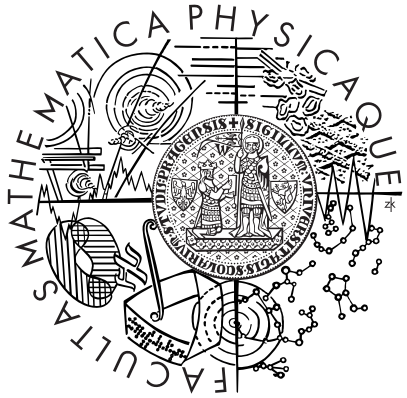
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jan Raszyk

Connector Generation Process Enhancement

Department of Software Engineering

Supervisor: RNDr. Michal Malohlava
Study Program: Computer Science, Software Systems

I would like to thank my supervisor Michal Malohlava for his invaluable suggestions, patience and help with writing this master thesis. Also, his master thesis was the initial resource that helped me understand the topic of my work. Special thanks goes to David O'Shea for his help with proper grammar and wording of this thesis. I would also like to thank my family for their support.

Prohlašuji, že jsem svou diplomovou práci napsal 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 3, 2010

Jan Raszyk

Contents

Název práce: Connector Generation Process Enhancement
Autor: Jan Raszyk
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Michal Malohlava
e-mail vedoucího: malohlava@d3s.mff.cuni.cz

Abstrakt: Softwarové konektory poskytují možnost, jak modelovat a realizovat propojení komponent v komponentových systémech. Jednou z výhod konektorů je, že sestavení jejich programového kódu lze provádět automatizovaně - generováním kódu.

Softwarové konektory také reflektují různé další extra-funkční požadavky (např. logování, bezpečnost, adaptaci, měření). Generátor konektorů by proto měl umožňovat snadnou rozšiřitelnost funkčnosti generovaných konektorů. Použitelnost generátoru konektorů také ovlivňuje jeho platformová přenositelnost.

Tato práce si klade za cíl vylepšit generování kódu softwarových konektorů v komponentovém systému SOFA. Na základě reálných požadavků tohoto systému rozšiřuje funkcionalitu existujícího generátoru konektorů za účelem rozšíření množiny konektorů, jež lze generovat. Rovněž práce řeší platformovou přenositelnost generátoru konektorů založeného na nástroji StrategoXT.

Klíčová slova: generování kódu, softwarové konektory, DSL, StrategoXT, Spoofax, šablony metod

Title: Connector Generation Process Enhancement
Author: Jan Raszyk
Department: Department of Software Engineering
Supervisor: RNDr. Michal Malohlava
Supervisor's e-mail address: malohlava@d3s.mff.cuni.cz

Abstract: Software connectors offer a way to model and realize connections between components in component systems. To create the source code for the software connectors, it is beneficial to use automated code generation.

Software connectors also reflect various non-functional properties (e.g. logging, security, adaptation, measurements). Therefore the connector generator should support simple extensibility of the functionality of generated connectors. The usability of the connector generator is also affected by its platform portability.

This thesis aims to enhance the generation of source code for software connectors in the SOFA component system. It enhances the functionality of the existing connector generator to extend the set of generated connectors. The enhancements are motivated by real requirements of the SOFA system. The work also solves the platform portability issues of the existing connector generator based on the StrategoXT tool.

Keywords: code generation, software connectors, DSL, StrategoXT, Spoofax, method templates

Chapter 1

Introduction

As the usage of computers has spread globally, they have evolved dramatically, becoming more sophisticated and they are now being utilized across a wide spectrum of specialized areas. As a result, software has steadily become more and more complex, which has necessitated, that previously created software be re-used in new software systems on an ever increasing basis.

There is a need to help software developers to be able to construct more complex software and reuse existing code in a systematic manner. One of the possible approaches to deal with both of these tasks is Component-based software engineering (CBSE). As it emphasizes *separation of concerns*, it helps to create independent smaller pieces of software called *components* and then connect them together.

Components have their defined *interfaces* through which they communicate with other components. The only thing that a component reveals about itself to other components is just the definition of the interfaces. Interfaces in component-based systems are of two types - *required* and *provided*.

Nowadays, complex software is typically used in distributed environments, where one system runs on multiple machines, with components of the software being distributed across the machines, while they communicate with each other via the computer network. The code responsible for connecting components and modeling the connection between them is called a *software connector*.

Connectors are responsible for transparently connecting components, taking care of network code, communication style and the internal bindings of interfaces. They reflect and internally handle all sorts of requirements imposed on the bindings between components.

The basic structure of connectors' code is often the same for similar connectors and writing all of the connector code manually can be a lengthy and error prone task. Therefore it is suitable, for the connector code to be generated in an automated manner. The part of component system which handles this task is typically called a *connector generator.*

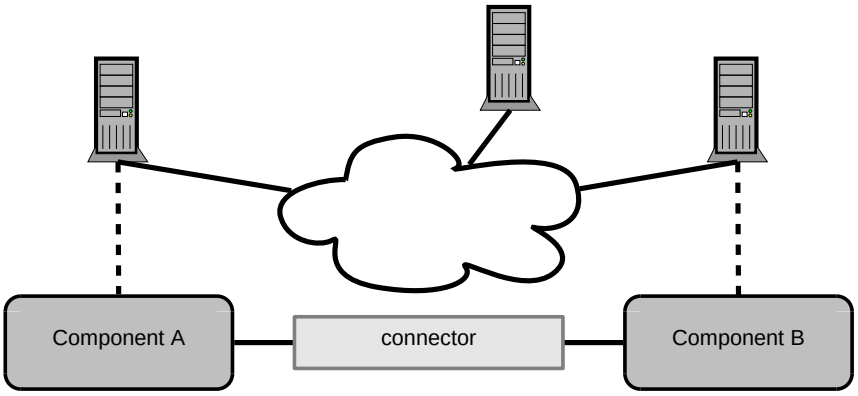


Figure 1.1: An example of a component-based system with distributed components and a connector connecting them

The automated generation process starts with the specification of the properties for the connection established by the connector. The result of this process, is a complete connector's source code, which can then be added to the code base of the whole distributed software system, and then instantiated.

1.1 Motivation

Software connectors are an important part of mature component systems [37]. When a component system employs explicit software connectors, many of its features are implemented within the connectors. The connector generator must provide ways to deal with requirements imposed on the connectors.

For example, if the component system aims to support multiple communication styles (e.g. method invocation, messaging, streaming) or other non-functional properties, the connectors have to reflect these requirements. This means that the connector generator has to contain support to prepare such connectors.

Also, if the component system aims to support multiple platforms (Linux, Windows), the connector generator should be cross-platform too. There are many tools available for code generation. These tools may bring platform and other dependencies into the component system, if they don't natively support all of the platforms supported by the component system.

There are many types of component systems which employ the components and connectors model. The connector generator in the SOFA [14] component system is tightly connected to the StrategoXT [18] program transformations toolset. This toolset is used as the basis for the generation of the connectors' code.

1.2 Goals

The goal of this thesis, is to enhance the existing connector generator in the SOFA component system. The StrategoXT toolset used to generate code for the connectors brings platform dependencies into the component system. Also, during the usage of the existing connector generator, new requirements have been identified that require extension of the functionality of the generator. New functionality should extend the set of generated connectors.

The first goal is to provide a cross-platform solution for connector preparation. This thesis aims to remove platform dependencies inherited by some parts of the StrategoXT toolset and reuse an existing implementation of the connector generator in the Stratego language.

The second goal is to extend the functionality of the connector generator. By enriching the code generation support of the generator, it should be made possible to generate a larger set of software connector types required by the SOFA component system.

The thesis should conclude by explaining the connector generation process in the context of classical development process. It should explain the requirements put on the roles in the development of a component-based system with connectors.

1.3 Structure of the thesis

Chapter 2 of this thesis provides an introduction into the topics of connectors and their generation. It describes the existing connector generator which this thesis extends. There is a brief introduction into the Stratego framework and aspects important for connector generation. Finally, it also describes the existing template language used in the connector generation process.

Chapter 3 summarizes and analyses the goals of this thesis and proposes an approach which should be taken when fulfilling them.

The solution is presented in Chapter 4. This chapter describes the enhancements in the connector generator and in the connector template language.

In Chapter 5 the connector generation process is described as a part of the classical development process. Different roles in the development process are described together with their tasks and required domain knowledge brought by components, connectors and connectors generation.

Finally, Chapter 6 evaluates the enhancements brought by this thesis, Chapter 7 discusses related works and Chapter 8 brings a summary of this thesis and discusses ideas for future work.

Chapter 2

Connector Generation

Software connectors are entities which connect components and their interfaces in component-based systems. They model the communication and interaction between components during the design time of a system. They capture the list of requirements imposed on the connection between components and their interfaces.

In the system development process, it is during the deployment phase when connectors get instantiated, as described in [28]. In this stage the specification of connector requirements gets transformed into actual source code. This code is then instantiated and used in the complete running software system.

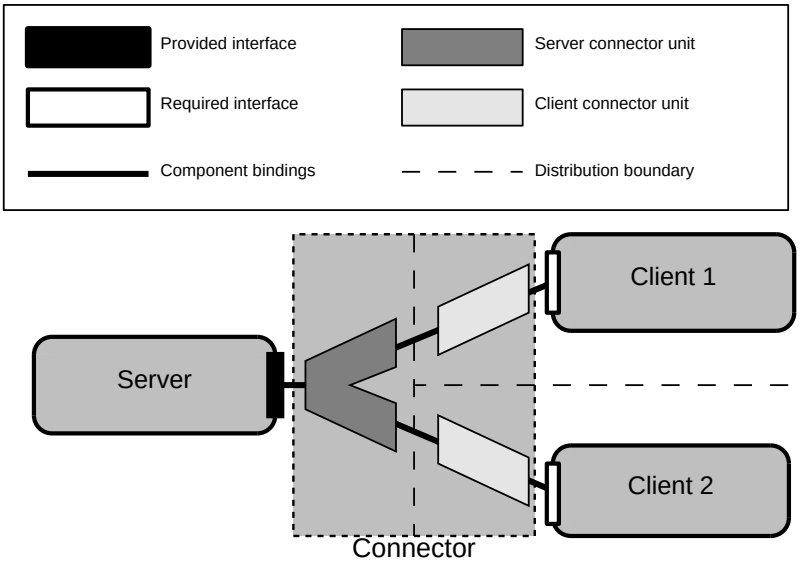


Figure 2.1: An example of a component system with distributed components and connector units connecting them

It is during the deployment stage, when it is known which components of the system will be placed on different machines and will therefore run in separated address spaces. This also affects the process of connector preparation. For example

connectors that bind components from different address spaces may have to employ network communication.

At system run-time, software connectors implement the communication between components and encapsulate the middleware code. In this phase they are represented by the concrete source code necessary for network communication, delegation of calls, bindings of interfaces. They also implement additional required functionality chosen by the designer of the system.

Software connectors have a predefined set of both functional and non-functional requirements placed on them. The component system offers the system developer a list of supported features and properties from which they can choose as appropriate when connecting components of the system.

Even in very simple systems with components, there is at least a few connectors to be prepared. Writing all of the source code by hand can be a lengthy and error-prone process.

These are some of the main arguments for why connectors are prepared in the deployment stage using automated code generation. The following sections will describe this automated process. At the beginning there is the specification of requirements imposed on the connectors and at the end of the process is the prepared connectors' code ready to be launched as a part of a complete component-based software system.

2.1 Generation Process

As proposed in the work [29], the input of the connector generation process should be a feature-oriented list of requirements. Such requirements should be familiar to the user of the component system, who does not need to be familiar with connectors and their architectures.

Therefore this initial model of the generated connector, is a list of more abstract prerequisites that the generator should convert into the final source code for the connector. Such requirements can be

- *Communication style* - for example one of: method invocation, messaging, streaming, blackboard.
- *Non-functional properties* - for example security, monitoring, logging, call context modification, performance measurements.
- *Deployment* - for example specification of whether the connector should connect components which run in separated address spaces.

To overcome the large gap between abstract system developer's requirements

and concrete connector's source code, the generation process is divided into two phases. These phases are handled by the following parts of the connector generator:

- *Architecture Resolver.* Handles the first phase. Here the abstract requirements are transformed into a more detailed technical specification of the connector. The following sub-section discusses this phase briefly.
- *Element Generator.* Handles the second phase, where the actual source code of the connector gets generated. This thesis aims to improve this phase and the rest of the thesis will talk only about this phase.

2.1.1 Architecture Resolver

The input to the architecture resolver is an abstract specification of requirements imposed on the connector. This specification is based on the requirements specified by the developer of the target system. The input specification is called *high-level connector specification.*

The same functionality of a connector, for example logging, network communication and others, can be typically handled by code that is almost the same in every connector. Therefore it is possible to categorize the requirements and the code that implements them into sets where every category has a rough code structure implementing requirements from a given set.

It is even possible to use the component-based approach for the generation of the connectors themselves. Each connector can be composed of smaller parts — components. These are called *connector elements*, where each element will implement specific functionality or requirements.

The goal of the architecture resolver is then to determine which elements the final connector should be made up of. The high-level connector specification needs to be transformed into a *low-level connector configuration* which is represented by a tree-like structure of connector elements and bindings between them.

The approach that is used to transform high-level connector specification into low-level connector configuration in the architecture resolver is described in detail in the work [29]. Basically the architecture resolver builds a knowledge database of available connector elements and then uses backtracking to find the best configuration of the elements that satisfies the high-level specification.

2.1.2 Element Generator

The element generator is a part of the connector generator, where more technical low-level connector configuration gets transformed into connector's source code units.

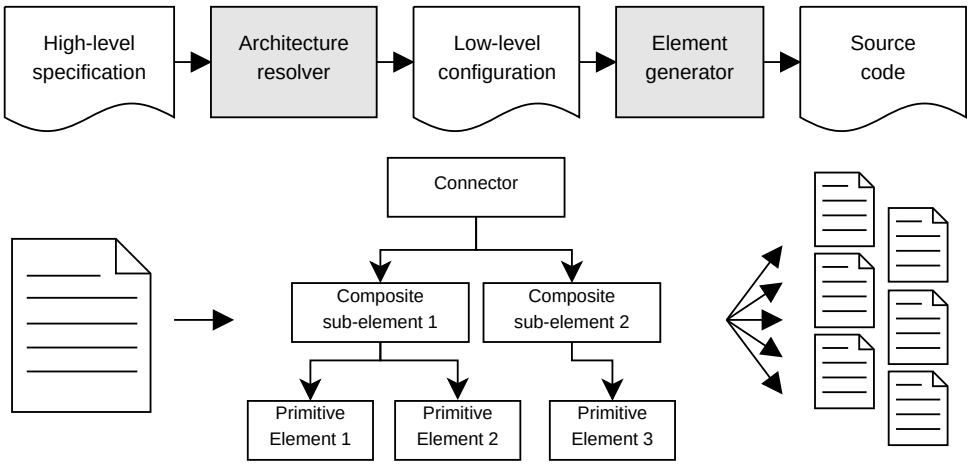


Figure 2.2: Main phases of the connector generation process

Low-level connector configuration, which is the input to the element generator, contains a list of connector elements which should be generated and the bindings between them. As with components, connector elements are connected to each other via interfaces. Interfaces of connector elements are called *ports*.

Ports can be of three types: *provided* and *required*, like with components, and also *remote*. Remote ports are used when a connection across different address spaces is established by the connector.

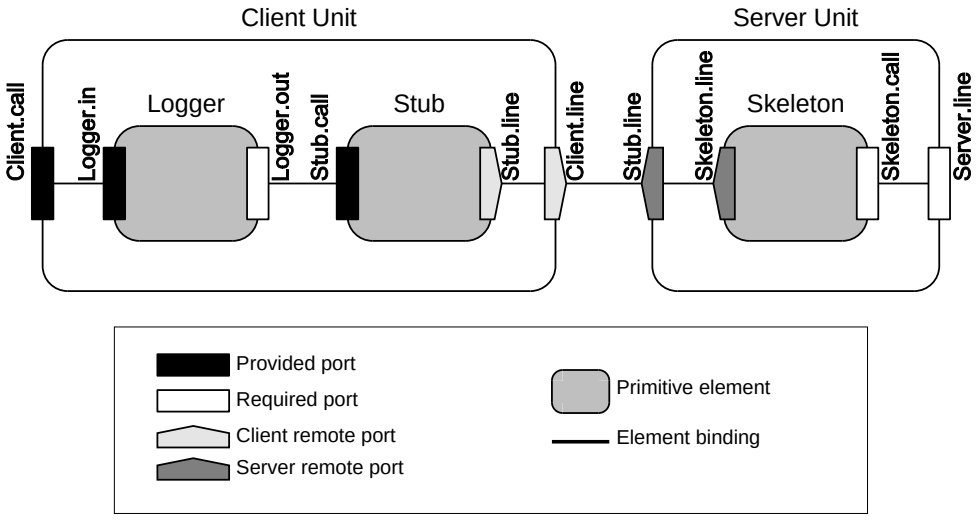


Figure 2.3: Example of a connector configuration with connector elements and their ports

Connector elements themselves are either *composite* or *primitive*:

- *Composite element* contains one or more inner elements and bindings between them. Ports of the composite element are delegated to ports of some of its inner elements.

- *Primitive element* contains source code which implements functionality, a given connector element type should provide.

The task of the element generator is to provide source code for each connector element prescribed in the low-level connector configuration. The configuration contains for each element a description of its:

- type,
- ports, their types and signatures,
- bindings of ports,
- identification of resulting source code unit like package name and class name.

Each connector element type has similar code in every instance. Therefore source code for each element type can be represented by a template with special areas, which denote places into which specific information should be inserted from the low-level connector configuration. The main role of the element generator therefore is to load the appropriate template for a given connector element type and substitute specific code into the denoted areas of the code.

2.1.3 Existing Element Generator

This thesis extends an element generator introduced in [34]. The element generator introduced in [29] was enhanced by removing the necessity to edit both simple code templates and also separate algorithms which would substitute specific code into them.

This is done by employing a new *domain-specific language* for the connector element templates, which allows the capturing of both the static part of the code and the code which needs to be generated and substituted into specific locations, based on the low-level connector configuration. The simple algorithms that substitute code into templates are then part of the template itself.

The element generator described in [34] is based on StrategoXT [18] program transformations toolset. The following chapters will introduce this toolset and the main features that are used in the element generator.

2.2 Stratego Program Transformation Language

StrategoXT is a language and toolset for program transformations [18]. It provides facilities for syntax definition, parsing, transformation and pretty-printing of languages. These are available either as standalone executables or part of the standard library available to be used when writing transformation programs.

2.2.1 Syntax Definition

StrategoXT uses Syntax Definition Formalism (SDF) as the basis for the specification of a new syntax. The formalism supports modular definition of the grammar. In every module it is possible to import other modules and define both lexical and context-free syntax of the newly defined language.

The common approach taken is by defining language sentence productions using strings of non-terminals and terminals. A string is generated by starting with the start non-terminal and repeatedly replacing non-terminal symbols according to the productions, until a string of terminal symbols is reached. [18]

2.2.2 Abstract Syntax

The grammar definition is used for parsing strings in the defined language. The parser could output the complete parse tree which would contain all the non-terminal nodes, from all productions, that were used while parsing the input. However most of such nodes would be unnecessary for further manipulation with the parsed input.

For this, the SDF allows for extending the production rules with *constructor annotations*. The new nodes that are specified in the constructor annotations are then used in the result of parsing. The resulting tree is then called *Abstract Syntax Tree* (AST).

```
module Expression
imports Operators

exports
  sorts Id IntConst
  lexical syntax
    [\ \t\n]  -> LAYOUT
    [a-zA-Z]+ -> Id
    [0-9]+    -> IntConst

exports
  context-free syntax
    Id          -> Exp {cons("Var")}
    IntConst    -> Exp {cons("Int")}
    "(" Exp ")" -> Exp {bracket}
```

Listing 2.1: An example of a short SDF module defining basic lexical and syntactical non-terminals

2.2.3 Terms and Term Rewriting

The abstract syntax tree is the result of parsing an input using given grammar. The resulting tree is represented in ATerm format inside of Stratego tools. This format has the form `Constructor(sub-term, ...)`. It also supports lists using `[` and `]`

notation. The code `4 + f(5 * x)` can be represented as:

```
Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))
```

The most important part of the StrategoXT platform is term rewriting, also called program transformation. It is possible to create programs which operate on ATerms. It is possible to parse an input, for example: source code of some language, and then manipulate it in a program written in Stratego.

For this, Stratego offers *rules* and *strategies*.

- *Rule* is an application of a rewrite command of the form `N: A -> B` meaning that rule labeled N rewrites term constructor A to constructor B. Such a rule will be applied when the current term matches the term A and the result of the application will be term B.
- *Strategy* is a defined way to traverse the AST and apply rules or other strategies during the traversal. Standard Stratego library defines a couple of basic and derived strategies which allow the traversing of the tree from its leafs up to the root or vice versa, apply selected rules to all children of the current root term, repeatedly or conditionally apply a rule (or strategy) to the current root term, and many others. Strategy can also be a sequence of other strategies.

There is a concept of *current term* when invoking transformation strategies. The current term is always an implicit parameter that is passed to every strategy. This is done to avoid the necessity to always pass the item that is being transformed into every strategy the transformation is composed of.

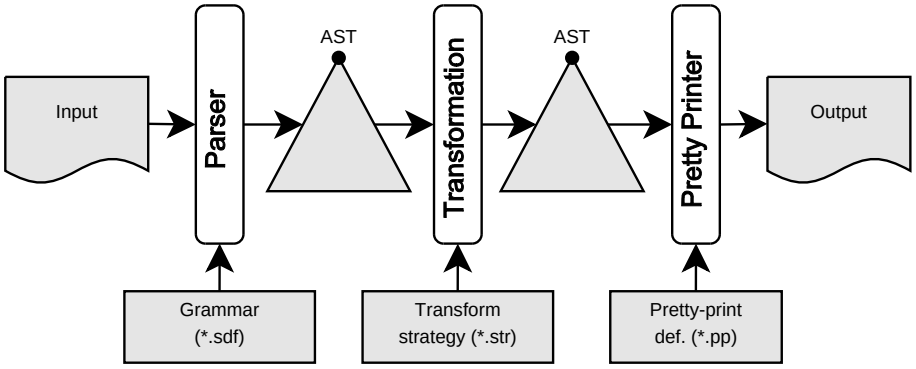


Figure 2.4: Example of usage of the Stratego toolset. Input is parsed, transformed in the form of AST and then printed into output.

2.2.4 Dynamic Rules

Standard rewriting rules in Stratego do not reflect the context in which they are used. The author of the transformation program defines the rule before the run-

time of the transformation and has to specify both sides of the rule. Such rules cannot reflect the current state of parsing.

Dynamic rules, on the other hand, offer a way to define rules during run-time and can contain the current parsing state in them. The final rule will be instantiated once the statement for its definition is met during the run-time of the transformation program. In both left and right sides of the rule it is possible to use patterns which will contain some information from the current parsing state.

Dynamic rules also provide a way to create and access global variables during the execution of the program transformation. By creating a dynamic rule which always returns a certain value at one place of the transformation and later invoking such rule to obtain the desired value, it's possible to transfer such values between different places of execution.

```
rules( GlobalVar := 100 )
...
localVar != <GlobalVar>
```

Dynamic rules allow for the performing of many operations directly on the source code of a program before the code is even compiled. For example constant propagation, variable renaming, function inlining, common subexpression elimination and others [26].

2.2.5 Language Composition

When designing a new language, developers often would like to start with some existing language and then only extend it with some additional desired features. It is more convenient, to insert new constructs into an existing language than to design a whole new language. Later during the compile stage it would be sufficient to just transform a program written in the mixed language back into the base language and use existing tools of the base language like compilers or virtual machines for further work.

StrategoXT offers a way to mix two languages together. It does so on the grammar level. In the syntax definition, it is possible to create a mixed module which imports grammars of both languages and defines the intersections between them. This technique was introduced in [25] and is called *MetaBorg*.

The mixed grammar approach makes non-terminals from both languages accessible and then allows for them to be combined. To avoid naming conflicts between non-terminals from two languages, Stratego offers concepts of renaming and SDF mix.

- Renaming allows the appending of a prefix or suffix to every non-terminal from a given language. Therefore for example: non-terminal Expr representing an

expression in the Java language can be become JavaExpr. Then the non-terminal for an expression in the extending language can still be named Expr and will not conflict with the Java non-terminal. For clarity it is better to also rename non-terminals from the extending language. This concept is being replaced by the following similar SDF mix concept.

- SDF mix is an approach similar to renaming. The non-terminals in selected syntax definitions are appended with a context variable. So similarly Expr from the Java language can become Expr[[JCTX]]. One of the differences is that the specific name of the context is defined when the mixed grammar is being imported into the mixed SDF module.

```
module MyJavaExtension

imports
  languages/java/JavaMix[JCTX]
  Foo[FOOCTX]

exports
  context-free syntax
    Expr[[FOOCTX]] -> Expr[[JCTX]] {avoid, cons("FromTargetLang")}
    Stm[[JCTX]] -> Statement[[FOOCTX]] {avoid, cons("ToTargetLang")}
```

Listing 2.2: An example of a grammar module which mixes together constructs of the Java and an extending language Foo.

2.2.6 XTC Repository

Stratego is a toolset for program transformations. The tools in the toolset are standalone binary executables that work with standard input and standard output. Every tool has its specific functionality. There are, for example, tools for parsing, pretty-printing, working with ATerms and others.

To easily access the full functionality of the tools, Stratego introduces the concept of the XTC repository. It is a database of Stratego tools and other programs added by the user into the database. This database is kept in a single file and can be accessed by a special Stratego tool or can be used from within the transformation.

There are strategies available, when writing program transformations, which invoke tools from the XTC repository. Such strategies will launch the tool as a process in the background and pass the current term to it as the standard input. The standard output of the tool can then be set as the new current term.

This however brings dependencies into the transformation program. It will be dependent on the Stratego tools native executables, which might not be available on all platforms. [16]

2.2.7 Stratego Compiler

Transformations written in Stratego are programs that operate on standard input and standard output. To be able to launch these programs, it is necessary to compile them into an executable form.

The approach in Stratego, is to transform the Stratego programs into an intermediate language. Then the compiler of the intermediate language can compile the final executable of the transformation program.

Original Stratego Compiler [17] compiles the transformations into the C language and lets the C compiler (typically `gcc`) compile the generated source code in C into a native executable.

Such executable will not be cross-platform portable and will depend on shared libraries of the operating system where the compilation was performed (e.g. `libc`). Calls to strategies from standard Stratego libraries will also create a dependency on shared Stratego libraries.

2.3 Connector Template Language

The work [34] aims to provide a connector template language that allows the capturing of everything that is needed to generate connector's code. Both the static parts of the code, and also dynamic parts which get generated after additional input data (the low-level connector configuration) is known.

Another goal is to define the grammar for the template language, to be able to run syntax checking and recognize some of the syntax errors before the actual final code of the connector gets generated.

The work uses Stratego and the MetaBorg method to achieve these goals. It implements a domain-specific language for connector templates. The resulting source code for the connectors is in the Java language. The base language for the MetaBorg method is Java. The Java language is then extended with constructs for generating connectors' code.

2.3.1 Syntax

The additional language constructs are first defined in the ElLang language (El = connector element, Lang = language). This language introduces basic template constructs like `if`, `foreach`, `set`, template meta-variables, importing and extending of other templates.

```
...
$set outports = 0$
$foreach (p in ${ports.port})$
```

```
  $if (p.name == "out")$
    $set outports = outports + 1$
  $end$
$end$
...
```

Listing 2.3: Example of the ElLang language and some of its constructs.

The ElLang language is then injected into the target language, which is the language in which the final code of a connector will be produced. In the case of the work [34] it is the Java language which is used.

The resulting language is called ElLang-J. There is one mixed module which imports grammar definitions of both ElLang and Java and defines the intersection of these languages by allowing the presence of ElLang constructs in specific places of the Java code.

For example ElLang statements (e.g. `if`, `foreach`) are allowed to be written in places where Java statements are expected. Therefore the template `foreach` construct can be written inside of Java code. And also Java statements are allowed to be written where ElLang statements are expected, for example in the body of the template `foreach` construct.

```
module ElLang-J

imports
  languages/java/JavaMix[JCX]
  ElLangMix[ECX]

exports
  context-free syntax
    ElementDec[[ECX]] -> TypeDec[[JCX]] {avoid, cons("FromTL")}
    TypeDec[[JCX]] -> ElementDec[[ECX]] {avoid, cons("ToTL")}

    Stm[[JCX]] -> Stm[[ECX]] {avoid, cons("ToTL") }
    Stm[[ECX]] -> Stm[[JCX]] {avoid, cons("FromTL") }
```

Listing 2.4: Part of the SDF mix module which defines the grammar of the ElLang-J language

2.3.2 Element Generation

The work [34] defines the syntax of the ElLang-J language and also the transformation from this language to Java. This is where the connector element's code gets generated.

The input of the Stratego transformation part is an *element descriptor*. It is a description of the connector element taken from the low-level connector configuration and also the description of which template represents the current element. The Connector generator prepares the descriptor and then launches the element generator in Stratego and passes the given element descriptor to its input.

Firstly the element generator invokes a strategy which calls an external tool from the XTC repository for parsing. This tool parses the element descriptor. When the tool finishes, the result of the parsing is returned back into the element generator and the parsed element descriptor is then stored as a global variable using a dynamic rule. The module responsible for this is called *the query module*. It is a general module for accessing any structured data, not just the element descriptor.

Then the path to the template is read from the descriptor and the element's template is parsed also via invocation of the external parse tool. The result, the abstract syntax tree of the parsing, then becomes the current term. What follows, are transformations of the parsed template in the ElLang-J language back into Java language. These transformations are described in the following section.

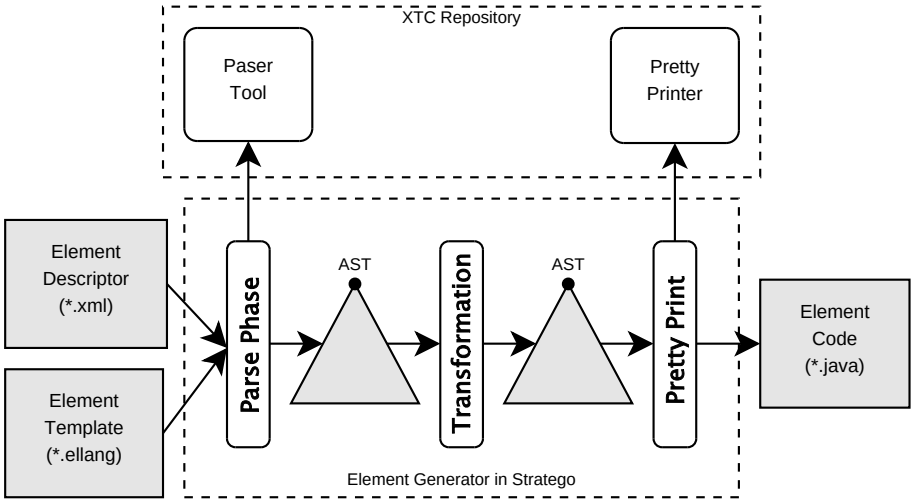


Figure 2.5: Schema of the process of connector element generation

Finally after the transformations are done, the abstract syntax tree of the transformed template contains only constructs from the Java programming language. The abstract syntax tree can now be printed as a unit of code by invoking an external tool from the XTC repository used for pretty-printing Java code.

2.3.3 Transformations

The goal of the transformation phase of the element generator is to transform the element template into the specific (Java) code of the connector element. The data from the element descriptor is being inserted into the template in this phase. ElLang constructs like `if` or `foreach` are expanded, imported and extended templates are inserted, meta-variable references are evaluated.

The whole process is motivated by a typical compilation process in compilers and is composed of the following phases:

- *Processing of extended and imported templates.* A template can extend some

part of another template, the contents of the extended template therefore needs to be inserted into the current one. Templates can also import another template and the contents of the imported template also needs to be inserted.

- *Template preparation* unifies constructs that can be present in multiple forms. For example the if-then-else statement can be missing the else branch. For further processing of the template it is easier to unify these statements into the same form.

- *Variables expansion* phase reads data from the element descriptor and substitutes it in the parts of the template which reference them.

- *Interface evaluation.* The template language allows for specifying that the implementing class of the connector element will implement another interface. The methods of the implemented interface then need to be inserted with some implementation. This is discussed in the following section.

- *Statements and expressions evaluation phase* processes template statements like `if`, `foreach` and `set`, also any expressions met during the traversal are evaluated, that is replaced by the resulting value of the given expression.

2.3.4 Interface Evaluation

Each connector element is implemented by a Java class. Therefore each connectors' template can specify that the resulting class should implement one or more interfaces. This is done by using a block `implements interface I { ... }`

The specific interface might not be known when the template of the connector element is specified. Therefore it should be possible to specify the implementation of all the implemented methods in some general way.

An example of such a situation is when a connector element should implement logging of method calls between components. The element then implements all methods of the interface that the components communicate through. The body of the methods logs the call and then passes the call on to further connector elements.

2.3.5 Method Templates

The possibility to specify a general implementation of all methods of any business interface is an important part of the connector element generation. It is the place in the template where the network code must reflect the business interfaces.

The work [34] introduces a concept of *method templates*. Inside of the *implements interface* block of the connector element it is possible to use a *method template*

block construct. In this block it is possible to specify a general implementation of every method of an interface.

Method template is a template of method's body. This body will be substituted, as a body of all the methods the given interface prescribes. Inside of the method template body, it is possible to use certain meta-variables, which will be filled with data specific to the given method, like method name or arguments:

- `${method.name}` - the name of the current method
- `${method.returnType}` - return type of the method
- `${method.declareReturnValue}` - declares a local temporary variable for storing the return value of the method
- `${method.returnVar}` - name of the temporary variable that stores the return value
- `${method.returnStm}` - generates a return statement where the temporary return variable will be returned
- `${method.variables}` - arguments of the method

```
package ${package};

import org.objectweb.dsrg.connector.*;

element console_log extends "primitive_default.ellang" {
  implements interface ${ports.port(name=in).signature} {
    method template {
      ${method.declareReturnValue}
        System.out.println(
          "Connector monitoring element: method -> ${method.name} < called");
        $if (method.returnVar) $
          ${method.returnVar} = this.target.${method.name}(${method.variables});
        $else$
          this.target.${method.name}(${method.variables});
        $end$

        //generates return statement if it is needed
        ${method.returnStm}
      }
  }
}
```

Listing 2.5: Example of method template for call logger connector element.

2.3.6 Integration Into SOFA Component System

The connector generator gets executed in the deployment phase [28] of a system. After the architecture resolver produces the low-level connector configuration, the

`GenerationManager` launches the `ElementGenerator` for each of the connector elements prescribed in the configuration.

The `ElementGenerator` object uses the following instructions to generate the connector element:

- Action script describing the steps required to generate a given element in a series of commands.
- Mapping of the commands to classes that implement them.

The action script can for example prescribe the following series of steps for the generation:

- A command to generate the source code for the given type of an element connector.
- A command to compile the code into binary (byte-code) form.
- A command to delete the generated source codes.

Every type of a connector element has its own command. For example an RMI Stub connector element is handled by the `RmiStubGenerator` command and a corresponding class.

To generate a connector element using the Stratego based element generator[1] there is a `StrategoGenerator` command available. This command is mapped to the `StrategoGenerator` class.

The `StrategoGenerator` generates the elements in a hierarchy, starting from the sub-elements that are lowest in the connector configuration tree. It prepares the element descriptor in a XML format and finally executes the Stratego generator binary via Java Native Interface (JNI [9]). In the arguments the element descriptor is passed to the Stratego generator.

[1]The initial implementation of the element generator was written in Java and is described in [29].

Chapter 3

Goals Revisited

This thesis aims to improve the connector generator introduced in the previous chapter. The generator uses the domain specific language to define templates for connector elements. Using the Stratego transformations, the generator produces final source code from the templates.

The method template concept is an important part of the element template. This construct is mainly used to adopt the business interfaces for the specific connector element.

The meta-variables of the method template allow access to the arguments of the method. Using the `${method.variables}` meta-variable it is possible to insert the list of the method's arguments into the final code.

It is, however, not possible to adjust or alter this list in the template. The list can only be printed out but not modified. For example, in some use cases it could be necessary to append additional arguments (e.g. call context) to the argument list when a call in method invocation communication style is passed to the next element in the connector element hierarchy.

Also the types of the arguments are not accessible or visible, therefore it is not possible to generate code which depends on the types of the arguments. This can be needed, when it is necessary to convert all arguments of a certain type to a different type which can be transferred via the network and then converted back.

In the previous chapter the `implements interface` construct has been introduced. It allows for specifying that the resulting class of the element should implement one or more interfaces. The template language however does not allow for the specifying, that the resulting class should extend some parent class. This is for example, necessary when producing the skeleton connector for the Java Remote Method Invocation.

The mentioned restrictions bring significant limitations into the generation process and do not permit the implementation and generation of a full stack of

connectors required by the SOFA component system.

The implementation uses Stratego Compiler to compile the code transformations from template to the final connector element code. This creates an element generator native executable, which is dependent on the specific platform and its shared libraries. Such an element generator cannot then be launched on different platforms.

To parse and print input and output files, the generator uses the XTC Repository to invoke external Stratego tools. This makes the generator dependent on the Stratego toolset which is not available on some platforms.

The goal of this thesis is to enhance the existing generator by removing the mentioned limitations. Therefore the thesis should extend the template language with the missing functionalities. Furthermore, the thesis should find a cross-platform solution for generating the connectors' code from the templates.

Particularly, the enhancement should provide template constructs to work with arguments of a method in the method template. It should be possible to add and remove arguments of a method, iterate through them and modify them separately.

It should also be made possible to access the type information of the arguments. For adaptation of arguments it should be possible to check if a given argument in Java is of primitive type or reference type.

The constructs that will offer this functionality should be easy to use by the connector developer. They should closely reflect the needs of the domain of connectors. Where more approaches are possible the template language should remain *domain specific* rather than trying to be a more general template language. [33]

Another set of goals should look for a cross-platform solution for template generation. The platform dependencies caused by usage of the Stratego Compiler and the XTC Repository tools should be removed. To achieve this, appropriate technologies need to be found and adapted for the existing connector generator.

The goal is also to integrate the enhanced template language and template transformations into the existing connector generator and evaluate it with the SOFA component system.

Finally, the thesis should explain the connector generation process as a part of the standard development process. It should describe the roles required by the presence of connectors and automated connector generation.

3.1 Goals Summary

- *(G1):* Provide template constructs for abstract manipulation with arguments and return values in method templates.

- *(G2):* Make the type information of the arguments accessible.
- *(G3):* Remove platform dependencies brought by Stratego Compiler and native Stratego tools.
- *(G4):* Describe connector generation process in the context of classical development process.

Chapter 4

Template Language Enhancements Proposal

Limitations with the existing connector generator have been briefly introduced in the previous chapters. This chapter will discuss enhancements that have been implemented by this master thesis. These enhancements should help overcome the limitations of the existing generator.

The limitations of the existing template language form a knowledge basis for creating a list of requirements for the proposed enhancement. Requirements are motivated by domain knowledge and real requirements coming from the SOFA component system.

In the first section, new template language constructs are designed, based on the enhancement requirements. In the following section the implementation phase is described. The new constructs need to be integrated into the existing element generator. The template language grammar has to be extended with the new constructs. Finally, the connector element generator which generates the final connectors' code from the templates, needs to support the new language constructs and produce appropriate code.

4.1 Requirements and Proposed Constructs

The communication in component systems is negotiated via defined business interfaces. These interfaces are also reflected in the connectors and their code. The external interfaces of a connector have the same signature as the interfaces of components they connect, that's why a connector is also a component from an external point of view.

Inside of the connector, however, the interfaces between each connector element might be different. Different network technologies and communication styles might

not allow or support every aspect that a general interface defines. Not all arguments and return value types might be supported. Along with the business arguments, certain technologies might prescribe to pass additional arguments for example to maintain the network connection.

Various strategies for method argument adaptation might be required. In the connector elements that implement the Java Remote Method Invocation it is necessary to change the arguments slightly. It is necessary to append additional argument to the argument list, which will contain the call context data. Arguments which are of the Java primitive type need to be converted to a different type. Also the return value might require conversion, depending on its type.

In such connector elements, the method template, which is responsible for processing a communication request, like method invocation, should prescribe how the arguments from an input interface should be altered, before they are used in the call on the output interface.

The template language therefore should support certain operations with general method arguments in the method template definition. These should include

- adding and removing of additional arguments,
- accessing the type of the argument,
- modifying arguments,
- modifying only certain arguments based on a custom condition, like the type of the argument.

The required functionality can be described by a number of steps, each performing a simple operation. For example, converting arguments of a non-primitive type could be described as following simple algorithm in pseudo-code:

```
foreach (arg in method.arguments)
  t = arg.type;
  if (is-primitive(t))
    arg.expression = encoder.encode(arg.expression);
  endif
endforeach
```

Multiple approaches are possible when designing the language constructs that should allow this functionality of the template. One factor can be the complexity of the constructs of the template language.

The aim can be to keep the template language simple. This would mean adding only the minimum necessary amount of new constructs so that it is at least possible to implement the required functionality in the template. Also the constructs themselves should be simple, perform well-defined simple operations over the code. For example the `if` construct is one of the simplest template constructs.

The connector developer would then have to use these simple constructs and in the created template, explicitly describe all the steps of the desired functionality using the simple construct. For example issuing a method call with some modified arguments and an added argument could be implemented as shown in *Listing 4.1.*

```
this.target.${method.name}(
$set c = 0$
$foreach(ARG in ${method.args.get})$
  $if (c > 0)$ , $end$
  $set type = ${method.args.ARG.type}$
  $if (type.isPrimitive)$
    ${ARG.name}
  $else$
    encoder.encode(${ARG.name})
  $end$
  $set c = c + 1$
$end$

$if (c > 0)$
,
$end$

this.getCallContext();
);
```

Listing 4.1: An example of template composition of a method call with modified arguments and an appended argument.

Such an approach would result in a simple and easy to learn template language. On the other hand, the resulting template code could get very lengthy. More complex operations would require the developer to write many lines of template code. This also means it is more likely the developer will make an error while writing the template.

Another approach would be to help the developer express desired functionality more briefly by introducing specialized template language constructs. This would require the language to contain more complex constructs than in the previous scenario. The constructs would implicitly perform some of the complex operations that the developer would otherwise have to describe manually.

This thesis follows the latter approach. The connector template language is a domain-specific language. It is desired that the constructs in it reflect the domain of where it is used. [31] The proposed language constructs aim to help express common tasks in this domain, in a concise way. This results in connector element template code which is easy to read and comprehend, has a reduced template length and has less space for semantic errors.

The new constructs, that reflect the domain of working with arguments in method templates, will be introduced in the following sections. First the data structure for arguments will be introduced. Secondly the operators to work with the structure will be presented.

4.1.1 Arguments As Array

The most suitable structure for storing and accessing arguments of a method in the method template is a list or an array. These allow for the, accessing of its elements one by one, accessing the beginning or the end of the arguments list, and appending or removing arguments.

This thesis therefore introduces implicit arrays in each method template. This thesis follows the notation convention introduced in [34]. The array with arguments in a method template is denoted as `method.args`. This construct encapsulates a list of all method arguments and information about their types. The template variable `${method.args.arg1.type.name}` will contain the type name of the method argument called `arg1`. The template variable `${method.args.arg1.type.isPrimitive}` will evaluate to one or zero depending on if the `arg1` method argument is of a Java primitive type.

To work with the list of method arguments, this thesis implements a number of new operators. The whole list `method.args` is then used as a list on which these operators work.

4.1.2 Arguments Operators

This thesis introduces the following general array operators that are suited for working with an array of method arguments in a method template. These operators can be easily extended to support general template array variables if the template language is about to support them in the future.

Count

The `count` statement is introduced to determine the number of elements in an array. The statement comes in two forms. In its basic form it always evaluates to the total number of elements. The extended form determines the number of elements that fulfill some condition. This is the syntax of both forms of the `count` statement:

```
"count" "(" VarRefPart ")" -> Expr {cons("ArrayCount")}
"count" "(" Id "in" VarRefPart "where" ArrayCondition ")" -> Expr {cons("ArrayCount")}
```

The `VarRefPart` stands for a name of an array variable. In the case of method arguments in a method template, it will always be `method.args`. The `Id` is a pattern representing a single element of the array. This pattern can be used inside of the `ArrayCondition` expression. The expression will be evaluated for every array element. During the evaluation of all occurences of the pattern, it will be substituted with the current array element. The count statement will return the number of elements for which the `ArrayCondition` evaluates to non-zero value.

```
// Evaluates to number of method arguments.
$count(method.args)$
// Equivalent to:
$count(ARG in method.args where 1)$

// Evaluates to the number of method arguments of non-primitive types.
$count(ARG in method.args where !method.args.ARG.type.isPrimitive)$
```

Peek, Pop, Append

To be able to add and remove method arguments in a method template, `append` and `pop` statements were introduced. The `peek` statement accesses the last element added to the argument array.

The typical usage scenario of adding extra arguments when processing a method invocation request in a connector element, is to insert an additional argument to the arguments of the method. Later, on the opposite side of the connector element hierarchy, these extra arguments are removed from the argument list and processed, and the rest of the regular arguments are then used.

For this usage scenario, the abstraction of a stack data structure reflects the typical usage needs best. Therefore the `append`, `pop` and `peek` statements operate on the last element of the argument list. The `append` statement adds a new argument after the last argument, `pop` removes the last argument. Finally, `peek` accesses the name of the last argument.

```
"$" "peek" "(" VarRefPart ")" "$" -> ArrayPeek {cons("ArrayPeek")}
"$" "pop" "(" VarRefPart ")" "$" -> Stm {cons("ArrayPop")}
"$" "append" "(" VarRefPart "," TargetLanguageExpr ")" "$" -> Stm {cons("ArrayAppend")}
```

The syntax of the `peek` and `pop` is simple, they take the name of the array as the only argument. The `append` statement accepts any valid expression of the target language in the place of `TargetLanguageExpr` non-terminal. When the list of the arguments is printed out, this expression will be printed at the place of the given argument.

```
// Removes last element from the method arguments list.
$pop(method.args)$

// Evaluates to last element from the method arguments list.
$peek(method.args)$

// Appends a new argument to the end of the method argument list.
$append(method.args, SOFAThreadHelper.getCallContext())$
```

Apply

The `apply` statement allows the modifying of multiple elements of an array in a single step. Often only certain method arguments need to be modified, for example surrounded by a type conversion code, based on some condition, like the type of the argument. The syntax of the `apply` statement is following:

```
"$" "apply" "(" TargetLanguageExpr "for" Id "in" VarRefPart
  "where" ArrayCondition ")" "$" -> Stm {cons("ArrayApply")}
```

The `TargetLanguageExpr` non-terminal stands for any expression from the target Java language, `Id` denotes the substitution pattern to be used in the condition marked by `ArrayCondition` and in `TargetLanguageExpr`. The pattern will be replaced with an element from the array. The `VarRefPart` stands for the name of the array.

Each element of the array will be inspected by the `apply` statement. Those elements that fulfill the condition given by `ArrayCondition` will be replaced by the expression prescribed in `TargetLanguageExpr`.

```
// All arguments that are not of a Java primitive type
// will be adapted using an rmiEncoder object.
$apply(rmiEncoder.adaptObject(ARG) for ARG in method.args
  where !method.args.ARG.type.isPrimitive)$
```

Implode

The statement, to output a comma-separated list of all method arguments in a method template, is called `implode`. It is mainly used when the connector element passes the call to further connector elements.

The statement takes a single argument which is the name of the array to be printed out. Typically it will always be `method.args`. The syntax is as following:

```
"$" "implode" "(" VarRefPart ")" "$" -> ArrayImplode {cons("ArrayImplode")}
```

```
// Invoke a method with the same name as the current one
// with previously customized argument list.
this.target.${method.name}($implode(method.args)$);
```

4.1.3 Return Value Handling

Once the arguments for a method call have been adjusted in a given connector element, the method call on the target connector element can be issued. After that it might be necessary to handle the return value of such a method call. With connectors typically three situations occur, based on the type of the return value.

- *void:* Target method does not return any value.
- *no conversion:* The target method returns a value of the same type as the current method and there is no need to adjust the returned value.
- *conversion needed:* The target method returns a different type than the method in the current element and a conversion is needed.

These situations need to be reflected in the connector element template. The template needs to prescribe how to handle all three situations, if the element is expected to handle business interfaces with methods of any return type.

One of the approaches, would be to specify multiple method templates, each for different return types of the target method as shown in *Listing 4.2*. Method templates for *void* methods could be shorter, only contain code for invoking the method and no code for handling and passing through the return value. Method templates for methods which do return some value but do not require its conversion, would also contain some code to store the returned value and then later a statement to pass the value to other connector elements in the connector hierarchy. Finally, the template for methods which also require conversion of the returned value would also contain code that would convert them.

```
implements interface IFace {
  // template for void methods
  method template void {
    System.out.println("method␣>␣${method.name}␣<␣called");
    this.target.${method.name}($implode(method.args)$);
    System.out.println("method␣>␣${method.name}␣<␣ended");
  }

  // template for methods without conversion of the returned value
  method template primitive {
    System.out.println("method␣>␣${method.name}␣<␣called");
    ${method.declareReturnValue}
    ${method.returnVar} = this.target.${method.name}($implode(method.args)$);
    System.out.println("method␣>␣${method.name}␣<␣ended");
    ${method.returnStm}
  }

  // template for methods with conversion of the returned value
  method template nonprimitive {
    System.out.println("method␣>␣${method.name}␣<␣called");
    ${method.declareReturnValue}
    ${method.returnVar} = this.target.${method.name}($implode(method.args)$);
    System.out.println("method␣>␣${method.name}␣<␣ended");
    return decoder.decode(${method.returnVar});
  }
}
```

Listing 4.2: An example of different method templates for methods with different return types.

This would lead to straightforward template code, on the other hand it would introduce lots of code repetition, which is always a problem. It would make editing the connector element template difficult and more error prone as some changes would require modifications to three different method templates.

This approach could be altered in a way that the common parts of all the three templates could be merged into a single method template. Then the parts that differ, would be handled in code branches, where each branch would contain code to handle one of the three mentioned situations as shown in *Listing 4.3*. This approach would remove some the of code repetition from the first approach. However, some parts would still be repeated and the three branches would make the understanding of the code a bit more difficult.

```
implements interface IFace {
  method template {
    System.out.println("method␣->␣${method.name}␣<␣called");
    if (${method.returnValType.isVoid})
      this.target.${method.name}($implode(method.args)$);
    $else if (${method.returnValType.isPrimitive})$
      ${method.declareReturnValue}
      ${method.returnVar} = this.target.${method.name}($implode(method.args)$);
    $else$
      ${method.declareReturnValue}
      ${method.returnVar} = decoder.decode(
        this.target.${method.name}($implode(method.args)$));
    $end$
    System.out.println("method␣->␣${method.name}␣<␣ended");
    ${method.returnStm}
  }
}
```

Listing 4.3: An example of a method template with branches that handle different return value types.

This approach can be further enhanced to handle some cases in a single statement. This thesis proposes such a solution. It allows the connector element template developer to write just one method template which will describe how to handle different types of return values in one place.

This thesis does not prescribe where the return statement should be placed in the final method, for example always at the end. It is still up to the template developer to choose where the return statement should be generated, using the `${method.returnStm}` meta-variable. Also a temporary storage area for the returned value is declared using the `${method.declareReturnValue}` meta-variable. This will create a declaration of a local variable inside of the target method body.

This thesis then introduces a new statement `setReturnValue` which transparently handles two mentioned cases of the return value type. For methods of void type it will create a statement identical to its argument. For methods that do return a value it will create an assign statement, where the returned value is assigned to

the local temporary variable.

```
"$" "setReturnValue" TargetLanguageExpr "$" -> Stm {cons("SetReturnValue")}
```

```
method template {
  // declares a variable, for example int a_0
  // or generates no code if the method does not return a value
  ${method.declareReturnValue}

  $setReturnValue this.target.${method.name}($implode(method.args)$)$
  // evaluates to either
  // this.target.foo(...);
  // or
  // a_0 = this.target.foo(...);

  // generates either an empty return statement or return a_0;
  ${method.returnStm}
}
```

Finally, this thesis proposes ways to create code based on the method return type. The `method.returnType.isPrimitive` meta-variable can be used to determine whether the target method return type is a Java primitive type.

```
$if (!method.returnType.isPrimitive) $
  try {
    return encoder.adaptObject(${method.returnVar});
  } catch (RMIObjectAdaptorException e) {
    throw new ConnectorTransportException (e);
  }
$else$
  ${method.returnStm}
$end$
```

4.1.4 Extended Class

The evaluation of the connector element template will output final code of the element. This code will be a class, implementing the functionality of the element. It might be necessary to specify that the resulting class should inherit from some extended class. For example Java Remote Method Invocation prescribes that the class implementing an RMI Skeleton should extend the `UnicastRemoteObject` class.

This thesis therefore introduces a simple `inherits` statement to specify which class the resulting element's class inherits from.

```
"inherits" SuperClassName -> InheritsDecHead {cons("InheritsDecHead")}
InheritsDecHead "{" InheritsMemberDec* "}" -> InheritsDec {cons("InheritsDec")}
```

For example, in the case of RMI skeleton element, the code that prescribes that the resulting class should extend the UnicastRemoteObject class is like this:

```
inherits java.rmi.server.UnicastRemoteObject
```

```
{
  // no methods to override
}
```

Multiple inheritance is not available in the Java language.[1] Therefore there can be only one instance of the `inherits` construct in a single connector element template, otherwise the element generator will issue a transformation error.

4.2 Implementation

This section will discuss the implementation details of the new template language constructs that have been introduced in the previous text. At first, the grammar of the existing template language is extended with the new constructs. Then the functionality of the constructs is implemented into the existing element generator.

4.2.1 Grammar

The syntax of the array operators has been presented in *subsection 4.1.2 Arguments Operators*. The `count`, `peek` and `implode` are defined as expressions. On the syntax level, they can exist anywhere an expression is expected. After evaluation they return a value.

```
%% These array operators are expressions.
ArrayCount -> Expr
ArrayPeek -> Expr
ArrayImplode -> Expr
```

The array operators `pop`, `append`, `apply` and `setReturnValue` are on the other hand defined as statements. On the syntax level, there are rewrite rules that allow these operators to exist where a statement (the `Stm` non-terminal) is expected.

```
"$" "pop" "(" VarRefPart ")" "$"
  -> Stm {cons("ArrayPop")}
"$" "append" "(" VarRefPart "," TargetLanguageExpr ")" "$"
  -> Stm {cons("ArrayAppend")}
"$" "apply" "(" TargetLanguageExpr "for" Id "in" VarRefPart
  "where" ArrayCondition ")" "$"
  -> Stm {cons("ArrayApply")}
"$" "setReturnValue" TargetLanguageExpr "$"
  -> Stm {cons("SetReturnValue")}
```

The array operators `append`, `apply` and `setReturnValue` expect as one of their arguments an expression from the target (Java) language. This is denoted by the `TargetLanguageExpression` non-terminal in the grammar.

[1] In some systems (e.g. Fractal [27] or Scala) it is possible to achieve multiple inheritance in the Java language by specifying special *mixin* classes. Abstract inheritance requirements specified in the mixin class are then satisfied by those super-classes which offer the required elements.

The base ELLANG language defines the array operators as general ones. However, in the ELLANG-J language, the operators are supposed to work on the array of arguments in a method template. Therefore the array operators also need to work with expressions from the target Java language.

This is reflected in the grammar as shown in *Listing 4.4*. In the ELLANG language any ElLang expression can be rewritten to the `TargetLanguageExpression` non-terminal. In the ELLANG-J language also any expression from the Java language can be rewritten to that terminal too. Finally, this rule gets assigned higher priority than the rule in ELLANG.

```
context-free syntax
  %% From the ElLang grammar:
  Expr -> TargetLanguageExpr

  %% From the ElLang-J grammar, JCX is Java language context:
  Expr[[JCX]] -> TargetLanguageExpr[[ECX]] {avoid, cons("ToTL")}

priorities
  <Expr[[JCX]]-CF> -> <TargetLanguageExpr[[ECX]]-CF>
  > <Expr[[ECX]]-CF> -> <TargetLanguageExpr[[ECX]]-CF>
```

Listing 4.4: Integration of the TargetLanguageExpr non-terminal into the grammars.

4.2.2 Arrays Representation

From the template developer point of view, the data structure that is the most suitable for maintaining the arguments of a method, is a list combined with an array, as described by the use-case scenarios in *subsection 4.1.1 Arguments As Array*. The goal of this thesis is then to implement the representation of the arguments list in Stratego.

The arguments list, is a type of template variable. The existing element generator already supports template variables. On the grammar level, it is not necessary to add any further support for arguments list. The `method.args` meta-variable which denotes the list of the arguments will be recognized as `VerRefPart` non-terminal, and `${method.args}` will be recognized as `VarRef` non-terminal. It will also produce a term in the AST. These pre-existing grammar rules define this syntax:

```
Id          -> IdElement     {cons("IdElement")}
{ IdElement "." }+  -> VarRefPart
"${" VarRefPart "}" -> VarRef      {cons("VarRef")}
```

4.2.2.1 Possible Reuse of Existing Functionality

When implementing the internal representation of an argument list or template lists in general, it could be possible to reuse some of the functionality of the existing

element generator. In *subsection 2.3.2 Element Generation*, the query module has been briefly introduced. It allows the accessing of structured data and is used when querying data from the element descriptor. There is also an existing initial support for arrays in the `set` command.

The query module can access structured data and is already integrated into the evaluation of the template variables. The following piece of template code will make use of the query module during the evaluation of the template:

```
protected ${ports.port(name=call).signature} target;
```

The query module could be reused to access arguments of a method in the method templates. Argument names and their type could be inserted as input data into the query module. During the evaluation of the template, the query module would return the required information about the method arguments.

The query module, however, does not allow the modification of the data it works with. It would not be possible to change, add or remove arguments of the method. Also the evaluation of the queries is done before the evaluation of the statements:

```
evaluation =
  ...
  // First process all variable references
  ; topdown(try(query));
  ...
  // Process statements
  ; alltd(eval-stats);
  ...
```

The statements that modify the arguments list would therefore have no effect. The query module would replace all references to method arguments before the evaluation process executes statements that modify these arguments.

To reuse a query module to work with method arguments in a method template, it would be necessary to change the evaluation process of the template. Also the query module would have to support modification of the data it returns.

The second reusage scenario could be to reuse existing support for arrays. The existing template language already has some support for arrays. The following code will initialize an array `el` with values from 0 up to number of items (minus one) returned by the `${elements.element}` query. The array will be indexed with strings.

```
$set i = 0$
$foreach(ELEMENT in ${elements.element})$
  /* remember ELEMENT index in array */
  $set el[ELEMENT.name] = i$
  $set i = i + 1$
$end$
```

During the evaluation of this piece of template code, the `set` statement will create variables named for example `el[in]` or `el[out]` where `in, out` are some of the values the `ELEMENT.name` expression evaluates to in the cycle. The `[` and `]` brackets are part of the variable name and do not have any special meaning. It is then possible to access the elements of this array by using the appropriate index. The index can be a string, a number or other terms, in the case of this example code, it is what the query module returns for a given query.

These variables, however, are not tied together. It is not possible to iterate through the `el` array or determine how many elements are in it. It is not possible to define an order for the elements of this array. This major functionality is lacking and the limitations prevent the existing array support to be used for handling method arguments in method templates. Using an invalid index to access the array will lead to invalid AST produced by the transformations, as the variable will not get replaced by any value.

4.2.2.2 New Implementation of Arrays

This thesis brings a new implementation of the representation of arrays and lists. It introduces a concept of arrays combined with lists inspired by arrays in PHP [11]. In PHP the array data structure can be also treated as a list (vector), hash table, dictionary, collection, stack, queue and so on.

From the template developer's point of view the operations that can be performed with the arrays are:

- printing and reading,
- usage in array operators introduced earlier in this thesis (count, peek, pop, append, ...).

Internally, the arrays are stored in a list structure that associates values to keys and defines an order. Therefore the array elements are also ordered. This array infrastructure is called *internal arrays*.

This thesis introduces a set of strategies to work with the internal array structures. In the `eval/eval-array` module the strategies to create, obtain and search internal arrays are present as well as strategies for inserting and removing elements from the end of the array, checking the length of the array and others. Arrays are then kept as a global variable via dynamic rules. The name of the dynamic rule that stores content of all internal arrays is `Arrays`.

Arrays are identified by the name of the array, which can be any Stratego term. The basic strategies to create an array with given name and content, access array by its name and delete an array are shown in *Listing 4.5*. All strategies expect the current term to be set to the identification of the array.

This general implementation of internal arrays builds a solid basis for possible future extension of the template language, if it is to support arrays as proper template variables.

```
// Assigns contents of given array.
array-set(|data) =
  where( id => identifier )
  ; rules( Arrays: identifier -> data )

// Obtains contents of the given array.
array-get =
  Arrays <+ ( log(|Warning(), "Array not found", <id>); fail )

// Deletes given array.
array-unset =
  where( id => identifier )
  ; rules( Arrays:- identifier )
```

Listing 4.5: An example of three basic strategies that create and delete internal arrays.

4.2.3 Array Operators Implementation

The implementation of array operators and arrays of method arguments in a method template is based on internal arrays. During the evaluation of the template the array of arguments is represented using the internal arrays.

The arrays get created during the evaluation of the method template, after the interface for which the template is prescribed gets parsed. For every method present in the interface the following steps are performed:

- The method declaration gets expanded. The method template body gets substituted as the body for the given method declared in the interface.
- The method declaration, is also used to initialize meta-variables and arrays containing info about the method. This is where the initialization of the array with method arguments was added. Also in this phase, other meta-variables with information about the method like `${method.name}` get initialized.
- Finally, the method gets evaluated. The replacements that can be done instantly are performed in this phase. Meta-variables like `${method.name}`, `${method.returnType}` and others get replaced with their content.

The existing element generator was replacing all meta-variables with information about the method in the last phase. This was possible because the final value of the variables was known in that phase and would not change later during further evaluation of the template. The introduction of a modifiable method arguments list,

however, changes this. Therefore the references to the arguments array do not get replaced with the contents of the array. This is left to be done in the statements evaluation phase. The references to `method.args` meta-variable are not expanded in the method template evaluation phase.

The evaluation of statements comes after the evaluation of method templates. It is performed on the whole connector element template. The evaluation of statements does not distinguish the method body blocks. Therefore it is unable to distinguish between references to `method.args` meta-variable in two different methods.

To distinguish the references to an array of arguments in each method, it would be necessary to change the process of evaluation of statements. It would have to evaluate statements in each method separately, right after the meta-variables for a given method are initialized. This would blend together the process of method template expansion and evaluation of the statements, which in the current implementation is strictly separated.

Another option to achieve distinction between method meta-variables is name mangling. This approach is implemented in this thesis. During the evaluation of the method template an additional step was added. References to method arguments and their types are mangled with a new prefix that is unique in every method. For example reference to `method.args` in the first method of an interface will be replaced with `a_0.method.args`, and appropriate internal array will be created storing the contents of the arguments array.

Finally, during the evaluation of the statements, the operators that work with arrays will have the mangled unique name of an array as their argument. They will use the internal array strategies to obtain contents of the array and will further process it.

4.3 Cross-Platform Generator Solution

The platform dependencies of the original element generator make its use in a cross-platform component system problematic. The goal (G3) of this thesis is to find a cross-platform solution for connector generation.

Stratego is still one of the richest toolsets for working with syntax and transformations of programs. It is also still under active development. The existing element generator is solely based on Stratego. Therefore this thesis does not aim to replace the whole Stratego infrastructure as it would require rewriting the element generator from scratch with completely different technology.

Instead, this thesis aims to replace the parts of the Stratego solution that introduce the platform dependencies. The Stratego Compiler and the usage of the XTC repository are the main sources of platform dependencies, as described in

subsection 2.2.6 XTC Repository. This thesis therefore looks for a solution that would replace just these two artifacts.

4.3.1 Stratego to Java

The original Stratego Compiler produces C code which is then compiled into native binaries. These depend on system shared libraries and also the parse tables of the ELLANG-J language. The compile process with Stratego Compiler is shown on *Figure 4.1*.

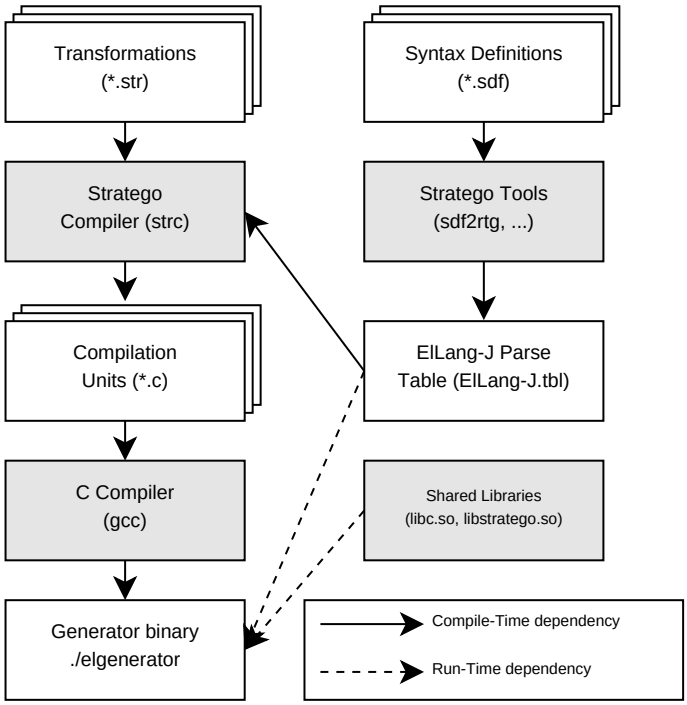


Figure 4.1: Previous element generator compilation process.

The STRJ compiler [16] is a Java-based variation of the Stratego Compiler. It was nearing maturity[2] in 2009 and the end of its alpha state has been announced in 2010.

The STRJ compiler compiles the transformation programs written in Stratego into Java source code. Following compilation with a standard Java compiler then outputs standalone cross-platform Java application[3].

[2]During the integration of STRJ compiler into the build process of the element generator there was a confirmed bug found in the Java implementation of Stratego parser (JSGLR): http://bugs.strategoxt.org/browse/JSGLR-6?page=all. Also a bug in the Windows Cygwin implementation of one of the Stratego tools has been reported and confirmed: http://bugs.strategoxt.org/browse/STR-779

[3]The transformation program compiled with STRJ will be dependent on Java libraries which are included in STRJ Java archive and are cross-platform too.

In the latest version of Stratego a new construct was introduced that allows inclusion of parse tables in the final binary. It is similar to static linking of libraries. The construct `import-term` allows inclusion of any term (ATerm) in the transformation source codes. Parse tables are internally represented as ATerms in binary format. It is therefore possible to include them into the source code of the transformation using the `import-term` construct. During the compilation of the transformation, a given parse table gets substituted in the place of the import. These are then used to parse input source code units.

Along with switching to the STRJ compiler this thesis also removed the runtime dependency on ElLang-J and other parse tables[4] used by the element generator. The new compilation process is shown in *Figure 4.2*.

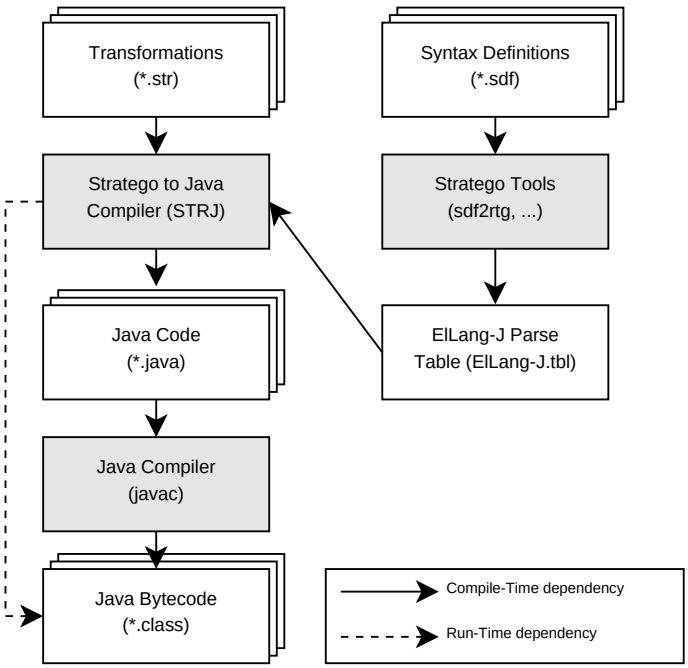


Figure 4.2: Compilation process with STRJ compiler.

The steps to generate the parse tables from the syntax definitions are still handled by the native Stratego tools. The STRJ compiler does not offer the functionality to compile syntax definitions into parse tables or other output formats.

4.3.2 XTC Repository and Stratego Libraries

The XTC repository is a storage of Stratego native tools that implement functionality required for the transformations, like parsing, pretty-printing and others. The repository can also store user defined transformation tools. The dependency on this

[4]The element generator also parses the element descriptor which is in XML format and Java interfaces. Therefore it needs parse tables for XML and Java.

repository and the native tools in it, bring platform dependencies into the connector generator.

The support of XTC repository itself is not implemented in the STRJ compiler and is not going to be supported there in the future [16]. Even if it is still possible to invoke standalone external tools (not via the XTC repository), this approach still brings platform dependencies.

The original element generator uses the XTC repository to invoke tools for parsing and pretty-printing of files. It is parsing the input element descriptor in the XML format, interfaces in Java and connector templates in the ElLang-J language. As the result it pretty-prints the Java code of the connector element. In case of error, the generator pretty prints the current abstract syntax tree when the error occured. The tools that handle these tasks are `sglri`, `pp-java` and `pp-aterm` tools.

To remove dependencies on these native tools the invocations of these tools were replaced with an invocation of corresponding strategies from Stratego standard libraries. The `libstratego-sglr` library contains strategies for parsing. The strategies take as input, the parse table, start symbol and the input string or file to be parsed, just like the `sglri` tool.

In the implementation phase, this thesis also replaced calls to native pretty-print tools with invocation of strategies that perform the same task. The library `aterm-front` contains strategies to pretty-print ATerms. The `java-front` library contains strategies to pretty-print Java code.

The mentioned libraries are standard libraries in the Stratego toolset. The standard Stratego (C) Compiler creates executables dependent on these shared libraries usually installed on the system together with Stratego. The Stratego to Java compiler contains these libraries in the runtime package which is part of the class-path when launching the compiled application.

4.4 Integration into SOFA

The original element generator is integrated into the SOFA component system as described in *subsection 2.3.6*. The `StrategoGenerator` class implements the invocation of the Stratego element generator native binary. This is done by using the Java Native Interface [9].

To integrate the enhanced generator into the SOFA component system, this thesis implements the `StrategoJGenerator` class. A command with an identical name is inserted into the configuration of the `ElementGenerator` and the generation of all main connector elements[5] is configured to be done using the command

[5]The architecture for the connector elements for the messaging communication style is still under development. This thesis does leave the generation of these elements to be handled by the

`StrategoJGenerator`. This is a non-invasive change that allows both element generators to be utilized at the same time by changing the XML configuration of the connector generator.

The Java Native Interface does not need to be used with the new element generator. The `StrategoJGenerator` launches the element generator directly as Java code. The `strj` compiler makes the *main strategy* of the element generator accessible as a static method in Java. The same parameters are passed to the element generator as in the case of the original element generator. The first one is the element descriptor, the second one is an output log file.

The enhancements brought about by this thesis allow for the generation of new types of connector elements. The architecture of the connector generator had to be slightly improved to support generation of these elements. Part of the connector elements have templates with the `import` statement. Before launching the generation of such connector elements, it is now necessary not only to prepare[6] the template of the given connector element, but also all included templates. However, before the execution of the element generator itself, it is not known which templates will be required by the generation process. The template would have to be parsed already before the execution of the element generator and this would make the parsing process duplicated. Therefore all the connector element templates are prepared along with the main template of the current connector element.

To allow instant testing of connector templates, it is possible for a connector developer to manually prepare connector templates before executing the connector generator. Then the connector generator will not overwrite these templates with the original ones contained in the distribution package of the connector generator. This way it is possible to test new templates instantly without modifying the connector generator.

Support for the generation of composite elements also had to be fixed slightly. Previously the architecture of the connector generator expected one element generator to be used only for one connector element. This was acceptable in the initial implementation of the connector generator, where generation of different connector elements was performed in different specialized Java classes. This thesis makes it possible to generate multiple connector elements by using just the `StrategoJGenerator` element generator. Therefore the support to re-launch the same element generator, for generation of multiple connector elements was added.

The implementation part of this thesis is replaceable with the original imple-

original generator written in Java [29].

[6]The connector generator is distributed as a Java archive (jar) and the Stratego implementation makes accessing files, like the connector element templates from jar archives difficult. Before invoking the element generator the templates are therefore extracted from the archive into the target file-system where the element generator is able to load them.

mentation of the connector generator. The implementation is separated into three files:

- `congen.jar`: The core of the connector generator, the part that contains the architecture resolver and configuration of the element generation. The element generation is configured to use the Stratego based element generator compiled with the STRJ compiler.
- `strjelgen.jar`: The element generator implemented in Stratego and compiled with the STRJ compiler.
- `strategoxt.jar`: The Stratego run-time support and libraries. The STRJ compiler is distributed in this archive too.

By replacing the `congen.jar` archive in the SOFA installation and supplying it with the other two archives it is possible to switch the SOFA component system to use the Stratego based element generator compiled with the STRJ compiler. The implementation is available in the source code repository of the SOFAproject[7]

[7] svn://svn.forge.objectweb.org/svnroot/sofa/trunk/congen/branches/strj/congen-core

Chapter 5

Role of Connector Generation in Development Process

5.1 Component-based Development Process

The introduction of Component-based software engineering brought changes into the classical development and life cycle processes [30]. The classical development models (e.g. V-model or the Waterfall model) prescribe series of phases, where the most common are analysis and specification, system design, implementation, integration, verification and validation and maintenance of the system.

Component-based software engineering adds additional component-oriented phases into the classical process and replaces the implementation phase in the process. Instead of implementing the system from scratch, existing components are selected to bring desired functions into the system. These components might need to be adapted to fit appropriately into the newly created system. The phase of selecting and adapting components becomes a part of the development cycle. Development of the system and development of the components compose two parallel flows in the development process, as shown in *Figure 5.1.*

Also some functionality might not be covered by any components. Then new components need to be implemented too. Their development process is then similar to the development process for the whole system. The components might also contain some sub-components which need to be adapted and implemented, etc.

The selection of components is done by the *system designer* role. This person is responsible for selecting appropriate components for the created system or delegate development of new components to a *component developer*. The component developer implements required functionality in standalone components and publishes them in the *component repository*.

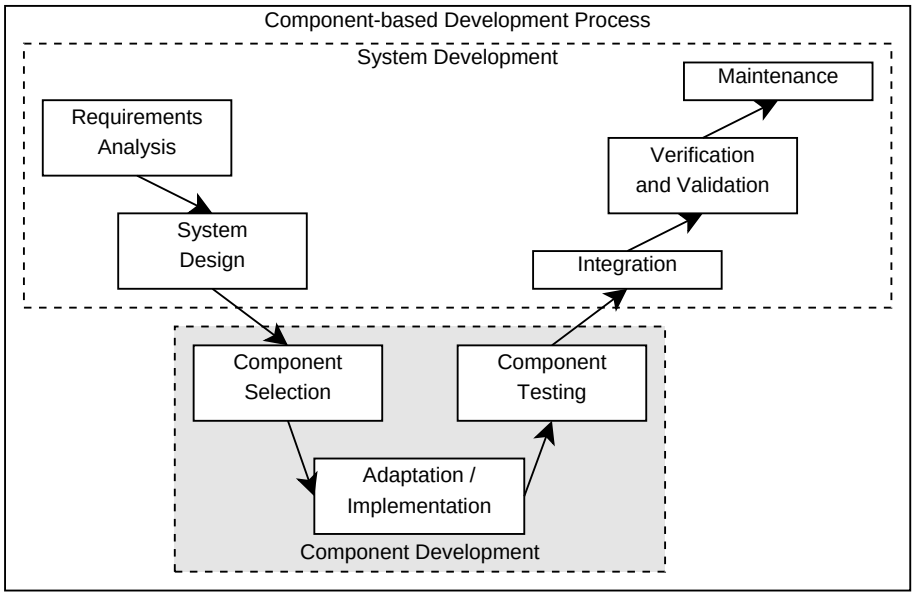


Figure 5.1: Development cycle in component-based development.

5.2 Connector Generation in Development Process

In those component systems which employ connectors and automated connector generation, the development process contains additional phases. The connector generation is the main phase to consider. The generation can be started as soon as distribution details about components are known, such as which components and interfaces need to be connected, and where the components are distributed. Therefore, the connector generation is done during the deployment stage [28], which is a part of the system integration development phase. It can also be done during system run-time, in the event of system reconfiguration requests.

To be able to generate connectors in an automated fashion, the connector model used for the development of the system should support the automated connector generation, that is, the connector generator should be part of the selected component framework. The component model should also provide some *initial set of connectors*, so that it is possible to establish basic types of connections between the components in the system.

The connectors that enter the connector generation process are in fact *abstract*. They describe the connection in a general way. The final code of the connector will be adjusted for the target components in the system [29]. It is the connector generation process that generates final connectors from the abstract ones. Abstract connectors are for example (abstract) connector templates. On the other side, the concrete connectors in the target system are represented by executable units of code.

However, the developed system might require connectors that are not part

of the initial set of connectors provided by the component system. For example a special type of networking technology due to performance, bandwidth or other requirements might be needed to connect components. In such case, it is necessary to develop additional connectors and include them in the connector generation process.

The situation is similar to the process of selecting, adapting and implementing new components. When looking for a technology to connect components, some connectors can be re-used. In some cases, though, it will be necessary to develop new connectors and make them part of the *connector repository*. The connector development process is another parallel process in the overall component-based development process, as shown in *Figure 5.2*. This separated connector development process brings additional roles into the overall development process.

5.3 Roles in Connector Development

5.3.1 Connector Designer

Firstly, it is necessary to decide what kind of connectors will be used in the system, whether some existing (abstract) connectors can be re-used or new ones need to be developed. This is the task of the *connector designer* role. This role is the most similar to the system designer role. The system designer role includes the task of selecting appropriate components for the system. The main task of the connector designer role is to select appropriate connectors for the system.

The task of selecting connectors from the *connector repository* can be automated. It is possible to build a database of properties for the connectors and let the automated generation process also select the most suitable set of connectors. The task of the connector designer is then simplified. The connector designer now only has to specify the properties that the selected and generated connectors should reflect. This specification is called high-level connector specification [29] and forms an input to the connector generation process. The generation process will select appropriate connectors automatically.

5.3.2 Connector Developer

In the case that no suitable (abstract) connectors are found in the connector repository, new connectors need to be developed. The role of the connector developer is then similar to the role of the component developer. The main task is to develop an abstract connector (e.g. in the form of connector template) and make it available in the connector repository.

The connector developer typically implements a chosen middleware infrastructure in a general way. The implementation must allow the abstract connector to

be adapted for the business interfaces of the target system. The concept of code templates is suitable for this task.

The connector developer is then restricted by the capabilities of the template language and the target connector generator. It might be possible to describe the process of adaptation in a general way in the template, but some abstract operations might not be supported by the template language or by the connector generator. In such case there is a need to extend the connector generator with the missing functionality. It is then possible to identify another level in the hierarchy of connector development roles, called the *connector generator developer*.

5.3.3 Connector Generator Developer

The role of the connector generator developer brings the tasks of implementing the support for abstract connector definitions, typically connector templates, and also implementing the connector generator functionality. The connector generator developer carries the task of defining the language of the templates. The goal is to support the abstract manipulations that might be required when adapting interfaces for different networking technologies and communication styles, the core of the connector generation process.

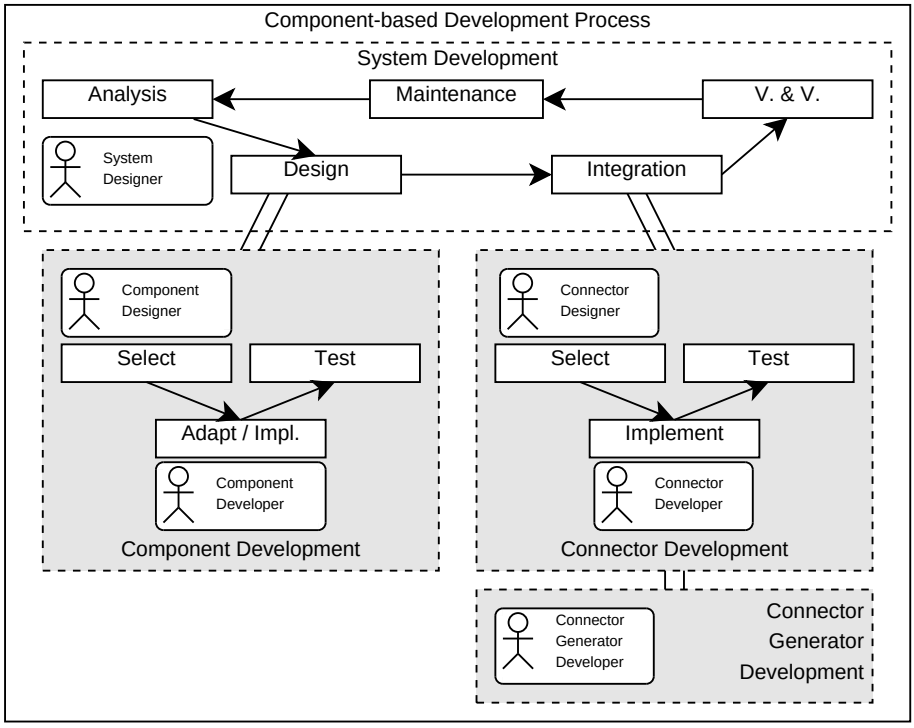


Figure 5.2: Component-based development cycle with connectors.

5.4 Required Knowledge

A connector model that contains support for automated connector generation and abstract connector templates imposes different domain knowledge requirements on the development.

- *Component model knowledge:* The component model prescribes the model of components, connectors, their cooperation, and so on. The understanding of these base concepts is necessary for the work of most of the roles in the development process, that is at least for system designer, component developer, connector designer, connector developer and connector generator developer.

- *Connector model knowledge:* The connector model prescribes the role of the connectors in the component model. It describes the connector architecture, and life-cycle [23]. For the role of component developer, the basic understanding of connectors is sufficient. Understanding the connector architecture or life-cycle is not required to develop components. Roles that however require understanding of the connector model are the connector designer, connector developer and connector generator developer.

- *Middleware knowledge:* The connector designer should be aware of the capabilities certain middleware offers. The developed system might contain requirements on communication style, bandwidth or performance and different middleware technologies satisfy these requirements differently.

  The connector developer role contains the task of implementing a given middleware technology in the form of abstract connectors. This role therefore requires understanding of the middleware domain on a higher, implementation level.

- *Connector template language knowledge:* This is fundamental knowledge required by the connector developer role. The template language defines how to denote the abstract code manipulations in the connector template.

- *Syntax formalism, program transformations formalism:* This is the domain of connector generator developer. To be able to define a template language, a syntax formalism needs to be chosen. Also the transformations from a connector template to the final connector code need to be implemented in some formalism too.

The *Table 5.1* summarizes the roles and their required knowledge in the development process.

Domain Knowledge / Role	Component Model	Connector Model	Middleware	Template Language	Syntax, Transformations
System Designer	•	○	○	○	○
Component Developer	•	○	○	○	○
Connector Designer	•	•	•	○	○
Connector Developer	•	•	•	•	○
Connector Generator Dev.	•	•	•	•	•

Table 5.1: Roles and their required knowledge in a development process with connector generation.

5.5 Connector Development

5.5.1 Template Editing

Creating and editing connector templates is a part of the development process. It is the main activity of the connector developer. An integrated development environment (IDE) can make this task easier than editing the templates as plain text. It can also help discover possible errors earlier by on-the-fly syntax checking.

For the Stratego toolset, there is a tool called Spoofax Language Workbench [15] available. The workbench is based on the Eclipse development environment and the Eclipse IDE Meta-tooling platform [8]. It supports the Syntax Definition Formalism (SDF) used in Stratego and also can execute the Stratego transformations. It offers features such as:

- syntax highlighting,
- syntax checking,
- custom error definitions,
- content completion,
- automatic indentation,
- reference resolving, and others.

By importing the grammar of the connector template language into the editor, the editor generates parse tables and a keywords list to enable syntax highlighting

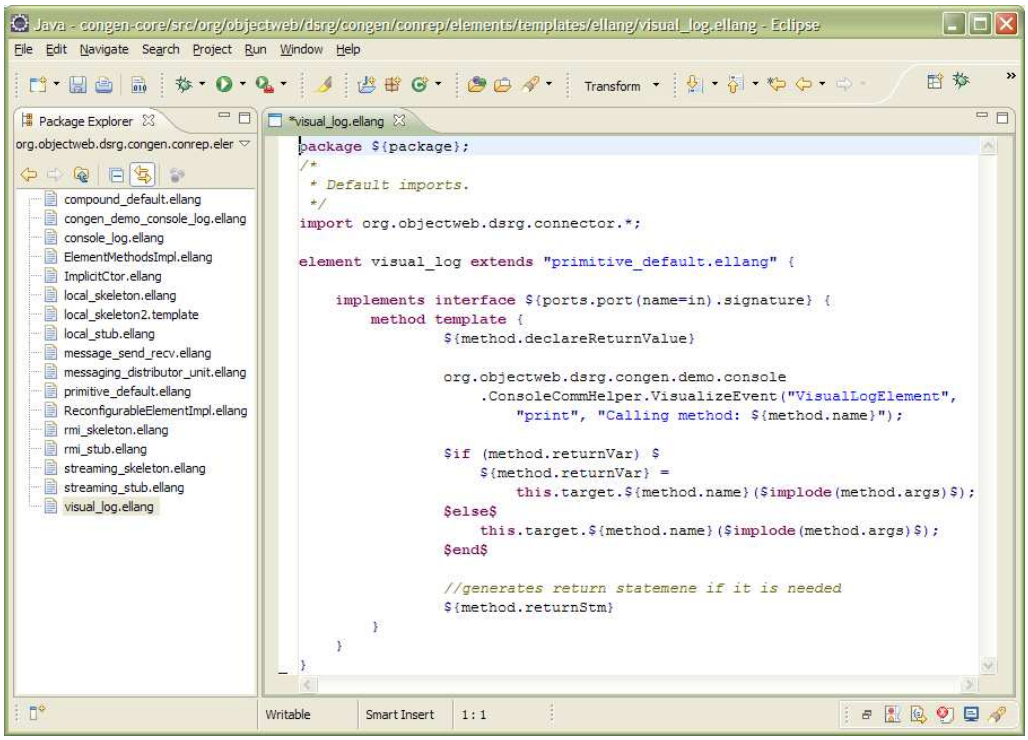


Figure 5.3: Spoofax editor with a connector element template opened.

and syntax checking automatically. It allows for the specification of strategies for custom error and warning notifications and also for content completion.

This thesis provides prepared editor of the ELLANG-J connector template language based on the Spoofax workbench.

5.5.2 Template Testing

There are several types of errors a connector developer can make while writing a connector element template.

- *Syntax Error:* Error on a template syntax level. The connector element template cannot be parsed as a valid input in ELLANG-J language. This includes errors both in the Java code and the ELLANG code of the template.
- *Transformation Error:* Syntactically correct template makes the element generator produce syntactically incorrect abstract syntax tree. It is not possible to produce correct Java code from the result of the transformations. Such error can occur for example when a method variable is put in a place where its evaluation produces invalid Java code. To fix such error, the developer has to remove the template code which produces the invalid parts of the code, for example ELLANG statements that evaluate to invalid code.

- *Semantic Error:* Template is syntactically correct and the resulting connector code is syntactically correct too. However, an error occurs when the connector code is compiled with the Java compiler (for example usage of undefined identifiers), or the connector does not work properly (for example the network communication is not implemented properly).

To help the developer discover these errors before the connector is used in the whole component framework, it is possible to test the connector template in several ways.

The template editor introduced in the previous chapter allows for the checking of syntax errors in the template. The *syntax errors* are highlighted by the editor with a help text describing the type of error briefly (e.g. invalid or unexpected character, unexpected construct).

It is possible to launch the element generator as a standalone application and test whether valid Java code gets produced. For such a test scenario, a testing element descriptor needs to be created, which will reference the tested template. The test element descriptor then may be passed to the element generator. The package with element generator source code contains such testing element descriptors and associated templates. In case there is a *transformation error* in the template, the element generator will exit with an error message saying that an invalid Java abstract syntax tree was produced during the transformation.

To test the connector template in action, a separated connector test suite is part of the connector generator source code tree. The suite contains test scenarios that test connectors from each of the communication styles (method invocation, messaging, streaming).

Finally, the template can be tested as a part of the component framework. When executing the element generator, connector templates get extracted into a temporary system directory. However, templates with the same name do not get overwritten at this stage. Therefore it is possible to put the tested template into the temporary directory in advance. It will then be used instead of the connector template that is included in the component framework and its connector generator build.

5.6 Connector Generator Development

5.6.1 Best Practices and Recommendations

This chapter summarizes best practices and concepts, important for maintaining and developing the Stratego-based element generator enhanced in this thesis. These include:

- usage of proper editor,
- separation of ElLang and ElLang-J syntax,
- strategies implementing the main flow of the template transformation,
- understanding of the usage of standard traversal strategies during the evaluation,
- different kinds of variable references and
- dynamic rules as a global storage for parsing context.

The Spoofax editor introduced in previous sections is suitable both for editing the templates of connectors and also for editing the grammar of the template language. It builds the grammar of the edited syntax and is able to immediately use it for highlighting and syntax checking of the edited files (connector templates). Part of this thesis is a prepared Spoofax editor project with imported ElLang-J syntax.

The syntax of the ElLang-J language is composed of two parts, as explained earlier in this thesis. The core subset of the language is the ElLang language definition, which is independent on the target language. This definition is kept in the `ElLang/syn` directory. The ElLang-J language is an intersection of the ElLang language and the Java language. The extensions done to ElLang-J should respect this separation. When an extension defines a general template construct, it should be defined as a part of ElLang. Only constructs specific to the target (Java) language should be defined in the grammar modules that define the intersection. Those can be found in `ElLang-J/syn`.

The grammar definitions should be kept separated from the transformation part of the generator. The transformations that generate connector's code are defined in the directory `ElLang-J/src`. The parser of the element descriptor is defined in `ElLang/src`. It forms a separated module that parses and prepares the data in the element descriptor.

The main execution strategy is defined in `elgenerator-strj.str` module. The main flow of the transformation is defined in the `process-template-file` strategy. First, the `evaluation` strategy is issued and then the result is printed as a Java code.

The evaluation strategy is composed of transformation phases described in *subsection 2.3.3*. There are different traversal strategies used in these phases. For example evaluation of statements is performed using the `alltd` Stratego standard strategy. This traversal strategy tries to apply its argument on the current term, or if unsuccessful, applies itself on all of its sub-terms. As soon as this traversal strategy succeeds on some term, it will not continue to traverse the syntax tree starting in this term. So when a statement gets successfully evaluated, the statements that are

part of it will not get evaluated. Therefore block statements like `foreach` have to issue additional evaluation of statements in their block explicitly.

On the other hand the `import` statement is evaluated using a combined traversal strategy:

```
bottomup(try(...))
```

This has the effect that the import statement will be evaluated also in templates that have been imported into the main template, and so on.

The strategy combinator `s1 <+ s2` is very important in Stratego. When evaluated, it will first try to apply the `s1` strategy, and if not successful, it will also try to apply the `s2` strategy to the current term. This is used in evaluation of statements and in preparation of the template. Constructs are being matched against a list of statements, until finally a specific statement evaluation strategy matches the current statements and evaluates it.

The same applies for the template preparation process. Constructs that are modified in the preparation phase are matched against the current term until one of them matches and performs the preparation.

As has been mentioned earlier, the preparation phase unifies the structure of some statements, like for example if-then-else block. Another important goal of the preparation phase is replacing `VarRefPart` terms with `VarRef` terms. The `VarRefPart` represents a syntax construct that references a variable but does not directly evaluate it, that is, the variable name is not surrounded with `${...}`. The constructs that work with variables expect the variables surrounded by the `VarRef` terminal. Therefore, in the preparation phase, most of the variable references are rewritten to the latter form.

Dynamic rules are an important concept for modifications that depend on the current parsing context. From a simpler point of view they can be viewed as a storage for global variables. One type of (global) variables in the template language are queries into the element descriptor data. This is handled by the `query` module introduced briefly in Chapter 2. The dynamic rule that stores the data for querying is named `GetXML`. Other types of variables are those created by the `set` statement. Those are managed by the `EvalVarRef` dynamic rule. Finally there are method template meta-variables. These are created in the `java/eval-iface` module and are managed by the `EvalMethod` dynamic rule and also by the `Arrays` dynamic rule, in case of method arguments.

Chapter 6

Evaluation

6.1 Stratego Suitability for Connector Generator Incremental Development

This thesis enhances the connector generator implemented in [34]. It is based on the Stratego program transformations toolset. The toolset employs Syntax Definition Formalist (SDF) for defining grammars of context-free languages and also an infrastructure for writing programs that transform code on the syntax tree level.

The modular approach allows for the incorporating of the enhancements, seamlessly into the existing grammar. The same applies for the extensions in the code generation part, which is based on Stratego transformations. The concept of dynamic rules, allows for context-sensitive changes to be performed on the syntax tree. It allows for the storing of information contained in one part of the input code, and the modification and use of it elsewhere.

Stratego is an established toolset and is still being developed. Since the initial implementation of the connector generator based on Stratego, a new version was released that introduced, among other changes, a rich set of Stratego (standard) libraries. The libraries contain functionality (parsing, pretty-printing) that was previously present in native Stratego tools. Replacing the invocation of these native tools with library calls helps in the removal of some of the existing platform dependencies.

The enhanced connector generator has been integrated back into the SOFA component system. Testing showed that the new functionality of the generator works with real component-based applications written for SOFA. The introduced changes were mainly motivated by requirements of the Java Remote Method Invocation technology. A part of this thesis was the implementation of the connectors for RMI. These were then also successfully tested in SOFA.

6.2 Stratego to Java Compiler

The `strj` compiler compiles the Stratego transformation programs into Java code. Subsequent compilation with the Java compiler produces executable Java bytecode and allows for the deployment of cross-platform connector generator solutions.

The compiler itself is available in Java, which allows for compilation to be launched on multiple platforms too. The development of the Stratego part is now not bound[1] to specific platforms[2].

The compile time duration of the element generator has extended with the usage of the `strj` compiler. The measured differences are shown in *Table 6.1*. The compilation process also produces hundreds of Java source code units. The subsequent compilation with the Java compiler has high memory requirements due to a high number of generated classes. The compiler also does not support incremental builds. It recompiles the whole element generator, even if only one of the source code modules has been modified. This makes the development and testing of the generator lengthy.

The run-time dependencies of the element generator are all contained in the `strj` compiler distribution archive. This archive contains both the compiler and also the run-time support, such as libraries and implementations of basic transformation strategies.

There is no separate run-time package available in the `strj` suite yet. The presence of the compiler in this package significantly increases the total size of the archive. It would be possible to use a Java shrinker tool like ProGuard [12] to strip the parts unnecessary for the run-time of the element generator from the `strj` package.

The execution time of the element generator, which is now not a native executable but is executable Java bytecode, has also increased, as shown in *Table 6.1*. The measurements were done on a virtual machine running Fedora Linux 11, the virtual machine (VirtualBox) was installed on a system with AMD Athlon XP 2400+ CPU and DDR333 3 GB RAM running Windows XP SP3 operating system.

[1] To compile the grammar of the template language, it is necessary to use the original Stratego toolset. It is also possible to use the Spoofax editor to generate parse tables from the grammar and then use these parse tables in the compilation process with the strj compiler. This alternative approach is not part of the automated compilation script.

[2] Complete Stratego toolset is available for Linux and Mac OS X. Windows is supported via the Cygwin [6] environment. The execution time of some of the Stratego tools is a couple of times longer in the Cygwin environment than with native Linux builds. This makes the development on Windows platform lengthy.

Activity	strc	strj	slowdown
compilation	38.1 s	89.7 s	*2.3x*
run-time	2.0 s	8.3 s	*4.2x*

Table 6.1: Speed differences between Stratego Compiler (strc) and Stratego to Java (strj) compiler for compilation and execution of the element generator. The *compilation* time includes the time necessary to compile the Stratego-based connector element generator. *Run-time* measures the total running time of the compiled generator executed on a simple connector element template.

6.3 Previous Implementations

In comparison to the previous, initial implementation of the connector generator in Stratego [34], this the enables the generation of richer sets of connectors and brings a cross-platform connector generation solution. The connector generator allows the generation of connectors for the Java RMI technology, enables manipulation of method arguments in the adapted interfaces, and allows the generator to be deployed on more platforms.

In comparison to the initial implementation of the connector generator in SOFA [29], based on simple code templates and Java code generation, driven by manually written Java code, this thesis keeps the enhancements brought by the initial Stratego connector generator implementation. A defined syntax of the template language allows for the checking of syntax errors in the template before the generation process is launched. The template code contains constructs that control the generation, like `if`, `foreach` and other flow-control constructs. Additional Java code driving the code generation is not necessary, everything is contained in the templates.

Chapter 7

Related Work

This thesis enhances a connector generator which employs program transformations from domain-specific template language into target code for the software connectors. Connector generation is a complex task that is affected by many aspects, for example by the details of the component model or the target environment of the connectors. Therefore different approaches to connectors and their generation are used in various works.

Matougui and Beugnard [35] define the connector in a similar way to that which is used in this thesis, also the life cycle of connectors have common parts. They implemented a connector generator based on the standard Java RMI generator *JRMICompiler*. The generator is an enriched version of JRMICompiler with the functionality to generate RMI connectors. Therefore this generator is limited to creating connectors for the Java RMI technology.

The work by Radermacher [36] introduces a component model with connector generation. Among other features, the proposed component model aims to support real-time properties of the target environment. Every component port has two properties - *kind* and *type*. *Kind* denotes the communication style used for such port, e.g. data flow, messaging. *Type* is represented by a specific programming interface.

The connector generation uses the concept of connector templates with parameters (meta-variables). The generation tool generates the final code from the template and the input model. The generator is based on Acceleo [1] code generator based on model-to-text transformations. The input template contains meta-statements like references to model data or simple meta statements like `if` or `for`. A language called *OCL* is used to reference data in the input model. In some aspects it is similar to the *query module* described in this thesis, but contains larger set of constructs to gather required data from the model. The compilation of the template produces the final source code.

In comparison to the template language enhanced in this thesis, the template

language of Acceleo is rather general. For example writing transformations that would iterate and modify parameters of methods of business interfaces would require longer and less readable template code. The template is not checked against a defined syntax. Instead, the error checking of the template is done by compiling the template and checking for potential compilation errors.

The work [22] introduces extensions to the ArchJava language to support connectors and their generation. The language is extended with constructs to describe ports, their defined and required interfaces. The work introduces a concept of type-checking, that is executed during compile-time, this is implemented by user-defined code which uses Java reflections.

Java reflections are also used during the run-time of the connector to pass messages between components. There is a lookup of the invoked method by its name and arguments are passed as an array. This additional processing of the communication affects the performance of the connector.

Apache Tuscany [3] is an implementation of the Service Component Architecture (SCA) standard which defines components as building blocks to assemble business solutions. The communication of components is established during the assembly of the system. To connect components, SCA defines a concept of *Bindings* which is similar to connectors, they bind external connection points of the components. There is a support for example for Web service, JMS and EJB Session bindings.

The Thorn [24] scripting language inspired this work with the list manipulation constructs like `count` or `apply`. As a scripting language, the Thorn language aims to be general, so a more domain-specific approach was chosen in this thesis.

The R-OSGi framework [13] provides distribution, for arbitrary OSGi framework implementations. Connectors in R-OSGi are called Network Channels. R-OSGi provides its own network channels and allows for the implementation of custom ones. This is done by registering a new *NetworkChannelFactory* service which should return instances of *NetworkChannel*. The network channel then negotiates the communication via a special kind of OSGi messages. There is no special aid to help in mplementing own network channels in R-OSGi.

Fractal [27] is a component model with remote components communication support. The Fractal RMI API enables remote method calls between the components. There is a special bootstrap component which is remotely accessible and provides a component factory to remotely instantiate components. Fractal RMI uses the ASM bytecode library for generating communication stubs. This approach is therefore limited to RMI and method invocation communication style.

Recently, Fractal introduced a concept of *Fractal Binding Factory* which allows for implementing wider sets of connectors than just for Java RMI. Different

communication styles are supported too.

One of the tasks for connectors and their generation, is the adaptation of different interfaces between components (or components and connectors). The topic of interface adaptation is important in other areas too. The work [32] proposes an approach for Web service adaptation. It prescribes a set of XSL transformation to convert messages across clients and servers with different signatures of Web service protocols. However, the adaptation transformations, have to be implemented and selected manually for every Web service.

Next issue targeted by this thesis is code generation. There are many tools for code generation based on code templates, one of them is the already mentioned Acceleo. Another similar tool is Xpand [19]. On its input, there is also a model and a template. The template references data, in the model and during the generation, the data get substituted into the template. It is possible to implement modifications of the model in Java. Implementation of such transformations can get more verbose than when using specialized Stratego language. The template language is again general. Velocity [4] is another templating engine which is targeted towards web development.

The general templating engines typically do not provide, real syntax checking of the template. They do perform error checking by compiling the template, but there still can be syntax errors in the generated file as its syntax is not reflected during the generation process. The parts of the template that are not part of the template syntax are typically treated as plain text strings.

Tools that support working with syntax and allow the defining of new grammars for languages, are for example ANTLR [2] or Xtext [20]. Up to recent versions, ANTLR supported the definition of grammars and parsing using the user-defined grammars. It was also possible to construct custom trees from the input by using a special grammar that prescribed which nodes should get created. The rewrite rules that produce tree nodes contained lexical syntax in them. Since ANTLR version 3.1 the parser is able to produce AST and there is support to transform AST using transformation steps. This better, separated approach is more similar to the approach in Stratego. However, the transformation part of ANTLR is quite new and not as rich in functionality as Stratego.

In Xtext it is possible to define a grammar of domain-specific languages and generate parsers from it. The difference is that the internal representation of the parsed input will not be an abstract syntax tree, but it will be an EMF [7] model. The combination of Xtext and Xpand tools is close to the concept of Stratego. It is possible to write transformation strategies on the internal representation (EMF model). Originally this was possible only by writing additional Java code, which resulted in not very readable transformations. Later specialized transformation languages like ATL [5] emerged. This allows for implementing the transformations

in a concise way.

Definition of new domain-specific languages can be made easier by using editor specialized for this purpose. IMP [8] is a platform for implementing development environments for domain-specific languages. The Spoofax editor introduced in this thesis is based on this platform.

This thesis described an approach where a grammar of an existing language (Java) is mixed with additional domain-specific language. This concept of mixing an established language with with domain-specific extensions is utilized in the MPS [10] environment. The editor of this environment allows to render additional constructs in the base language in a novel and very readable way.

Chapter 8

Conclusion and Future Work

The goal of this thesis, was to enhance the connector generation process, in an existing connector generator, based on the Stratego toolset. The thesis aimed to allow the generation of a wider set of connectors and find a cross-platform solution for the connector generation process.

This thesis extended the connector template language with new constructs. These allow for abstract manipulations in interfaces adapted by the connector. It is possible to manipulate with arguments and return values of the methods. Manipulation of method arguments of interfaces is a common task with connectors which implement different network technologies. By adding the support for these manipulations directly into the connector template language, this work helps the connector developers to produce concise template code and better express the needs of the domain of software connectors.

The enhanced connector template language allowed to implement connector templates for the Java RMI technology. These were integrated and tested with the SOFA component system.

The adoption of the Stratego to Java compiler was part of the cross-platform solution for connector generation. The compiler allows the compiling of Stratego transformation programs into Java. This resulted in being able to keep most of the functionality of the original Stratego toolset, with smaller changes to suit the requirements of the compiler. Other technical changes were performed to eliminate any remaining sources of platform dependencies.

The cross-platform solution was also successfully tested on the Windows platform where previously it was not possible to launch the connector generator based on Stratego.

This thesis described the connector generation as a part of standard development process with components. It explained the roles in the development process with components and connector generation, and their required domain knowledge.

8.1 Future Work

The *concrete syntax* is a feature of Stratego toolset, that helps to express code transformations in the syntax of the target language. The current implementation of the connector element generation, defines the transformations, by denoting the specific terms in the transformed abstract syntax tree of the template. The usage of concrete syntax could bring better readability and maintainability to certain specific parts of the connector generator implementation. The support of concrete syntax in current versions of Stratego tools is mature enough.

The Spoofax IDE supports launching of Stratego transformation strategies. With some necessary technical changes in the source code of the connector element generator, the Spoofax IDE could be further utilized to execute the generator. This could be further used for some on-the-fly template testing and also for smoother connector generator testing.

When new connectors are implemented in the future, new common operations might be identified in the code of connector templates. The process of connector generator development should keep up with the needs of the domain of connectors. Often performed tasks should be supported by the template language itself. The future extensions of the template language depend on the needs discovered when implementing different middleware technologies.

The connector element repository is a static part of the connector generator. Even if easily configurable, adding additional elements into the repository requires altering the source code base of the connector generator. Addition of new connector elements could be made possible without the need to edit the internal configuration of the connector generator. The users of the component system then would be able to add new connector elements with new desired middleware technologies into the component system transparently. As the configuration is already loaded dynamically when executing the connector generator, this would require only minimal technical changes in the connector generator code.

Bibliography

[1] Acceleo - transforming models into code, http://www.eclipse.org/acceleo/.

[2] ANTLR Parser Generator, http://www.antlr.org/.

[3] Apache Tuscany, http://tuscany.apache.org/.

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

[5] ATL - a model transformation technology, http://www.eclipse.org/atl/.

[6] Cygwin: Linux-like environment for Windows, http://www.cygwin.com/.

[7] Eclipse Modeling Framework Project (EMF), http://www.eclipse.org/modeling/emf/.

[8] IMP: The IDE Meta-Tooling Platform, http://www.eclipse.org/imp/.

[9] Oracle, Java Native Interface Specification, http://download.oracle.com/docs/cd/E17476_01/javase/1.5.0/docs/guide/jni/spec/jniTOC.html.

[10] MPS, the Meta Programming System, http://www.jetbrains.com/mps/index.html.

[11] PHP: Arrays, http://php.net/manual/en/language.types.array.php.

[12] ProGuard - Java class file shrinker, optimizer, obfuscator, and preverifier, http://proguard.sourceforge.net/.

[13] R-OSGi - transparent OSGi remote extension for distributed services, http://r-osgi.sourceforge.net/.

[14] SOFA component system, http://sofa.ow2.org/.

[15] The Spoofax Language Workbench, http://strategoxt.org/Spoofax.

[16] Stratego / STRJ: The Stratego-to-Java Compiler, http://strategoxt.org/Stratego/STRJ.

[17] Stratego Compiler, http://strategoxt.org/Stratego/StrategoCompiler.

[18] Stratego Program Transformation Language, http://strategoxt.org/.

[19] Xpand template language, http://wiki.eclipse.org/Xpand.

[20] Xtext - Language Development Framework, http://www.eclipse.org/Xtext/.

[21] Object Management Group, Deployment and Configuration of Component Based Distributed Applications Specification, http://www.omg.org/docs/ptc/04-08-02.pdf, 2004.

[22] Aldrich, J., Sazawal, V., Chambers, C., and Notkin, D. Language Support for Connector Abstractions. In *ECOOP 2003 – Object-Oriented Programming* (2003), vol. 2743 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 179–204.

[23] Bálek, D., and Plášil, F. Software Connectors and their Role in Component Deployment. In *Proceedings of the IFIP TC6 / WG6.1 Third International Working Conference on New Developments in Distributed Applications and Interoperable Systems* (Deventer, The Netherlands, The Netherlands, 2001), Kluwer, B.V., pp. 69–84.

[24] Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strniša, R., Vitek, J., and Wrigstad, T. Thorn: robust, concurrent, extensible scripting on the JVM. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications* (New York, NY, USA, 2009), ACM, pp. 117–136.

[25] Bravenboer, M., de Groot, R., and Visser, E. MetaBorg in Action: Examples of Domain-specific Language Embedding and Assimilation using Stratego/XT. In *Participants Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)* (Braga, Portugal, July 2005).

[26] Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. Program Transformation with Scoped Dynamic Rewrite Rules. *Fundam. Inf. 69*, 1-2 (2005), 123–178.

[27] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. The Fractal component model and its support in Java. In *Software: Practice and Experience* (2006), Wiley InterScience, pp. 1257 – 1284.

[28] Bulej, L., and Bures, T. Using Connectors for Deployment of Heterogeneous Applications in the Context of OMG D&C Specification, 2005, http://citeseer.ist.psu.edu/bulej05using.html.

[29] 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.

[30] Crnkovic, I., Larsson, S., and Chaudron, M. Component-based Development Process and Component Lifecycle. In *International Conference on Software Engineering Advances* (2006).

[31] Fowler, M. *DSL Book WIP*. http://martinfowler.com/dslwip/, 2009.

[32] Iyer, A., Smith, G., Roe, P., and Pobar, J. An Example of Web Service Adaptation to Support B2B Integration.

[33] Kelly, S., and Tolvanen, J.-P. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Pr, March 2008.

[34] 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, 2006.

[35] Matougui, S., and Beugnard, A. How to Implement Software Connectors? A Reusable, Abstract and Adaptable Connector. In *Distributed Applications and Interoperable Systems* (2005), vol. 3543 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 83–94.

[36] Radermacher, A., Cuccuru, A., Gerard, S., and Terrier, F. Generating execution infrastructures for component-oriented specifications with a model driven toolchain: a case study for MARTE's GCM and real-time annotations. In *GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering* (New York, NY, USA, 2009), ACM, pp. 127–136.

[37] Robert, S., Radermacher, A., Seignole, V., Gérard, S., Watine, V., and Terrier, F. Enhancing Interaction Support In The CORBA Component Model.

Appendix A

Connector Templates

This thesis introduced template constructs that allowed implementation of Java Remote Method Invocation connectors, namely RMI skeleton and RMI stub. The code below are templates of connector elements for RMI skeleton and RMI stub. The constructs introduced by this thesis are used at the end of each template, inside of the method template block. This block specifies abstract implementation of methods of a target adapted interface.

A.1 RMI Skeleton

```
package ${package};

/* imports */
import org.objectweb.dsrg.connector.*;
import java.rmi.server.UID;
import java.rmi.Naming;
import java.net.MalformedURLException;
import java.rmi.RemoteException;

/**
 * RMI skeleton.
 *
 * A skeleton element has a server remote port "line" through which
 * remote clients connect to the target object, which is bound to
 * the "call" required port.
 *
 * This RMI skeleton always provides "rmi" reference to itself in
 * the remote reference bundle, because it resolves connector references
 * returned instead of interface references in business methods.
 *
 * Consequently, this element never invalidates its "line" remote port.
 */
element rmi_skeleton {
    /**
     * Local target of business method invocations.
     */
    protected ${ports.port(name=call).signature} target;

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

    protected org.objectweb.dsrg.connector.mgr.GlobalConnectorManager gcm;

    /**
     * Reference to parent connector unit.
```

```
     */
    protected ConnectorUnit parentUnit;

    protected boolean isTopLevel;

    /**
     * Name of this element in RMI registry.
     */
    protected String regName;

  /*
   * Name of java property - it cannot be in cod because of grammar restriction.
   */
  public static final String PROPERTY_RMI = "connector.gcm.rmihost";


    /* ********************************************************************
     * CONSTRUCTORS
     * ********************************************************************/
    public ${classname} (
            ConnectorUnit parentUnit,
            boolean isTopLevel)
                throws RemoteException,
                  MalformedURLException,
                        ElementLinkException,
                        org.objectweb.dsrg.connector.mgr.ConnectorManagerException {
        String GCMHost = null, GCMPort = null;

        GCMHost = System.getProperty ("connector.gcm.rmihost");
        GCMPort = System.getProperty ("connector.gcm.rmiport", "2008");

        this.parentUnit = parentUnit;
        this.isTopLevel = isTopLevel;

        // FIXME cannot write String rmiHost = System.getProperty(PROPERTY_RMI); because the output is ambigious
  if (GCMHost == null) {
            throw new org.objectweb.dsrg.connector.mgr.ConnectorManagerException(
                "Property 'connector.gcm.rmihost' must be set to a location of Global Connector Manager.");
        }

        /*uid = new UID ();*/
        regName = "//" + GCMHost + ":" + GCMPort + "/connector/element/rmi/"+ new UID().toString();

        java.rmi.Naming.rebind(regName, this);

        dcm = org.objectweb.dsrg.connector.mgr.
            DockConnectorManagerHelper.getDockConnectorManager ();
        gcm = org.objectweb.dsrg.connector.mgr.
            GlobalConnectorManagerHelper.getGlobalConnectorManager ();
  }

    /* ********************************************************************
     * Element Methods
     * ********************************************************************/

  $import("ElementMethodsImpl.ellang")$

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

implements interface ElementLocalClient {

    /**
     * Binds a given required port of this element to a target object using
     * a local object reference.
     */
    public void bindElPort (String portName, Object target)
        throws org.objectweb.dsrg.connector.ElementLinkException {

        if ("call".equals (portName)) {
            this.target = (${ports.port(name=call).signature}) target;

        } else {
            throw new org.objectweb.dsrg.connector.ElementLinkException (
                "Attempt to bind non-existent required port '"+ portName +"'.");
        }
    }


    /**
     * Unbinds a given required port of this element from its target.
     */
    public void unbindElPort (String portName)
```

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

        if ("call".equals (portName)) {
            this.target = null;

        } else {
            throw new org.objectweb.dsrg.connector.ElementLinkException (
                "Attempt to unbind non-existent required port '"+ portName +"'.");
        }
    }
}


    /* ********************************************************************
     * ElementRemoteServer Methods
     * ********************************************************************/

implements interface ElementRemoteServer {
    /**
     * Provides a bundle of remote references to the given remote server port.
     * Since this is an RMI skeleton, "rmi" reference is returned in the
     * reference bundle.
     */
    public org.objectweb.dsrg.connector.RemoteRefBundle lookupElRemotePort (
            String portName)
        throws org.objectweb.dsrg.connector.ElementLinkException {

        org.objectweb.dsrg.connector.RemoteRefBundle result =
            new org.objectweb.dsrg.connector.RemoteRefBundle ();

        if ("line".equals (portName)) {
            result.addRef (
                new org.objectweb.dsrg.connector.RemoteRef ("rmi", regName));

        } else {
            throw new org.objectweb.dsrg.connector.ElementLinkException (
                "Attempt to look up non-existent remote port '"+ portName +"'.");
        }

        return result;
    }


    /**
     * Returns the names of remote ports supported by this element.
     */
    public String [] listElRemotePorts () {
        return new String [] { "line" };
    }
}


    /* ********************************************************************
     *  Methods method of target iface.
     * ********************************************************************/
implements interface ${ports.port(name=line).signature} {
  method template {
    // Buffer for the return value
    ${method.declareReturnValue}

    // The last argument is the call context, process it.
    org.objectweb.dsrg.sofa.SOFAThreadHelper.setCallContext((String) $peek(method.args)$);
    $pop(method.args)$

    // Number of method arguments of non-primitive type.
    $set decoderNeeded = count(ARG in method.args where !method.args.ARG.type.isPrimitive)$
    $if (decoderNeeded)$
    try {
      // Non-primitive arguments require conversion using the decoder.
      org.objectweb.dsrg.connector.rmi.RMIObjectDecoder rmiDecoder =
        new org.objectweb.dsrg.connector.rmi.RMIObjectDecoder();

      // Wrap the arguments in an adaptaion call.
      $apply(rmiDecoder.adaptObject(ARG) for ARG in method.args where !method.args.ARG.type.isPrimitive)$

      // Call the target method, remember the return value.
      $setReturnValue this.target.${method.name}($implode(method.args)$)$
    } catch (org.objectweb.dsrg.connector.rmi.RMIObjectAdaptorException e) {
      throw new org.objectweb.dsrg.connector.ConnectorTransportException (e);
    }
    $else$
      // All arguments primitive, put them in directly.
      $setReturnValue this.target.${method.name}($implode(method.args)$)$
    $end$
```

```
    $if (!method.returnType.isPrimitive) $
      try {
        // Non-primitive return value requires a conversion.
        return (new org.objectweb.dsrg.connector.rmi.RMIObjectEncoder())
          .adaptObject(${method.returnVar});
      } catch (org.objectweb.dsrg.connector.rmi.RMIObjectAdaptorException e) {
        throw new org.objectweb.dsrg.connector.ConnectorTransportException (e);
      }
    $else$
      // Primitive return value does not require a conversion.
      ${method.returnStm}
    $end$
  }
}

/* Final class must extend this, required by Java RMI */
inherits java.rmi.server.UnicastRemoteObject
{
}

}
```

Listing A.1: Source code of the RMI Skeleton connector element template.

A.2 RMI Stub

```
package ${package};

import org.objectweb.dsrg.connector.ElementLocalServer;
import org.objectweb.dsrg.connector.ElementRemoteClient;

/**
 * RMI stub.
 *
 * A stub element has a client remote port "line" through which
 * it connects to the target object. When binding the "line" port
 * to a remote target using a bundle of references, a local stub
 * is only interested in the "rmi" reference.
 *
 * This RMI stub uses "rmi" reference from the reference bundle and always
 * returns itself as the reference to the provided "call" port, because it
 * creates connectors for interface references in bussines methods.
 *
 * Consequently, this element never invalidates its "call" provided port.
 */
element rmi_stub {
    /**
     * Local target of business method invocations.
     */
    protected ${ports.port(name=line).signature} target;

    /**
     * A bundle of remote references to the target of business method
     * invocations.
     */
    protected org.objectweb.dsrg.connector.RemoteRefBundle targetRemoteRef;

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

    protected org.objectweb.dsrg.connector.mgr.GlobalConnectorManager gcm;

    /*
     * Reference to parent connector unit.
     */
    protected org.objectweb.dsrg.connector.ConnectorUnit parentUnit;

    protected boolean isTopLevel;


    /* ******************************************************************
     * CONSTRUCTORS
     * ******************************************************************/

    public ${classname} (
```

```
            org.objectweb.dsrg.connector.ConnectorUnit parentUnit,
            boolean isTopLevel)
        throws org.objectweb.dsrg.connector.ElementLinkException {

        this.parentUnit = parentUnit;
        this.isTopLevel = isTopLevel;

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

        try {
            gcm = org.objectweb.dsrg.connector.mgr.
                GlobalConnectorManagerHelper.getGlobalConnectorManager ();

        } catch (org.objectweb.dsrg.connector.mgr.ConnectorManagerException e) {
            throw new org.objectweb.dsrg.connector.ElementLinkException (e);
        }
    }

    /* ************************************************************************
     * Element Methods
     * ************************************************************************/

$import("ElementMethodsImpl.ellang")$

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

implements interface ElementLocalServer {
    /**
     * Queries given provided port for a local object reference.
     */
    public Object lookupElPort (String portName)
      throws org.objectweb.dsrg.connector.ElementLinkException {

      if ("call".equals (portName)) {
        /*
         * Always return reference to itself when asked for
         * a reference to the "call" port.
         */
        if (isTopLevel) {
          dcm.reregisterConnectorUnitReference (
            parentUnit, "call", this);
        }

        return this;

      } else {
        throw new org.objectweb.dsrg.connector.ElementLinkException (
          "Attempt to look up non-existent provided port '"+ portName +"'.");
      }
    }
}


    /* ************************************************************************
     * ElementRemoteClient Methods
     * ************************************************************************/

implements interface ElementRemoteClient {
    /**
     * Binds a remote client port of this element to a remote server port
     * using a bundle of named references. This RMI stub has one remote
     * client port "line" and requires the reference bundle to contain a
     * reference named "rmi" which can be then resolved to a local
     * object reference.
     */
    public void bindElRemotePort (
        String portName,
        org.objectweb.dsrg.connector.RemoteRefBundle refBundle)
      throws org.objectweb.dsrg.connector.ElementLinkException {

      if ("line".equals(portName)) {
        org.objectweb.dsrg.connector.RemoteRef rmiRef;

        rmiRef = refBundle.getRef ("rmi");
        if (rmiRef != null) {

          try {
            target = (${ports.port(name=line).signature}) java.rmi.
              Naming.lookup (rmiRef.stringifiedRef);

          } catch (java.rmi.RemoteException e) {
```

```
            throw new org.objectweb.dsrg.connector.ElementLinkException (e);
          } catch (java.net.MalformedURLException e) {
            throw new org.objectweb.dsrg.connector.ElementLinkException (e);
          } catch (java.rmi.NotBoundException e) {
            throw new org.objectweb.dsrg.connector.ElementLinkException (e);
          }

          targetRemoteRef = refBundle;

        } else {
          target = null;
        }

      } else {
        throw new org.objectweb.dsrg.connector.ElementLinkException (
          "Attempt to bind non-existent remote port '"+ portName +"'.");
      }
    }


    /**
     * Unbinds a given remote client port of this element
     * from a remote server port.
     */
    public void unbindElRemotePort (String portName)
      throws org.objectweb.dsrg.connector.ElementLinkException {

      if ("line".equals (portName)) {
        target = null;
        targetRemoteRef = null;

      } else {
        throw new org.objectweb.dsrg.connector.ElementLinkException (
          "Attempt to unbind non-existent remote port '"+ portName +"'.");
      }
    }


    /**
     * Queries given remote client port of this element
     * for its target reference bundle.
     */
    public org.objectweb.dsrg.connector.RemoteRefBundle getElRemoteTarget (
        String portName)
      throws org.objectweb.dsrg.connector.ElementLinkException {

      if ("line".equals (portName)) {
        return targetRemoteRef;

      } else {
        throw new org.objectweb.dsrg.connector.ElementLinkException (
          "Attempt to query target of non-existent remote port '"+ portName +"'.");
      }
    }
}


  /* ************************************************************************
   * Methods of target iface
   * ************************************************************************/
implements interface ${ports.port(name=call).signature} {
  method template {
    // Buffer for the return value
    ${method.declareReturnValue}

    // Number of arguments of non-primitive types.
    $set encoderNeeded = count(ARG in method.args where !method.args.ARG.type.isPrimitive)$
    $if (encoderNeeded)$
      // Non-primitive arguments require conversion using the encoder.
      org.objectweb.dsrg.connector.rmi.RMIObjectEncoder rmiEncoder
        = new org.objectweb.dsrg.connector.rmi.RMIObjectEncoder();

      // Wrap the arguments in an adaptaion call.
      $apply(rmiEncoder.adaptObject(ARG) for ARG in method.args
        where !method.args.ARG.type.isPrimitive)$
    $end$

    try {
      // Pass the call-context as the last argument.
      $append(method.args, org.objectweb.dsrg.sofa.SOFAThreadHelper.getCallContext())$

      // Call the target method, remember the return value.
      $setReturnValue this.target.${method.name}($implode(method.args)$)$
    }
```

```
      $if (encoderNeeded)$
      catch (org.objectweb.dsrg.connector.rmi.RMIObjectAdaptorException e) {
        // Adaptation exception can occur
        // This catch branch will be added only if the encoder was present.
        throw new org.objectweb.dsrg.connector.ConnectorTransportException (e);
      }
      $end$
      catch (java.rmi.RemoteException exc) {
        throw new org.objectweb.dsrg.connector.ConnectorTransportException (exc);
      }

      $if (!method.returnType.isPrimitive) $
        try {
          // Non-primitive return value requires a conversion.
              return (new org.objectweb.dsrg.connector.rmi.RMIObjectDecoder()).adaptObject(${method.returnVar});
            } catch (org.objectweb.dsrg.connector.rmi.RMIObjectAdaptorException e) {
              throw new org.objectweb.dsrg.connector.ConnectorTransportException (e);
            }
            $else$
              // Primitive return value does not require a conversion.
              ${method.returnStm}
            $end$
    }
  }
}
```

Listing A.2: Source code of the RMI Stub connector element template.

Appendix B

Online Materials

SOFA Homepage
http://sofa.ow2.org/

SOFA Subversion
svn://svn.forge.objectweb.org/svnroot/sofa

SOFA SVN Browser
http://websvn.ow2.org/listing.php?repname=sofa

SOFA Project
http://forge.ow2.org/projects/sofa/

Implementation part of this work
svn://svn.forge.objectweb.org/
svnroot/sofa/trunk/congen/branches/strj/congen-core

Appendix C

Contents of the attached CD

The CD-ROM that comes with this thesis contains implementation of the connector generator discussed in this thesis and other relevant files. The structure of the CD is following:

/editor/
: Connector template editor based on Spoofax (Eclipse).

/implementation/
: Implementation of the connector generator enhanced in this thesis.

/prereq/
: Prerequisites such as Stratego, ATerm library, SDF Bundle and Java Front library.

/sofa/
: Installation of the SOFA component system integrated with the enhanced connector generator implemented in this thesis.

/thesis/
: Text of this thesis.

/readme.txt
: Description of the structure of the CD.