language specification, unreachable code should be an error [30]. But the syntax of the *rforeach* meta-statement allows the user to write a construction which causes the unreachable code problem after the meta-statement tagging. An example of such a problematic construction is shown in *Listing 8.1*. The very last `replace_end()` method call will certainly cause the failure of the compilation. One solution of this problem is to wrap all *return* and *raise* statements with a dummy condition

```
if(true) {
  return;
}
```

in all methods containing *rforeach*. This step will guarantee that every statement in the method will be reachable. A particular drawback of the solution is that the user has to pay special attention to a real unreachable code.

```
org.objectweb.dsrg.congen.meta.MetaClass.replace_rforeach(987651234);
if ("${PORT.name}".equals(portName)) {
  Object result = subElements.get("${PORT.linked.name}");
  return result;
} else
  org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
  throw new Exception("Port not found");
org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
```

Listing 8.1: An example of problematic recursive foreach construction

Now we get to the meta-variables tagging. Since every situation of using the meta-variable is specific, we will discuss all the situations separately.

If the `${package}` meta-variable is found in the package descriptor, it is directly replaced with the path of the package containing the templates.

If the meta-variable is used as a type of the field, it is replaced with the universal class

```
org.objectweb.dsrg.congen.meta.MetaFieldType fieldName;
```

and the fieldName is used as an identifier in the output XML. This is the only kind of a textual identifier in the XML.

If the meta-variable is used as a type of the implemented interface, it is replaced with new dynamically created interface. An integer identifier of the meta-variable for the output XML is used as a part of the interface name. The implemented interface, for example, appears as follows:

```
class TestElement
  implements
    org.objectweb.dsrg
      .congen.conrep.elements.templates.ellang.temp.MetaInterface987651235 {
    ...
}
```

The identifier in the name of the interface is also used as a part of the names of the expanded method templates to bind the methods with the interface.

If the `${classname}` meta-variable is found in the constructor name, it is directly replaced with the name of the currently processed element. Since constructor is unambiguously recognizable in the class file, it is not needed to store this meta-variable usage.

A special case arises when using a meta-variable inside a Java string literal. For this purpose, ELLANG-J defined a new mechanism of treating string literals in *Java* statements. However, the only step during the preparation process is the replacing of the meta-variable with a unique string value containing the identifier to the XML output. The string value, for example, appears as follows:

```
System.out.println("org.objectweb.dsrg.congen.meta.MetaString987651237;");
```

Another peculiar usage of the meta-variable is a typecasting. In this case, we will assume that the only target of this typecasting is an assignment to the field of the type given by the meta-variable. As it was written above, such a field's type is always replaced with the same intermediate type. This type will also be used as a replacement in the typecasting. Since there can be more meta-typed fields in one class and the replacement type does not contain the identifier, we cannot directly determine the replaced meta-variable. Thus, as the identifier for the reconstruction, we will use the name of the field, which is the target of the type casted value's assignment.

If the meta-variable is used as an array index, we will use the identifier of the value in the output XML as the index value. The only limitation of this approach is that there should not be constant indexes in a certain range of values specified by the user. This is one of the reasons that the identifiers start at the high value.

Analogous to the previous situation is the situation in which the meta-variable is used as an index in an array allocation. The identifier of the meta-variable is also used as the index in the allocation.

The most tricky usage of the meta-variable is, when the meta-variable stands for a constructor name in the *new* allocation. In this case, we do not have enough information to build a call, because we need the exact method descriptor of the constructor. One solution would be to use the types of parameters given to the constructor call. Unfortunately, this solution does not comply with optional implicit typecasting of the parameter values. Thus, the correct solution is to extend the ElLang-J grammar with a special syntax for using the meta-variable as the constructor name in the *new* allocation. This meta-variable has to bring the parameter types with it to build the descriptor of constructor. The change of the grammar is discussed later. The second problem in the *new* call is the type of the allocated object. This type is dynamic, so we cannot exactly say what the expected type will be. The solution is to use the *null* value, instead of the output value of *new* and to call the original *new* statement as the next statement. This fact should be considered later in the decompilation part to merge the split command again. For a simplicity, we assume that the dynamic constructor call is used only in an assignment statement.

Most of the replacements introduced above contain the numeric identifier that determines the index of the original meta-information stored in the additional XML output. As we already said, the identifier starts at value 987651234. The first reason to start with the high number as a way of loading constants in bytecode is because multiple basic constant values close to 0 have a dedicated instruction to load it, but all the higher numbers use the `LDC` instruction. Therefore, to start the indexing at 0 would cause slightly complicated reconstruction of the code. The second reason, which is more important, was already explained in the paragraph describing the replacement in the array indexed by the meta-variable. Since users typically do not use such high indexes as constants, we will use this particular range for our purposes. The range is bound to the maximum of 512 indexed meta-information, so the constants higher than 987651234 + 512 can be used by the user as an index again.

8.0.2.2 ElLang-J grammar changes

As we explained in the previous text, for the purpose of the precompilation, we have to extend the existing ElLang-J grammar. This extension is related to the meta-variable used as a constructor name a *new* allocation. In this operation we need the full type information about the parameters of the constructor. Therefore, we extend the meta-variable for this situation:

```
new ${var as (type1, type2, ...)} (par1, par2, ...);
```

where *var* is a name of the variable, *typei* is a type of i-th parameter and *pari* is the value of i-th parameter given to the call. This change does not affect any other part of the grammar.

8.0.3 Java class file decompiler

As soon as the *Java* code is produced out of the *elprecomp* program, the additional temporary interfaces and classes are created and the prepared program is compiled using *javac*. The product of the compilation step is a class file. The class file is decompiled in the last step.

All the meta-information is reconstructed during the decompilation. To recognize the original meta-information's locations in bytecode, the decompiler searches for the specific code patterns described in the previous text. A part of most of the code patterns is an identifier of the meta-information. This identifier is the search key for the additional output XML file obtained from the *elprecomp* program.

The way of the meta-information's embedding into the ElLang-J-BC language was already described in the previous text.

In *Chapter 5*, we introduced the major frameworks for bytecode manipulation. For our purposes we chose *ASM*, which is more efficient, although it does not contain many addition features in comparison with *BCEL*. Thus, a structure of the decompiled class as well as every single bytecode instruction is read using the *ASM* framework.

There are several remarkable points in the decompilation process. The first one is a recording of conditional and unconditional jumps, try-catch blocks and switch decision in the bytecode. Since the deployment-time evaluation of the element's template can replace one instruction with multiple others, it is inefficient to store the jump target as an absolute position in the bytecode. Instead, there are numbered labels inserted as additional pseudo-instructions and the jumping instructions use the numbers from the label as parameters. An example of code is shown in *Listing 8.2*.

```
package ${package};

element ElementExample {
  public void exampleMethod(String indent) {
    LABEL(0);
    LDC(S("line"));
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
 /* If the java.lang.String.equals call returned 0, the code continues at the LABEL(1) */
    IFEQ(1);
    LABEL(2);
    GETSTATIC("java.lang.System","out","java.io.PrintStream");
    LDC(S("Value is line"));
    INVOKEVIRTUAL("java.io.PrintStream","println","void(java.lang.String)");
 /* Target of the conditional jump */
    LABEL(1);
    RETURN;
  }
}
```

Listing 8.2: An example of labels in ElLang-J-BC template

A part of the decompilation is also the reconstruction of the method templates. We already described the way of binding the method templates' implementations to an implemented interface. While decompiled, both implementations of the one original method template are merged together using the meta-condition statement. The decompiled form is also marked using the additional pseudo-instruction `TEMPLATE();`. The described code pattern is shown in *Listing 8.3*. However, there is one limitation of this approach. It is the fact that a relative modification of the meta-variables is performed twice. For example, if there is

```
$set var = var + 1$
```

used in a method template, it will be performed twice. The result value will be different from the expected number, so a potential usage of the meta-variable *var* in latter code is incorrect. Fortunately, if the template's designer keeps this limitation in mind, he can write the code in a different way.

```
...
  method template {
    TEMPLATE();
    $if (method.returnVar)$
      /* Bytecode for methods returning a value */
    $else$
      /* Bytecode for void methods */
    $end$
  }
...
```

Listing 8.3: A merge pattern used while decompiling a method template

The last noteworthy step of the decompilation procedure is the processing of the implemented interfaces. Some of the interfaces can be used while processing the

method templates. The others are simply added to the decompiled template with an empty body. The original information which binds the interfaces to groups of methods is unnecessary for further evaluation of the template.

The decompilation step is a part of the *ElLangJPrecompiler* program written in Java. This program is responsible for calling the *javac* compiler for the code produced out of the *elprecomp* program. *ElLangJPrecompiler* is called from the build script of the ConGen framework. The script calls the Stratego preparation program followed by the *Java* decompilation program. The final product of the whole precompilation is stored in the same directory as the original ElLang-J template with a new extension *.ellang.bc*.

If the precompilation of an element succeeds, the element's descriptor is automatically redirected to the new precompiled template and the new compiler. The redirection is processed by the *SwitchElLangElementDesc* program.

Chapter 9

Deployment-time ElLang-J-BC compilation

This chapter will explain the construction of the class file without calling the *Java* compiler at deployment-time.

As we already described, our proposed solution preserves the original step of evaluating the code of the template. The precompiled templates syntactically correspond to the ElLang-J format expected by the *elgenerator* program. The set of bytecode instructions in the form of the *Java* statements is stored as method bodies of the precompiled methods. Thus, *elgenerator* will generate a *Java* source file containing methods consisting of bytecode instructions.

As soon as we obtain the precompiled source code of the element we can get to its compilation. Since we want to keep the compilation step as simple as possible, we decided for the form of a single *Java* program. This program reads the input file as a textual stream and it generates a class, fields of the class, methods of the class and the code of the methods on-the-fly.

An important issue in the reading of the input file are syntax aberrations in the stored code. To read the input file without any advanced parsing techniques, we need a guarantee that the code has a strict form and there are no aberrations. Namely, the rules are:

- There will not be any *Java* comments going over multiple rows.
- Every *Java* statement, which actually represents the bytecode instruction, will always be stored at most one per row.
- A row containing a *Java* statement will not contain any comments.
- Method and field definitions will not be going over multiple rows.

Fortunately, the code of the prepared file is generated by the *pp-java* program (a pretty-printer for the *Java* language) called from *elgenerator*, which formats *Java* code in the form that is fully compliant with the given rules.

Analogous to the decompilation step of the precompilation process, we use the *ASM* framework to create the final class file structure and to construct the code of the methods. Moreover, the linear instrumentation of the code is pretty straightforward, so we do not have to use any extended features of *ASM* (such as tree code view) and we can call visitors for adding the instructions only.

While translating the textual bytecode aliases into the real instructions, we have to keep in mind that there are particular specific code extensions introduced in the previous chapter. Each of the extensions has to be treated in a special way. Specifically, it is a parameter list in the expanded method template and a return type of the method template.

A usage of the method template parameters is marked using the prefix `EXT_` in the template's code and it looks as follows:

```
EXT_ILOAD(${method.variables});
```

During the evaluation of the template preceding this compilation, the meta-variable `${method.variables}` is replaced with the list of the parameter names of the real expanded method. The prefix `EXT_` of the instruction is preserved. As soon as the compiler gets to this instruction, the instruction is replaced by the set of the instruction loading all parameters. In addition to this particular instruction replacement, all local variable indexes greater than the indexes of the variables carrying method parameters have to be adjusted to the number of method parameters. The local variables in a method code are indexed starting at 0 in following order: *this* object reference, followed by the method parameters and then local variables used in the method. An example of the replacement is shown in *Listing 9.1.*

An important fact is that the code pattern for fetching the parameters' values is the same in classes generated from various *Java* SDK vendors. Thus, the described procedure is not vendor-specific.

```
/* Code after the meta-variable is replaced */
void test(int p1, java.lang.String p2, long p3) {
  ...
  EXT_ILOAD(p1, p2, p3);
  ...
  ASTORE(2);
  ...
}

/* Code of the final class file */
void test(int p1, java.lang.String p2, long p3) {
  ...
  ILOAD(1);
  ALOAD(2);
```

```
  LLOAD(3);
  ...
  ASTORE(4);
  ...
}
```

Listing 9.1: A sample compilation of a reading of the method template's parameters

The second specific extension that we mentioned is the dealing with a return type of the method template. A particular method contains bytecode that treats the return value as integer after the expansion of the method template. This is caused by the precompilation process using the integer type for precompilation of the method template (see *Chapter 8*). The problem can be solved by keeping track of methods' return types in method headers. All manipulations with the returning values are adjusted according to the collected types. It is important that the local variable reserved for the return value is always indexed right after the method parameters, because the initialization of the return value was forced as the first statements in the precompiled method template. An example is shown in *Listing 9.2* for better understanding.

```
/* Code after evaluation of the template */
java.lang.String test(int p1) {
  ...
  ISTORE(2);
  ILOAD(2);
  IRETURN();
  ...
}

/* Code after compilation */
java.lang.String test(int p1) {
  ...
  ASTORE(2);
  ALOAD(2);
  ARETURN();
  ...
}
```

Listing 9.2: A sample compilation of a method template's returning value

As a result of the described compilation procedure we get a class file. This class file is equivalent to a class file obtained from the original approach working without the precompilation.

The compilation is performed by the *BCCompiler.java* program in our proposed solution. The program is integrated as the new action named *javacbc* into the ConGen framework. This way offers a convenient way of the switching between solutions (existing and the precompiled) without any source code changes as it was described in the previous chapter.

Chapter 10

Evaluation

In this chapter, we summarize the changes of the existing architecture and present measurements and explanations that support the eligibility of the solution. Finally, the limitations of the optimizations are concluded.

10.1 Architecture revisited

We introduced partial changes as well as the new steps that have to be performed for optimization of the existing connector generation process. In this section we will conclude the changes and explain what changes need to be done in the current version of the ConGen framework.

Because we want to omit the *Java* compiler from deployment-time, we have decided to use the precompilation approach. In this approach, we transform connector elements' code templates written in the ElLang-J language into the precompiled form. The precompiled form does not require *javac* to build a final class file. The precompiled form is denoted as ElLang-J-BC and it stores method code as bytecode instruction. Each instruction is represented as a *Java* method call. The precompiled form preserves also the meta-information used for dynamic evaluation of the connector's element source code. Since the syntax of the precompiled file is compliant with the ElLang-J syntax, the existing source code generator written in Stratego can be used.

The precompilation takes place at design-time of the connector generator's framework. It processes templates in the entire element repository and if it succeeds, it redirects the element's descriptor in the repository to the precompiled template. One practical advantage of this solution is that only one more step is required to use the precompiled solution. This step is an extra target in the Ant build script that builds the distribution as the original solution and then invokes the precompilation.

The precompilation consists of two parts. The first part is a program written

in STRATEGO which transforms the template into the compilable *Java* form and storing the meta-information into the separated XML file. This meta-information is used in the second part, which is realized by a program written in *Java*. The program transforms the bytecode of all methods into method bodies of the precompiled template. Both parts are coupled together in the Ant build script. The build script ensures also the final redirecting of the descriptors in the element repository.

As a part of the precompilation designing, we had to slightly extend the existing ELLANG-J grammar to be able to process a constructor call denoted by a meta-variable.

We also had to develop a standard STRATEGO pretty-printing application responsible for formatting the ELLANG-J meta-information back from the ATerm. The formatted form written in an XML file is a part of the precompilation's output.

The compiler of the precompiled code is markedly simpler than *javac*. It takes only the source file, which is already dynamically evaluated by the existing STRATEGO generator, and produces the class file within a one-pass read.

An important fact is how the new compiler is involved in a generation at deployment-time. This is handled by the new element-type-specific action *javacbc*, which replaces the existing unwanted *javac* action.

10.2 Measurements

In this section, timing and memory-consumption measurements will be presented.

The testbed for the measurements was a machine with the following configuration: Single Pentium-M CPU 1300MHz, 512MB of DDR266 RAM and a local 80GB PATA disk. The installed operating system is Debian Linux 3.1 with kernel version 2.4.27. There is one JVM installation, Sun JDK 1.6.0.01. We used another operating system for better comparison in the memory-consumption test. The second operating system is installed on the same machine and it is Windows XP Service Pack 2. There is also one JVM installation, Sun JDK 1.6.0.01, available in the system.

10.2.1 Element source code compiler timing

The first of the measurements is a timing of the new compiler of the source code of the element in comparison with the *javac* compiler. *Javac* was called in this test as a shell process. For this purpose, a measuring capability for the element generation actions was added to the connector generator framework.

The test was performed on the demo connector included in the CONGEN's repository. It consists of 11 *Java* classes that are constructed using the STRATEGO

element	javac	javacbc	speed-up
LocalSkeleton	1182 ms	380 ms	3.1
ServerUnit	724 ms	123 ms	5.9
ClientUnit	675 ms	97 ms	7.0
LocalStub	399 ms	61 ms	6.5
ClientUnit	845 ms	88 ms	9.6
LocalSkeleton	493 ms	56 ms	8.8
ServerUnit	840 ms	67 ms	12.5
LocalStub	498 ms	43 ms	11.6
ClientUnit	597 ms	66 ms	9.0
ClientUnit	617 ms	364 ms	1.7
ServerUnit	709 ms	60 ms	11.8
total	7579 ms	1405 ms	5.4

Table 10.1: Compilation times for demo connector's elements using the original javac compiler and the new javacbc compiler

generator. The test was repeatedly run 20 times, the extreme values were filtered and the average times for each of the class can be found in *Table 10.2*.

If we look at the table, we can see notably greater values in the first row than in the rest of the sample. This was most probably caused by the time needed for loading all of the parts required for the compilation, e.g., classes, libraries, etc.

The second interesting observation in the table is the surprisingly high value 364ms for the last ClientUnit unit. By more detailed examination of the measurement logs, we have found that during this *javacbc* step, the garbage collector regularly performed its clean-up, which prolonged the timing value.

The overall results in the table prove that using the new compiler is about 5 times faster than using the *javac*.

10.2.2 Saved memory spent by javac

In the second measurement, we focus on the memory spent by the *javac* compiler. This amount will be saved by omitting *javac* calls from the connector generation procedure. The memory used for *javacbc* is insignificant in this case, because the new compiler runs internally in JVM and utilizes only very limited set of classes.

This test was performed on both JVMs introduced at the beginning of this section, one in Linux and one in Windows. Since there is no common way in measuring JVM's memory consumption from within a *Java* program, we have used information from the external listing of the system processes. The measurement consisted of the compilation of two samples. The first sample was the extremely simple class with the link to core *Java* classes only. The class is shown in *Listing 10.1*. The second

environment	simple class	ConGen classes
Linux / Sun JDK 1.6.0.01	21.0 MB	42.0 MB
Windows XP / Sun JDK 1.6.0.01	17.0 MB	38.6 MB

Table 10.2: Memory used by the javac compiler for the compilation of a simple class and a complex set of classes

sample was the set of the ConGen project's internal classes, which is about 170 classes.

The results are shown in *Table 10.2*. Although the values in the table are inaccurate, they give us a general view of how much memory *javac* consumes. Since the compilation of the element's code uses all libraries that are required for the ConGen framework, the required memory is probably closer to the values of the second column.

```
class Dummy{
  public Dummy() {
    System.out.println("This is dummy class");
  }
}
```

Listing 10.1: Dummy class compiled during *Java* memory consumption measurements

10.2.3 ConGen distribution build timing

In the last measurement, we focus on the comparison of the time needed for the initial build of the application in two cases. The first case is the existing ConGen framework and the second case is the optimized version including the full precompilation procedure. Although the initial build typically proceeds only once in design environment, it is involved in the precompilation approach, therefore we will present the values.

The test was performed on the optimized version of the ConGen framework. The first part was the original distribution build (launched as *ant dist*) and the second was the precompiled build (launched as *ant dist.precompile*).

The measurement was repeatedly run 20 times, the extreme values were filtered and the average times are shown in *Table 10.3*. The time needed for the precompilation build is significantly higher than the time needed for pure build. Through a more detailed examination of the precompilation process' composition we have found that the most of the time is spent in the Stratego program transforming templates into a compilable form. This is caused by the intensive utilization of external Stratego/XT tools from the program. However, the inefficiency is also

environment	ant dist	ant dist.precompile
Linux / Sun JDK 1.6.0.01	11.3 s	92.2 s

Table 10.3: Timing of the full initial build of ConGen in existing version and in version with the precompilation

multiplied by the number of descriptors in the element repository.

10.2.4 Measurements conclusion

The most important benefit in the proposed optimization is the fact that *javac* is no longer required at deployment-time. Besides, this omission saves at least 20MB of memory during the generation process. Moreover, the time needed for the compilation of the precompiled templates is significantly lower than the *javac* values.

10.3 Limitations of the proposed solution

Most of the limitations of the proposed solution were already mentioned in the previous text. In this section, we will summarize them.

- The main and most important drawback of the precompiled templates is that no code is validated within the compilation process. Particular instructions and all linked class names are only constructed without any validation whether the classes exists and is accessible or not. Though, all errors will show up as soon as the code is run.
- One of the practical disadvantages is the time that is needed for the initial build of the entire ConGen framework. Since the design-time is not a critical point of the project lifetime, we do not see this as a major problem.
- As we explained, a call of the constructor given by a meta-variable is restricted only to an assignment statement and has to specify the exact description (set of parameter types) of the constructor. For this purpose, the new feature was introduced in ElLang-J syntax. All other commands and operations (except the assignment) using the result of the *new* allocation can be solved by using a temporary local variable.
- Another limitation is the use of the meta-variables and the dependent *set* meta-statements in two methods, especially in the case of the method template. Since the code of the meta-template is compiled twice, the designer of the

templates should be aware of this fact and thought out every usage of the meta-variable set.

- The next one is the restriction of using certain integer constants in the code, because these values are used for the indexing of the meta-variables during the precompilation. However, the restricted values go only from 987651234 increased by 1 and are bound to 512 values. This also means there can be only 512 meta-information in the template. On the other hand the value of 512 can be easily increased if needed.

- One of the important restrictions is using the typecasting described by a meta-variable. This kind of typecasting can only be used in assignment to a class' field, whose type is also described by the meta-variable. However, while developing the optimized solution, we have not found any other code pattern that strictly requires typecasting based on the meta-variable.

- The last mentionable limitation is related to the exceptions in the method templates. The method calls described by a meta-variable in the method templates are tagged by a dummy method calls during precompilation. Therefore, some of the `catch XXXException` constructions can become unreachable code. For this reason, we recommend to catch more general exceptions that are always reachable, e.g., *java.lang.Exception* and then check the real exception type within the code.

Chapter 11

Related works

This thesis participates in the research of the software connectors generation. It proposes a solution of how to generate the connector's elements code at deployment-time without using the *javac* compiler. One of the reasons for this optimization is the quite restrictive character of the deployment-time environment.

We can see an even stronger restriction on timing, memory and additional resources, when we look at the run-time environment, where the connectors pursue their activities. As we described in *Chapter 1*, in certain situations in the connector system the new reference can appear while the connector is already running and the reference can be of the type which the connector is not ready for. In such a situation the run-time generation of the new connector's element can be an option.

11.1 Connectors generated at run-time

There are several existing connector generation approaches. Unfortunately, most of them prepare the connectors at design-time. For example, the Openwings system [14] introduced in the first chapter, is one them. In this system, all the connectors part are set up based on the special interfaces added into components at design-time.

At the beginning of the text, we have also mentioned one of the simplest solutions of the connector generation, which is *Java* RMI used in EJB components. It realizes the component interconnection by generating stubs and skeletons that lie at the opposite end of the connection link. The stubs and the skeletons can be generated at run-time according to the application's current need. However, the stubs and the skeletons are not based on any template system as the connector elements in the existing ConGen project. ConGen is based on more complex model, thus the simple RMI approach does not gives us any significant idea for involvement in ConGen's future extensions.

One representative of the component systems is *Fractal* [6]. The concept of

the system is very similar to the concept that ConGen is based on. However, *Fractal* does not manage any template system. Rather, its components consists of the controller and the content, where the controller has the responsibility for internal and external connections of the sub-components contained in the content part. The variability of the connector is given by the set of the internal and the external interfaces of the controller. The exact semantics of the overall connector configuration is determined by a wrapping component that encloses all the components related to the configuration. An important fact for us is that *Fractal* provides a way of modifying the connector at run-time. It is possible to stop a component, e.g., the server component, suspend calls from all the other components, unbind the component from the connector, then bind its replacement, e.g., new server component, and then restart the connector again.

11.2 Precompilation as a way of making faster compilation

A major aspect appearing in this thesis is the precompilation. The word precompilation is used quite often in software engineering. However, in some situations it is used with slightly different meanings.

The first kind is a preparation of the source code for the compiler. Typically, the grammar of the source code language is extended with a set of meta-information used for example for omission of the code parts (e.g., `#define` and `#ifdef` in *C/C++/C#*) or embedded information of another language (e.g., embedded SQL in *C++* language processed by the *Pro*C/C++* precompiler).

Second kind of precompilation is the meaning we use for our templates. This means that certain code is precompiled into the fashion that will significantly decrease time and/or memory requirements at the time when the compilation is really needed. Typically, this approach is used in the area of web applications (e.g., the precompilation of JSP [24] or the precompilation of ASP [1]).

From a certain point of view every virtual machine using JIT uses the precompilation approach, for example, JVM using JIT. To compile native code directly from *Java* source code would be too expensive. But if the *Java* code is first compiled into a bytecode, then we have a binary template to build the native code instructions, which is much cheaper in the sense of time and memory.

Chapter 12

Conclusion and future work

12.1 Summary of work

The main goal of this thesis is to optimize the compilation step in the generation process of the connector's element. We use the technique of bytecode manipulation to create the class file. In the existing ConGen the class file is produced by using the *javac* compiler. For building up the class file in our solution, we use the template in the precompiled ElLang-J-BC language form as an input. The ElLang-J-BC template is created during the distribution build which takes place at design-time of the ConGen framework.

The important fact in the template precompilation is that the syntax of the ElLang-J language is fully preserved in the ElLang-J-BC language. Therefore, the existing source code generator written in Stratego can be used without modification. Precompiled bytecode is stored as a set of *Java* method calls where all instruction parameters, *Java* strings, integers and floats are. This approach also makes the template well readable for debugging purposes.

Since the approach uses the existing templates written in ElLang-J as the input, the strongest benefit of the existing template system is also preserved.

In general, we can say that the particular goals listed at the beginning of this thesis are met in our proposed solution. As we proved in *Chapter 10*, the time needed for the generation of the elements' classes has been significantly decreased. The need for additional resources need in the form of the *Java* compiler was also reduced. The prototype implementation of the optimized solution is the part of the thesis, too.

The last question to complete the goals is whether the proposed optimization is reusable for an architecture other than only the *Java* architecture or not. The answer depends on the concrete architecture very much. We can generally say that every language with its syntax and a compiled form of its code, which will allow us

to tag the meta-information in source code and to recognize the tag pattern later in the decompilation step, is suitable for our precompilation approach.

In the first chapter, we also mentioned the possible generation of the connector's part at run-time. The run-time environment, compared to the environment of a template and connector designer is very restrictive. Hence, our optimization makes one step closer to producing code at run-time.

Nevertheless, the precompilation of the templates also has certain limitations and brings some slight restrictions for a template designer. Fortunately, these restrictions do not impact on the overall architecture of the template system and can be easily got around.

12.2 Prototype implementation

The part of this thesis is the prototype implementation of the precompilation approach. The solution extends the Stratego/XT version of the connector generator (ConGen) presented in [32]. The proposed extension adds the full precompilation step into the build process of the ConGen distribution framework as well as the replacement of the *javac* compiler. Both of these new features are fully optional and their utilization depends only on the usage of the different ConGen build target.

The current version of ConGen's source code was used as the starting point for the development of our solution. The modified implementation can be found in the SVN repository on the site `http://aiya.ms.mff.cuni.cz/var/svn/congen/`.

12.3 Future work

As we mentioned in the text several times, the particular drawback in the generation of the connector elements is that the programs generated by the Stratego compiler are platform-dependent. The *strc* compiler of the Stratego/XT bundle transforms the Stratego program into the *C* language, which is then compiled into the binary program. Using a Stratego compiler producing *Java* code instead of *C* would be the solution of this problem. Such a compiler can be found in [10]. However, the project was stopped in an unstable state some time ago, so it would have to be reopened for its usage first.

The generation of the connectors' parts at run-time is another possible field of a research in the connector generation and subsequently in the ConGen project. The optimization proposed in this thesis could be a good starting point for the fast generation of the code.

The last obvious future work in the connectors' area, specifically in ConGen, is the extension of the generation engine also for other languages. The work [32] has

already laid out the future extension of the template system for the *C++* and *C#* languages. Other research on re-using the optimization from this thesis for the new languages could follow, as well.

Bibliography

[1] ASP.NET Web Site Precompilation, http://msdn2.microsoft.com/en-us/library/ms228015.aspx.

[2] Apache Software Foundation, The BCEL Project manual, http://jakarta.apache.org/bcel/manual.html.

[3] Microsoft Corp., Component Object Model, http://www.microsoft.com/com/.

[4] Object Management Group, CORBA component model, http://www.omg.org/technology/documents/formal/components.htm.

[5] Sun Microsystems, Inc., Enterprise JavaBeans, http://java.sun.com/products/ejb/.

[6] Fractal component model, http://fractal.objectweb.org/.

[7] Helix community, Helix DNA, http://helixcommunity.org/.

[8] Intrinsyc Software, J-Integra, http://j-integra.intrinsyc.com/.

[9] Janino - An Embedded Java compiler, http://www.janino.net/index.html.

[10] Java Backend for the Stratego Compiler, http://nix.cs.uu.nl/dist/stratego/strc-java-0.1pre12067/.

[11] Jikes, http://jikes.sourceforge.net/.

[12] JNBridge, http://www.jnbridge.com/.

[13] JORAM: Java (TM) Open Reliable Asynchronous Messaging, http://joram.objectweb.org/.

[14] General Dynamics C4 Systems, Openwings, http://www.openwings.org/.

[15] SOFA component system, http://sofa.objectweb.org/.

[16] Stratego/XT, http://www.stratego-language.org/.

[17] Sun Microsystems, Inc., Sun Java System Application Server Platform Edition 8.1 2005Q1 Administration Guide, Monitoring Components and Services, `http://docs.sun.com/source/819-0076/monitor.html`.

[18] Sun Microsystems, Inc., Sun News Archive, Press Releases, 2000/05, `http://www.sun.com/smi/Press/sunflash/2000-05/sunflash.20000508.3.xml`.

[19] Sun Microsystems, Inc., Java Remote Method Specification, `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.htm`, 2001.

[20] Sun Microsystems, Inc., Java Message Service Specification, `http://java.sun.com/products/jms/docs.html`, 2002.

[21] Sun Microsystems, Inc., JavaSpaces Service Specification, `http://www.sun.com/software/jini/specs/jini1.2html/js-title.html`, April 2002.

[22] Object Management Group, Common Object Request Broker Architecture: Core Specification, version 3.0.3, `http://www.omg.org/docs/formal/04-03-12.pdf`, 2004.

[23] Object Management Group, CORBA Messaging, `http://www.omg.org/docs/formal/04-03-09.ps`, 2004.

[24] Bailey, B. L., and Smet, A. D. U.S. Patent 7093243, Software mechanism for efficient compiling and loading of java server pages (JSPs).

[25] Balek, D. *Connectors in Software Architectures*. PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, March 2002.

[26] Bulej, L., and Bures, T. A connector model suitable for automatic generation of connectors. Tech. Rep. 2003/1, Dep. of SW Engineering, Charles University, Prague, January 2003.

[27] Bures, T. *Generating Connectors for Homogeneous and Heterogeneous Deployment*. PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2006.

[28] E. Visser, E. Scannerless generalized-LR parsing. Tech. Rep. P9707, Programming Research Group, University of Amsterdam, July 1997.

[29] 'Eric Bruneton, Lenglet, R., and Coupaye, T. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journ'ees Composants 2002 : Syst'emes 'a composants adaptables et extensibles (Adaptable and extensible component systems)* (Nov. 2002).

[30] Gosling, J., Joy, B., Steele, G., and Bracha, G. *The Java^{TM} Language Specification, Third Edition.* Prentice Hall, June 2005.

[31] Lindholm, T., and Yellin, F. *The Java^{TM} Virtual Machine Specification, Second Edition.* Addison-Wesley, Apr. 1999.

[32] Malohlava, M. Using Stratego/XT for Generation of Software Connectors. Master's thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Januar 2007.

[33] Venners, B. *Inside the Java^{TM} Virtual Machine, Second Edition.* McGraw-Hill, 2000.

Appendix A

Template code examples

A.1 Original ElLang-J template

The existing version of ConGen [32] includes code templates written in the ElLang-J language. The following example shows a real template for the composite element using most of the meta-language features given by ElLang-J.

```
package ${package};

import org.objectweb.dsrg.connector.*;
/*
 * Meta declaration of connector compound_default
 */
element compound_default {
  /**
   * The instances of subelements of this composite connector element.
   */
  protected Element[] subElements;

  /**
   * Remote targets to which the remote ports are bounded to.
   */
  protected RemoteRefBundle[] remoteTargetRefs;

  /**
   * Instance of dock connector manager which is responsible for connector
   * units within the dock.
   */
  protected final org.objectweb.dsrg.connector.mgr.DockConnectorManager dcm;

  /**
   * Reference to parent connector unit.
   */
  protected final ConnectorUnit parentUnit;

  protected boolean isTopLevel;

  /**
   * Reconfiguration request handler.
   */
  protected org.objectweb.dsrg.connector.ReconfigurationHandler reconfigurationHandler;

  /*********************************************************************
   * CONSTRUCTORS
   *********************************************************************/
  public ${classname}(ConnectorUnit parentUnit, boolean isTopLevel)
      throws ElementLinkException {
    this.parentUnit = parentUnit;
    this.isTopLevel = isTopLevel;

    dcm = org.objectweb.dsrg.connector.mgr.
          DockConnectorManagerHelper.getDockConnectorManager();

    initializeArchitecture();
```

```
}

/**
 * When this method is called?
 */
void initializeArchitecture() throws ElementLinkException {
  subElements = new Element[${elements.element#count}];

  try {
    /* create sub-elements */
    $set i = 0$
    $foreach(ELEMENT in ${elements.element})$
      subElements[${i}] = new ${ELEMENT.class as (org.objectweb.dsrg.connector.ConnectorUnit,boolean)}
         (parentUnit, false);
      /* remember ELEMENT index in array */
      $set el[ELEMENT.name] = i$
      $set i = i + 1$
    $end$

    /* 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);
  }

  /* boundings to remote references - this need number of remote ports */
  remoteTargetRefs = new RemoteRefBundle[${ports.port(type=REMOTE)#count}];

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

  distributeReconfigurationHandler();
}

/**
 * Distributes a reference to this element as a reference to
 * reconfiguration handler to all reconfigurable subelements.
 */
protected final void distributeReconfigurationHandler () {
  for (Element elem : subElements) {
    if (elem instanceof ReconfigurableElement) {
      ((ReconfigurableElement) elem).
        setElReconfigurationHandler (this);
    }
  }
}


/**********************************************************************
 * Element Methods
 **********************************************************************/
public String getElementInfo(String indent) {
  StringBuilder result = new StringBuilder();

  result.append(indent + "Implementation: ${implementation}\n");

  $foreach(ELEMENT in ${elements.element})$
    result.append(indent + "Sub-element: ${ELEMENT.name} \n");
    result.append(subElements[ ${el[ELEMENT.name]} ].getElementInfo(indent + "        "));
  $end$

  return result.toString();
}

/**********************************************************************
 * ElementLocalServer Methods
 **********************************************************************/

implements interface ElementLocalServer {
  /**
   * Look for element port.
   *
   * Only one element port is returned.
   */
  public Object lookupElPort(String portName) throws ElementLinkException {
```

```
    /*
     * this part needs number of PROVIDED ports and right
     * index to subElements[]
     */
    $rforeach(PORT in ${ports.port(type=PROVIDED)} )$
      if ("${PORT.name}".equals(portName)) {
        Object result = ((ElementLocalServer) subElements[${el[PORT.boundedTo.element.name]}])
          .lookupElPort("${PORT.boundedTo.port}");
        if (isTopLevel) {
          dcm.reregisterConnectorUnitReference(parentUnit, portName, result);
        }
        return result;
      } else $recpoint$
    $final$
      throw new  ElementLinkException("Invalid provided port '"+portName+"'.");
    $end$
  }
}

/*********************************************************************
 * ElementLocalClient Methods
 ********************************************************************/

implements interface ElementLocalClient {
  /*
   * this needs number of REQUIRED ports and find right index of
   * selected element
   */
  public void bindElPort(String portName, Object target) throws ElementLinkException {
    $rforeach(PORT in ${ports.port(type=REQUIRED)})$
      if ("${PORT.name}".equals(portName)) {

        $foreach(BELEMENT in ${PORT.boundedTo})$
          ((ElementLocalClient) subElements[${el[BELEMENT.element.name]}]).bindElPort("${BELEMENT.port}", target);
        $end$

      } else $recpoint$
    $final$
      throw new ElementLinkException("Invalid required port '"+portName+"'.");
    $end$
  }

  public void unbindElPort(String portName) throws ElementLinkException {
    $rforeach(PORT in ${ports.port(type=REQUIRED)})$
      if ("${PORT.name}".equals(portName)) {
        $foreach(BELEMENT in ${PORT.boundedTo})$
         ((ElementLocalClient) subElements[${el[BELEMENT.element.name]}]).unbindElPort("${BELEMENT.port}");
        $end$
      } else
        $recpoint$
    $final$
      throw new ElementLinkException("Invalid required port '"+portName+"'.");
    $end$
  }
}
/*********************************************************************
 * ElementRemoteServer Methods
 ********************************************************************/
implements interface ElementRemoteServer {

public RemoteRefBundle lookupElRemotePort(String portName) throws ElementLinkException {
    RemoteRefBundle result = new RemoteRefBundle();
    // here we need number of REMOTE ports and right binding
    $rforeach(PORT in ${ports.port(type=REMOTE)})$
      if ("${PORT.name}".equals(portName)) {

        $foreach(BELEMENT in ${PORT.boundedTo})$
          if (subElements[${el[BELEMENT.element.name]}] instanceof ElementRemoteServer) {
            result.addRefBundle( ( (ElementRemoteServer) subElements[${el[BELEMENT.element.name]}] )
              .lookupElRemotePort("${BELEMENT.port}") );
          }
        $end$
      } else $recpoint$
    $final$
      throw new ElementLinkException("Invalid remote port '"+portName+"'.");
    $end$
    return result;
  }

  public String[] listElRemotePorts() {
    String[] result = new String[${ports.port(type=REMOTE)#count}];
    $set i = 0$
    $foreach(REMOTE_PORT in ${ports.port(type=REMOTE)})$
      result[${i}] = "${REMOTE_PORT.name}";
```

```
        $set i = i + 1$
      $end$
      return result;
    }
  }

  /*********************************************************************
   * ElementRemoteClient Methods
   *********************************************************************/
  implements interface ElementRemoteClient {

    public void bindElRemotePort(String portName, RemoteRefBundle refBundle) throws ElementLinkException {
      // here we need number of REMOTE ports and right binding
      $rforeach(PORT in ${ports.port(type=REMOTE)})$
        if ("${PORT.name}".equals(portName)) {
          $foreach(BELEMENT in ${PORT.boundedTo})$
            if (subElements[${el[BELEMENT.element.name]}] instanceof ElementRemoteClient) {
              ((ElementRemoteClient) subElements[${el[BELEMENT.element.name]}])
                .bindElRemotePort("${BELEMENT.port}", refBundle);
            }
          $end$
          remoteTargetRefs[${ref[PORT.name]}] = refBundle;
        } else $recpoint$
      $final$
        throw new ElementLinkException("Invalid remote port '"+portName+"'.");
      $end$
    }

    public void unbindElRemotePort(String portName) throws ElementLinkException {
      $rforeach(PORT in ${ports.port(type=REMOTE)})$
        if ("${PORT.name}".equals(portName)) {
          $foreach(BELEMENT in ${PORT.boundedTo})$
            if (subElements[${el[BELEMENT.element.name]}] instanceof ElementRemoteClient) {
              ((ElementRemoteClient) subElements[${el[BELEMENT.element.name]}]).unbindElRemotePort("${BELEMENT.port}");
            }
          $end$

          remoteTargetRefs[${ref[PORT.name]}] = null;
        } else $recpoint$
      $final$
        throw new ElementLinkException("Invalid remote port '"+portName+"'.");
      $end$
    }

    public RemoteRefBundle getElRemoteTarget(String portName) throws ElementLinkException {
      // here we need number of REMOTE ports and right index of port
      // into array remoteTargetRefs
      $rforeach(PORT in ${ports.port(type=REMOTE)})$
        if ("${PORT.name}".equals(portName)) {
          return remoteTargetRefs[${ref[PORT.name]}];
        } else $recpoint$
      $final$
        throw new ElementLinkException("Invalid remote port '"+portName+"'.");
      $end$
    }
  }

  /*********************************************************************
   * ReconfigurableElement Methods
   *********************************************************************/
  implements interface ReconfigurableElement {
    /**
     * Sets the reconfiguration handler.
     */
    public final void setElReconfigurationHandler (ReconfigurationHandler reconfigurationHandler) {
      this.reconfigurationHandler = reconfigurationHandler;
    }
  }

  /*********************************************************************
   * ReconfigurationHandler Methods
   *********************************************************************/
  implements interface ReconfigurationHandler {

    public void invalidateElPort (ElementLocalServer element, String portName) {
    }
    public void invalidateElRemotePort (ElementRemoteServer element, String portName)
      throws ReconfigurationException {
      throw new ReconfigurationException ("Reconfiguration not supported on remote ports, yet!");
    }
  }
}
```

A.2 Erlang-J template transformed for precompilation

Our proposed solution precompiles the ELLANG-J templates. The precompilation of the template file proceeds in several steps. In the first step the template is transformed into the compilable form by replacing meta-information with special *Java* statements. Following example shows the template above prepared for the *Java* compiler.

```
package org.objectweb.dsrg.congen.conrep.elements.templates.ellang;

import org.objectweb.dsrg.connector.*;

public class compound_default implements ElementLocalServer, ElementLocalClient, ElementRemoteServer,
  ElementRemoteClient, ReconfigurableElement, ReconfigurationHandler
{
  protected Element[] subElements;

  protected RemoteRefBundle[] remoteTargetRefs;

  protected final org.objectweb.dsrg.connector.mgr.DockConnectorManager dcm;

  protected final ConnectorUnit parentUnit;

  protected boolean isTopLevel;

  protected org.objectweb.dsrg.connector.ReconfigurationHandler reconfigurationHandler;

  public compound_default (ConnectorUnit parentUnit, boolean isTopLevel) throws ElementLinkException
  {
    this.parentUnit = parentUnit;
    this.isTopLevel = isTopLevel;
    dcm = org.objectweb.dsrg.connector.mgr.DockConnectorManagerHelper.getDockConnectorManager();
    initializeArchitecture();
  }

  void initializeArchitecture() throws ElementLinkException
  {
    subElements = new Element[987651235];
    try
    {
      org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651747);
      org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651748);
      subElements[987651237] = null;
      new org.objectweb.dsrg.congen.conrep.elements.templates.ellang.temp
        .MetaConstructor987651236(parentUnit, false);
      org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651749);
      org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651750);
      org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
      org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651751);
      org.objectweb.dsrg.congen.meta.MetaClass.replace_if(987651752);
      ((ElementLocalClient)subElements[987651238]).bindElPort("org.objectweb.dsrg.congen.meta.MetaString987651239;",
        ((ElementLocalServer)subElements[987651240]).lookupElPort("org.objectweb.dsrg.congen.meta.MetaString987651241;"));
      org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
      org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
    }
    catch(Exception e)
    {
      throw new ElementLinkException(e);
    }
    remoteTargetRefs = new RemoteRefBundle[987651242];
    org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651747);
    org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651753);
    remoteTargetRefs[987651237] = null;
    org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651754);
    org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651750);
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
    distributeReconfigurationHandler();
  }

  protected final void distributeReconfigurationHandler()
  {
    for(Element elem : subElements)
    {
      if(elem instanceof ReconfigurableElement)
```

```
    {
      ((ReconfigurableElement)elem).setElReconfigurationHandler(this);
    }
  }
}

public String getElementInfo(String indent)
{
  StringBuilder result = new StringBuilder();
  result.append(indent + "Implementation: org.objectweb.dsrg.congen.meta.MetaString987651243;\n");
  org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651748);
  result.append(indent + "Sub-element: org.objectweb.dsrg.congen.meta.MetaString987651244; \n");
  result.append(subElements[987651245].getElementInfo(indent + "        "));
  org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  return result.toString();
}

public Object lookupElPort(String portName) throws ElementLinkException
{
  org.objectweb.dsrg.congen.meta.MetaClass.replace_rfor(987651755);
  if("org.objectweb.dsrg.congen.meta.MetaString987651246;".equals(portName))
  {
    Object result = ((ElementLocalServer)subElements[987651247])
      .lookupElPort("org.objectweb.dsrg.congen.meta.MetaString987651248;");
    if(isTopLevel)
    {
      dcm.reregisterConnectorUnitReference(parentUnit, portName, result);
    }
    if(true)
    {
      return result;
    }
  }
  else
    org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
  org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
  if(true)
  {
    throw new ElementLinkException("Invalid provided port '" + portName + "'.");
  }
  org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  return null;
}

public void bindElPort(String portName, Object target) throws ElementLinkException
{
  org.objectweb.dsrg.congen.meta.MetaClass.replace_rfor(987651756);
  if("org.objectweb.dsrg.congen.meta.MetaString987651246;".equals(portName))
  {
    org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651757);
    ((ElementLocalClient)subElements[987651249])
      .bindElPort("org.objectweb.dsrg.congen.meta.MetaString987651250;", target);
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  }
  else
    org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
  org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
  if(true)
  {
    throw new ElementLinkException("Invalid required port '" + portName + "'.");
  }
  org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  return;
}

public void unbindElPort(String portName) throws ElementLinkException
{
  org.objectweb.dsrg.congen.meta.MetaClass.replace_rfor(987651756);
  if("org.objectweb.dsrg.congen.meta.MetaString987651246;".equals(portName))
  {
    org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651757);
    ((ElementLocalClient)subElements[987651249])
      .unbindElPort("org.objectweb.dsrg.congen.meta.MetaString987651250;");
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  }
  else
    org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
  org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
  if(true)
  {
    throw new ElementLinkException("Invalid required port '" + portName + "'.");
  }
  org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  return;
```

```
}

public RemoteRefBundle lookupElRemotePort(String portName) throws ElementLinkException
{
  RemoteRefBundle result = new RemoteRefBundle();
  org.objectweb.dsrg.congen.meta.MetaClass.replace_rfor(987651758);
  if("org.objectweb.dsrg.congen.meta.MetaString987651246;".equals(portName))
  {
    org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651757);
    if(subElements[987651249] instanceof ElementRemoteServer)
    {
      result.addRefBundle(((ElementRemoteServer)subElements[987651249])
        .lookupElRemotePort("org.objectweb.dsrg.congen.meta.MetaString987651250;"));
    }
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  }
  else
    org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
  org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
  if(true)
  {
    throw new ElementLinkException("Invalid remote port '" + portName + "'.");
  }
  org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  if(true)
  {
    return result;
  }
  return null;
}

public String[] listElRemotePorts()
{
  String[] result = new String[987651242];
  org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651747);
  org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651753);
  result[987651237] = "org.objectweb.dsrg.congen.meta.MetaString987651251;";
  org.objectweb.dsrg.congen.meta.MetaClass.replace_set(987651750);
  org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  return result;
}

public void bindElRemotePort(String portName, RemoteRefBundle refBundle) throws ElementLinkException
{
  org.objectweb.dsrg.congen.meta.MetaClass.replace_rfor(987651758);
  if("org.objectweb.dsrg.congen.meta.MetaString987651246;".equals(portName))
  {
    org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651757);
    if(subElements[987651249] instanceof ElementRemoteClient)
    {
      ((ElementRemoteClient)subElements[987651249])
        .bindElRemotePort("org.objectweb.dsrg.congen.meta.MetaString987651250;", refBundle);
    }
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
    remoteTargetRefs[987651252] = refBundle;
  }
  else
    org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
  org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
  if(true)
  {
    throw new ElementLinkException("Invalid remote port '" + portName + "'.");
  }
  org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
  return;
}

public void unbindElRemotePort(String portName) throws ElementLinkException
{
  org.objectweb.dsrg.congen.meta.MetaClass.replace_rfor(987651758);
  if("org.objectweb.dsrg.congen.meta.MetaString987651246;".equals(portName))
  {
    org.objectweb.dsrg.congen.meta.MetaClass.replace_for(987651757);
    if(subElements[987651249] instanceof ElementRemoteClient)
    {
      ((ElementRemoteClient)subElements[987651249])
        .unbindElRemotePort("org.objectweb.dsrg.congen.meta.MetaString987651250;");
    }
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
    remoteTargetRefs[987651252] = null;
  }
  else
    org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
  org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
```

```
    if(true)
    {
      throw new ElementLinkException("Invalid remote port '" + portName + "'.");
    }
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
    return;
  }

  public RemoteRefBundle getElRemoteTarget(String portName) throws ElementLinkException
  {
    org.objectweb.dsrg.congen.meta.MetaClass.replace_rfor(987651758);
    if("org.objectweb.dsrg.congen.meta.MetaString987651246;".equals(portName))
    {
      if(true)
      {
        return remoteTargetRefs[987651252];
      }
    }
    else
      org.objectweb.dsrg.congen.meta.MetaClass.replace_recpoint();
    org.objectweb.dsrg.congen.meta.MetaClass.replace_final();
    if(true)
    {
      throw new ElementLinkException("Invalid remote port '" + portName + "'.");
    }
    org.objectweb.dsrg.congen.meta.MetaClass.replace_end();
    return null;
  }

  public final void setElReconfigurationHandler(ReconfigurationHandler reconfigurationHandler)
  {
    this.reconfigurationHandler = reconfigurationHandler;
  }

  public void invalidateElPort(ElementLocalServer element, String portName)
  { }

  public void invalidateElRemotePort(ElementRemoteServer element, String portName) throws ReconfigurationException
  {
    throw new ReconfigurationException("Reconfiguration not supported on remote ports, yet!");
  }
}
```

A.3 Meta-information from precompiled template

The previous section contains the template transformed for the compilation. The transformation replaces all meta-information occurrences with the *Java*code. Before the replacement the meta-information is collected and stored for the latter decompilation process. The collected information is stored in a XML file and the following example shows the one related to the template from *Section A.1*.

```
<?xml version="1.0" ?>

<result>
  <metavar id="987651235">${elements.element#count}</metavar>
  <metavar id="987651236">C ${ELEMENT.class}(org.objectweb.dsrg.connector.ConnectorUnit,boolean)</metavar>
  <metavar id="987651237">${i}</metavar>
  <metavar id="987651238">${el[BINDING.from.element.name]}</metavar>
  <metavar id="987651239">${BINDING.from.port}</metavar>
  <metavar id="987651240">${el[BINDING.to.element.name]}</metavar>
  <metavar id="987651241">${BINDING.to.port}</metavar>
  <metavar id="987651242">${ports.port(type=REMOTE)#count}</metavar>
  <metavar id="987651243">${implementation}</metavar>
  <metavar id="987651244">${ELEMENT.name}</metavar>
  <metavar id="987651245">${el[ELEMENT.name]}</metavar>
  <metavar id="987651246">${PORT.name}</metavar>
  <metavar id="987651247">${el[PORT.boundedTo.element.name]}</metavar>
  <metavar id="987651248">${PORT.boundedTo.port}</metavar>
  <metavar id="987651249">${el[BELEMENT.element.name]}</metavar>
  <metavar id="987651250">${BELEMENT.port}</metavar>
  <metavar id="987651251">${REMOTE_PORT.name}</metavar>
  <metavar id="987651252">${ref[PORT.name]}</metavar>
  <metastm id="987651747">$set i = 0$</metastm>
```

```
  <metastm id="987651748">$foreach (ELEMENT in ${elements.element})$</metastm>
  <metastm id="987651749">$set el[ELEMENT.name] = i$</metastm>
  <metastm id="987651750">$set i = i + 1$</metastm>
  <metastm id="987651751">$foreach (BINDING in ${bindings.binding})$</metastm>
  <metastm id="987651752">$if (BINDING.type == &quot;BINDING&quot;)$</metastm>
  <metastm id="987651753">$foreach (REMOTE_PORT in ${ports.port(type=REMOTE)})$</metastm>
  <metastm id="987651754">$set ref[REMOTE_PORT.name] = i$</metastm>
  <metastm id="987651755">$rforeach (PORT in ${ports.port(type=PROVIDED)})$</metastm>
  <metastm id="987651756">$rforeach (PORT in ${ports.port(type=REQUIRED)})$</metastm>
  <metastm id="987651757">$foreach (BELEMENT in ${PORT.boundedTo})$</metastm>
  <metastm id="987651758">$rforeach (PORT in ${ports.port(type=REMOTE)})$</metastm>
</result>
```

A.4 Precompiled Erlang-J-BC template

The following example shows the precompiled form of the template above.

```
/****** DO NOT EDIT *******/
package ${package};

element compound_default {


  /****** FIELDS ******/

  protected org.objectweb.dsrg.connector.Element[] subElements;

  protected org.objectweb.dsrg.connector.RemoteRefBundle[] remoteTargetRefs;

  protected final org.objectweb.dsrg.connector.mgr.DockConnectorManager dcm;

  protected final org.objectweb.dsrg.connector.ConnectorUnit parentUnit;

  protected boolean isTopLevel;

  protected org.objectweb.dsrg.connector.ReconfigurationHandler reconfigurationHandler;


  /****** METHODS ******/

  public CLASS_CONSTRUCTOR(org.objectweb.dsrg.connector.ConnectorUnit arg0, boolean arg1)
    throws org.objectweb.dsrg.connector.ElementLinkException {

    LABEL(0);
    LINE(20);
    ALOAD(0);
    INVOKESPECIAL("java.lang.Object","<init>","void()");
    LABEL(1);
    LINE(21);
    ALOAD(0);
    ALOAD(1);
    PUTFIELD("${classname}","parentUnit","org.objectweb.dsrg.connector.ConnectorUnit");
    LABEL(2);
    LINE(22);
    ALOAD(0);
    ILOAD(2);
    PUTFIELD("${classname}","isTopLevel","boolean");
    LABEL(3);
    LINE(23);
    ALOAD(0);
    INVOKESTATIC("org.objectweb.dsrg.connector.mgr.DockConnectorManagerHelper","getDockConnectorManager",
      "org.objectweb.dsrg.connector.mgr.DockConnectorManager()");
    PUTFIELD("${classname}","dcm","org.objectweb.dsrg.connector.mgr.DockConnectorManager");
    LABEL(4);
    LINE(24);
    ALOAD(0);
    INVOKEVIRTUAL("${classname}","initializeArchitecture","void()");
    LABEL(5);
    LINE(25);
    RETURN();
  }


  void initializeArchitecture() throws org.objectweb.dsrg.connector.ElementLinkException {

    LABEL(0);
    LINE(29);
```

```
ALOAD(0);
LDC(I(${elements.element#count}));
ANEWARRAY("org.objectweb.dsrg.connector.Element");
PUTFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
TRY_CATCH(1,2,3,"java.lang.Exception");
LABEL(1);
LINE(32);
  $set i = 0$
LABEL(4);
LINE(33);
  $foreach (ELEMENT in ${elements.element})$
LABEL(5);
LINE(34);
ALOAD(0);
GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
LDC(I(${i}));
LABEL(6);
LINE(35);
NEW("${ELEMENT.class}");
DUP();
ALOAD(0);
GETFIELD("${classname}","parentUnit","org.objectweb.dsrg.connector.ConnectorUnit");
ICONST_0();
INVOKESPECIAL("${ELEMENT.class}","<init>","void(org.objectweb.dsrg.connector.ConnectorUnit,boolean)");
AASTORE();
LABEL(7);
LINE(36);
  $set el[ELEMENT.name] = i$
LABEL(8);
LINE(37);
  $set i = i + 1$
LABEL(9);
LINE(38);
  $end$
LABEL(10);
LINE(39);
  $foreach (BINDING in ${bindings.binding})$
LABEL(11);
LINE(40);
  $if (BINDING.type == "BINDING")$
LABEL(12);
LINE(41);
ALOAD(0);
GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
LDC(I(${el[BINDING.from.element.name]}));
AALOAD();
CHECKCAST("org.objectweb.dsrg.connector.ElementLocalClient");
LDC(S("${BINDING.from.port}"));
ALOAD(0);
GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
LDC(I(${el[BINDING.to.element.name]}));
AALOAD();
CHECKCAST("org.objectweb.dsrg.connector.ElementLocalServer");
LDC(S("${BINDING.to.port}"));
INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementLocalServer","lookupElPort",
  "java.lang.Object(java.lang.String)");
INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementLocalClient","bindElPort",
  "void(java.lang.String,java.lang.Object)");
LABEL(13);
LINE(42);
  $end$
LABEL(14);
LINE(43);
  $end$
LABEL(2);
LINE(48);
GOTO(15);
LABEL(3);
LINE(45);
ASTORE(1);
LABEL(16);
LINE(47);
NEW("org.objectweb.dsrg.connector.ElementLinkException");
DUP();
ALOAD(1);
INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.Throwable)");
ATHROW();
LABEL(15);
LINE(49);
ALOAD(0);
LDC(I(${ports.port(type=REMOTE)#count}));
ANEWARRAY("org.objectweb.dsrg.connector.RemoteRefBundle");
PUTFIELD("${classname}","remoteTargetRefs","org.objectweb.dsrg.connector.RemoteRefBundle[]");
LABEL(17);
```

```
  LINE(50);
    $set i = 0$
  LABEL(18);
  LINE(51);
    $foreach (REMOTE_PORT in ${ports.port(type=REMOTE)})$
  LABEL(19);
  LINE(52);
  ALOAD(0);
  GETFIELD("${classname}","remoteTargetRefs","org.objectweb.dsrg.connector.RemoteRefBundle[]");
  LDC(I(${i}));
  ACONST_NULL();
  AASTORE();
  LABEL(20);
  LINE(53);
    $set ref[REMOTE_PORT.name] = i$
  LABEL(21);
  LINE(54);
    $set i = i + 1$
  LABEL(22);
  LINE(55);
    $end$
  LABEL(23);
  LINE(56);
  ALOAD(0);
  INVOKEVIRTUAL("${classname}","distributeReconfigurationHandler","void()");
  LABEL(24);
  LINE(57);
  RETURN();
}

protected final void distributeReconfigurationHandler() {

  LABEL(0);
  LINE(61);
  ALOAD(0);
  GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
  ASTORE(1);
  ALOAD(1);
  ARRAYLENGTH();
  ISTORE(2);
  ICONST_0();
  ISTORE(3);
  LABEL(1);
  ILOAD(3);
  ILOAD(2);
  IF_ICMPGE(2);
  ALOAD(1);
  ILOAD(3);
  AALOAD();
  ASTORE(4);
  LABEL(3);
  LINE(63);
  ALOAD(4);
  INSTANCEOF("org.objectweb.dsrg.connector.ReconfigurableElement");
  IFEQ(4);
  LABEL(5);
  LINE(65);
  ALOAD(4);
  CHECKCAST("org.objectweb.dsrg.connector.ReconfigurableElement");
  ALOAD(0);
  INVOKEINTERFACE("org.objectweb.dsrg.connector.ReconfigurableElement","setElReconfigurationHandler",
    "void(org.objectweb.dsrg.connector.ReconfigurationHandler)");
  LABEL(4);
  LINE(61);
  IINC(3,1);
  GOTO(1);
  LABEL(2);
  LINE(68);
  RETURN();
}

public java.lang.String getElementInfo(java.lang.String arg0) {

  LABEL(0);
  LINE(72);
  NEW("java.lang.StringBuilder");
  DUP();
  INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
  ASTORE(2);
  LABEL(1);
  LINE(73);
  ALOAD(2);
```

```
  NEW("java.lang.StringBuilder");
  DUP();
  INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
  ALOAD(1);
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  LDC(S("Implementation: ${implementation}\n"));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  POP();
  LABEL(2);
  LINE(74);
    $foreach (ELEMENT in ${elements.element})$
  LABEL(3);
  LINE(75);
  ALOAD(2);
  NEW("java.lang.StringBuilder");
  DUP();
  INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
  ALOAD(1);
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  LDC(S("Sub-element: ${ELEMENT.name} \n"));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  POP();
  LABEL(4);
  LINE(76);
  ALOAD(2);
  ALOAD(0);
  GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
  LDC(I(${el[ELEMENT.name]}));
  AALOAD();
  NEW("java.lang.StringBuilder");
  DUP();
  INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
  ALOAD(1);
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  LDC(S("        "));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
  INVOKEINTERFACE("org.objectweb.dsrg.connector.Element","getElementInfo","java.lang.String(java.lang.String)");
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  POP();
  LABEL(5);
  LINE(77);
    $end$
  LABEL(6);
  LINE(78);
  ALOAD(2);
  INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
  ARETURN();
}

public java.lang.Object lookupElPort(java.lang.String arg0) throws org.objectweb.dsrg.connector.ElementLinkException {

  LABEL(0);
  LINE(83);
    $rforeach (PORT in ${ports.port(type=PROVIDED)})$
  LABEL(1);
  LINE(84);
  LDC(S("${PORT.name}"));
  ALOAD(1);
  INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
  IFEQ(2);
  LABEL(3);
  LINE(86);
  ALOAD(0);
  GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
  LDC(I(${el[PORT.boundedTo.element.name]}));
  AALOAD();
  CHECKCAST("org.objectweb.dsrg.connector.ElementLocalServer");
  LDC(S("${PORT.boundedTo.port}"));
  INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementLocalServer","lookupElPort",
    "java.lang.Object(java.lang.String)");
  ASTORE(2);
  LABEL(4);
  LINE(87);
  ALOAD(0);
  GETFIELD("${classname}","isTopLevel","boolean");
  IFEQ(5);
  LABEL(6);
  LINE(89);
```

```
  ALOAD(0);
  GETFIELD("${classname}","dcm","org.objectweb.dsrg.connector.mgr.DockConnectorManager");
  ALOAD(0);
  GETFIELD("${classname}","parentUnit","org.objectweb.dsrg.connector.ConnectorUnit");
  ALOAD(1);
  ALOAD(2);
  INVOKEINTERFACE("org.objectweb.dsrg.connector.mgr.DockConnectorManager","reregisterConnectorUnitReference",
    "void(org.objectweb.dsrg.connector.ConnectorUnit,java.lang.String,java.lang.Object)");
  LABEL(5);
  LINE(93);
  ALOAD(2);
  ARETURN();
  LABEL(2);
  LINE(97);
    $recpoint$
  LABEL(7);
  LINE(98);
    $final$
  LABEL(8);
  LINE(101);
  NEW("org.objectweb.dsrg.connector.ElementLinkException");
  DUP();
  NEW("java.lang.StringBuilder");
  DUP();
  INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
  LDC(S("Invalid␣provided␣port␣'"));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  ALOAD(1);
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  LDC(S("'."));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
  INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.String)");
  ATHROW();
    $end$
}


public void bindElPort(java.lang.String arg0, java.lang.Object arg1)
  throws org.objectweb.dsrg.connector.ElementLinkException {

  LABEL(0);
  LINE(109);
    $rforeach (PORT in ${ports.port(type=REQUIRED)})$
  LABEL(1);
  LINE(110);
  LDC(S("${PORT.name}"));
  ALOAD(1);
  INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
  IFEQ(2);
  LABEL(3);
  LINE(112);
    $foreach (BELEMENT in ${PORT.boundedTo})$
  LABEL(4);
  LINE(113);
  ALOAD(0);
  GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
  LDC(I(${el[BELEMENT.element.name]}));
  AALOAD();
  CHECKCAST("org.objectweb.dsrg.connector.ElementLocalClient");
  LDC(S("${BELEMENT.port}"));
  ALOAD(2);
  INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementLocalClient","bindElPort",
    "void(java.lang.String,java.lang.Object)");
  LABEL(5);
  LINE(114);
    $end$
  GOTO(6);
  LABEL(2);
  LINE(117);
    $recpoint$
  LABEL(6);
  LINE(118);
    $final$
  LABEL(7);
  LINE(121);
  NEW("org.objectweb.dsrg.connector.ElementLinkException");
  DUP();
  NEW("java.lang.StringBuilder");
  DUP();
  INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
  LDC(S("Invalid␣required␣port␣'"));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  ALOAD(1);
```

```
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    LDC(S("'."));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
    INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.String)");
    ATHROW();
      $end$
}

public void unbindElPort(java.lang.String arg0) throws org.objectweb.dsrg.connector.ElementLinkException {

    LABEL(0);
    LINE(129);
      $rforeach (PORT in ${ports.port(type=REQUIRED)})$
    LABEL(1);
    LINE(130);
    LDC(S("${PORT.name}"));
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
    IFEQ(2);
    LABEL(3);
    LINE(132);
      $foreach (BELEMENT in ${PORT.boundedTo})$
    LABEL(4);
    LINE(133);
    ALOAD(0);
    GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
    LDC(I(${el[BELEMENT.element.name]}));
    AALOAD();
    CHECKCAST("org.objectweb.dsrg.connector.ElementLocalClient");
    LDC(S("${BELEMENT.port}"));
    INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementLocalClient","unbindElPort","void(java.lang.String)");
    LABEL(5);
    LINE(134);
      $end$
    GOTO(6);
    LABEL(2);
    LINE(137);
      $recpoint$
    LABEL(6);
    LINE(138);
      $final$
    LABEL(7);
    LINE(141);
    NEW("org.objectweb.dsrg.connector.ElementLinkException");
    DUP();
    NEW("java.lang.StringBuilder");
    DUP();
    INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
    LDC(S("Invalid required port '"));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    LDC(S("'."));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
    INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.String)");
    ATHROW();
      $end$
}

public org.objectweb.dsrg.connector.RemoteRefBundle lookupElRemotePort(java.lang.String arg0)
  throws org.objectweb.dsrg.connector.ElementLinkException {

    LABEL(0);
    LINE(149);
    NEW("org.objectweb.dsrg.connector.RemoteRefBundle");
    DUP();
    INVOKESPECIAL("org.objectweb.dsrg.connector.RemoteRefBundle","<init>","void()");
    ASTORE(2);
    LABEL(1);
    LINE(150);
      $rforeach (PORT in ${ports.port(type=REMOTE)})$
    LABEL(2);
    LINE(151);
    LDC(S("${PORT.name}"));
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
    IFEQ(3);
    LABEL(4);
    LINE(153);
      $foreach (BELEMENT in ${PORT.boundedTo})$
```

```
  LABEL(5);
  LINE(154);
  ALOAD(0);
  GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
  LDC(I(${el[BELEMENT.element.name]}));
  AALOAD();
  INSTANCEOF("org.objectweb.dsrg.connector.ElementRemoteServer");
  IFEQ(6);
  LABEL(7);
  LINE(156);
  ALOAD(2);
  ALOAD(0);
  GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
  LDC(I(${el[BELEMENT.element.name]}));
  AALOAD();
  CHECKCAST("org.objectweb.dsrg.connector.ElementRemoteServer");
  LDC(S("${BELEMENT.port}"));
  INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementRemoteServer","lookupElRemotePort",
    "org.objectweb.dsrg.connector.RemoteRefBundle(java.lang.String)");
  INVOKEVIRTUAL("org.objectweb.dsrg.connector.RemoteRefBundle","addRefBundle",
    "void(org.objectweb.dsrg.connector.RemoteRefBundle)");
  LABEL(6);
  LINE(158);
    $end$
  GOTO(8);
  LABEL(3);
  LINE(161);
    $recpoint$
  LABEL(8);
  LINE(162);
    $final$
  LABEL(9);
  LINE(165);
  NEW("org.objectweb.dsrg.connector.ElementLinkException");
  DUP();
  NEW("java.lang.StringBuilder");
  DUP();
  INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
  LDC(S("Invalid␣remote␣port␣'"));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  ALOAD(1);
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  LDC(S("'."));
  INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
  INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
  INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.String)");
  ATHROW();
    $end$
}

public java.lang.String[] listElRemotePorts() {

  LABEL(0);
  LINE(177);
  LDC(I(${ports.port(type=REMOTE)#count}));
  ANEWARRAY("java.lang.String");
  ASTORE(1);
  LABEL(1);
  LINE(178);
    $set i = 0$
  LABEL(2);
  LINE(179);
    $foreach (REMOTE_PORT in ${ports.port(type=REMOTE)})$
  LABEL(3);
  LINE(180);
  ALOAD(1);
  LDC(I(${i}));
  LDC(S("${REMOTE_PORT.name}"));
  AASTORE();
  LABEL(4);
  LINE(181);
    $set i = i + 1$
  LABEL(5);
  LINE(182);
    $end$
  LABEL(6);
  LINE(183);
  ALOAD(1);
  ARETURN();
}

public void bindElRemotePort(java.lang.String arg0, org.objectweb.dsrg.connector.RemoteRefBundle arg1)
```

```
    throws org.objectweb.dsrg.connector.ElementLinkException {

    LABEL(0);
    LINE(188);
      $rforeach (PORT in ${ports.port(type=REMOTE)})$
    LABEL(1);
    LINE(189);
    LDC(S("${PORT.name}"));
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
    IFEQ(2);
    LABEL(3);
    LINE(191);
      $foreach (BELEMENT in ${PORT.boundedTo})$
    LABEL(4);
    LINE(192);
    ALOAD(0);
    GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
    LDC(I(${el[BELEMENT.element.name]}));
    AALOAD();
    INSTANCEOF("org.objectweb.dsrg.connector.ElementRemoteClient");
    IFEQ(5);
    LABEL(6);
    LINE(194);
    ALOAD(0);
    GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
    LDC(I(${el[BELEMENT.element.name]}));
    AALOAD();
    CHECKCAST("org.objectweb.dsrg.connector.ElementRemoteClient");
    LDC(S("${BELEMENT.port}"));
    ALOAD(2);
    INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementRemoteClient","bindElRemotePort",
      "void(java.lang.String,org.objectweb.dsrg.connector.RemoteRefBundle)");
    LABEL(5);
    LINE(196);
      $end$
    LABEL(7);
    LINE(197);
    ALOAD(0);
    GETFIELD("${classname}","remoteTargetRefs","org.objectweb.dsrg.connector.RemoteRefBundle[]");
    LDC(I(${ref[PORT.name]}));
    ALOAD(2);
    AASTORE();
    GOTO(8);
    LABEL(2);
    LINE(200);
      $recpoint$
    LABEL(8);
    LINE(201);
      $final$
    LABEL(9);
    LINE(204);
    NEW("org.objectweb.dsrg.connector.ElementLinkException");
    DUP();
    NEW("java.lang.StringBuilder");
    DUP();
    INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
    LDC(S("Invalid remote port '"));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    LDC(S("'."));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
    INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.String)");
    ATHROW();
      $end$
}

public void unbindElRemotePort(java.lang.String arg0) throws org.objectweb.dsrg.connector.ElementLinkException {

    LABEL(0);
    LINE(212);
      $rforeach (PORT in ${ports.port(type=REMOTE)})$
    LABEL(1);
    LINE(213);
    LDC(S("${PORT.name}"));
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
    IFEQ(2);
    LABEL(3);
    LINE(215);
      $foreach (BELEMENT in ${PORT.boundedTo})$
```

```
    LABEL(4);
    LINE(216);
    ALOAD(0);
    GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
    LDC(I(${el[BELEMENT.element.name]}));
    AALOAD();
    INSTANCEOF("org.objectweb.dsrg.connector.ElementRemoteClient");
    IFEQ(5);
    LABEL(6);
    LINE(218);
    ALOAD(0);
    GETFIELD("${classname}","subElements","org.objectweb.dsrg.connector.Element[]");
    LDC(I(${el[BELEMENT.element.name]}));
    AALOAD();
    CHECKCAST("org.objectweb.dsrg.connector.ElementRemoteClient");
    LDC(S("${BELEMENT.port}"));
    INVOKEINTERFACE("org.objectweb.dsrg.connector.ElementRemoteClient","unbindElRemotePort",
      "void(java.lang.String)");
    LABEL(5);
    LINE(220);
      $end$
    LABEL(7);
    LINE(221);
    ALOAD(0);
    GETFIELD("${classname}","remoteTargetRefs","org.objectweb.dsrg.connector.RemoteRefBundle[]");
    LDC(I(${ref[PORT.name]}));
    ACONST_NULL();
    AASTORE();
    GOTO(8);
    LABEL(2);
    LINE(224);
      $recpoint$
    LABEL(8);
    LINE(225);
      $final$
    LABEL(9);
    LINE(228);
    NEW("org.objectweb.dsrg.connector.ElementLinkException");
    DUP();
    NEW("java.lang.StringBuilder");
    DUP();
    INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
    LDC(S("Invalid remote port '"));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    LDC(S("'."));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
    INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.String)");
    ATHROW();
      $end$
  }


  public org.objectweb.dsrg.connector.RemoteRefBundle getElRemoteTarget(java.lang.String arg0)
    throws org.objectweb.dsrg.connector.ElementLinkException {

    LABEL(0);
    LINE(236);
      $rforeach (PORT in ${ports.port(type=REMOTE)})$
    LABEL(1);
    LINE(237);
    LDC(S("${PORT.name}"));
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.String","equals","boolean(java.lang.Object)");
    IFEQ(2);
    LABEL(3);
    LINE(241);
    ALOAD(0);
    GETFIELD("${classname}","remoteTargetRefs","org.objectweb.dsrg.connector.RemoteRefBundle[]");
    LDC(I(${ref[PORT.name]}));
    AALOAD();
    ARETURN();
    LABEL(2);
    LINE(245);
      $recpoint$
    LABEL(4);
    LINE(246);
      $final$
    LABEL(5);
    LINE(249);
    NEW("org.objectweb.dsrg.connector.ElementLinkException");
    DUP();
```

```
    NEW("java.lang.StringBuilder");
    DUP();
    INVOKESPECIAL("java.lang.StringBuilder","<init>","void()");
    LDC(S("Invalid␣remote␣port␣'"));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    ALOAD(1);
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    LDC(S("'."));
    INVOKEVIRTUAL("java.lang.StringBuilder","append","java.lang.StringBuilder(java.lang.String)");
    INVOKEVIRTUAL("java.lang.StringBuilder","toString","java.lang.String()");
    INVOKESPECIAL("org.objectweb.dsrg.connector.ElementLinkException","<init>","void(java.lang.String)");
    ATHROW();
      $end$
  }


  public final void setElReconfigurationHandler(org.objectweb.dsrg.connector.ReconfigurationHandler arg0) {

    LABEL(0);
    LINE(257);
    ALOAD(0);
    ALOAD(1);
    PUTFIELD("${classname}","reconfigurationHandler","org.objectweb.dsrg.connector.ReconfigurationHandler");
    LABEL(1);
    LINE(258);
    RETURN();
  }


  public void invalidateElPort(org.objectweb.dsrg.connector.ElementLocalServer arg0, java.lang.String arg1) {

    LABEL(0);
    LINE(261);
    RETURN();
  }


  public void invalidateElRemotePort(org.objectweb.dsrg.connector.ElementRemoteServer arg0, java.lang.String arg1)
    throws org.objectweb.dsrg.connector.ReconfigurationException {

    LABEL(0);
    LINE(265);
    NEW("org.objectweb.dsrg.connector.ReconfigurationException");
    DUP();
    LDC(S("Reconfiguration␣not␣supported␣on␣remote␣ports,␣yet!"));
    INVOKESPECIAL("org.objectweb.dsrg.connector.ReconfigurationException","<init>","void(java.lang.String)");
    ATHROW();
  }


  /****** METHOD TEMPLATES ******/

  /****** REMAINING INTERFACES ******/

  implements interface org.objectweb.dsrg.connector.ElementLocalServer {
  }

  implements interface org.objectweb.dsrg.connector.ElementLocalClient {
  }

  implements interface org.objectweb.dsrg.connector.ElementRemoteServer {
  }

  implements interface org.objectweb.dsrg.connector.ElementRemoteClient {
  }

  implements interface org.objectweb.dsrg.connector.ReconfigurableElement {
  }

  implements interface org.objectweb.dsrg.connector.ReconfigurationHandler {
  }

}
```

A.5 Original element descriptor

Element code templates are not the primary information about the element. The primary information is the element XML descriptor stored in the element repository. The descriptor defines actions that must be processed to generate the element. The following example is an original element descriptor using the template from *Section A.1*

```
<?xml version="1.0" encoding="UTF-8" ?>

<element name="server_unit" type="rpc_server_unit" impl-class="ServerUnit">

  <architecture cost="0">
    <inst name="skeleton" type="skeleton" cardinality="multiple"/>
    <binding port1="line" element2="skeleton" port2="line"/>
    <binding element1="skeleton" port1="call" port2="call"/>
  </architecture>

  <nfp-declarations>
    <nfp-mapping name="communication_style" value="method_invocation"/>
  </nfp-declarations>

  <script>
    <command action="jimpl">
        <param name="generator" value="org.objectweb.dsrg.congen.elemgen.generators.stratego.StrategoGenerator"/>
        <param name="class" value="ServerUnit"/>
        <param name="template" value="ellang/compound_default.ellang" />
    </command>

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

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

  </script>

</element>
```

A.6 Redirected element descriptor

As soon as the ELLANG-J template are precompiled the descriptor is redirected to the precompiled template and the compiler action is changed from the original *javac* to the new *javacbc*. The following example shows the redirected version of the descriptor from *Section A.5*.

```
<?xml version="1.0" encoding="UTF-8"?>
<element name="server_unit" type="rpc_server_unit" impl-class="ServerUnit">

  <architecture cost="0">
    <inst name="skeleton" type="skeleton" cardinality="multiple" />
    <binding port1="line" element2="skeleton" port2="line" />
    <binding element1="skeleton" port1="call" port2="call" />
  </architecture>

  <nfp-declarations>
    <nfp-mapping name="communication_style" value="method_invocation" />
  </nfp-declarations>

  <script>
    <command action="jimpl">
        <param name="generator" value="org.objectweb.dsrg.congen.elemgen.generators.stratego.StrategoGenerator" />
        <param name="class" value="ServerUnit" />
        <param name="template" value="ellang/compound_default.ellang.bc" />
    </command>
```

```
    <command action="javacbc">
        <param name="class" value="ServerUnit" />
    </command>

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

  </script>

</element>
```

Appendix B

Content of attached CD-ROM

This work is accompanied by the CD-ROM containing source code and binaries of the prototype implementation. The content of the CD-ROM is structured as follows:

`/README.TXT`
: Description of the CD-ROM content.

`/thesis/`
: Electronic version of this document.

`/src/java/`
: Newly added or modified source code of the *Java* part of the prototype implementation.

`/src/stratego/`
: Newly added or modified source code of the Stratego part of the prototype implementation.

`/src/prototype.congen.full/`
: Full prototype implementation.

`/examples/`
: Examples of the precompiled templates written in the ElLang-J-BC language.

`/prereqs/`
: Software prerequisites of the prototype implementation: StrategoXT v0.16, ATerm library v2.4.2, SDF2-bundle v2.3.3, Java-front v0.8.