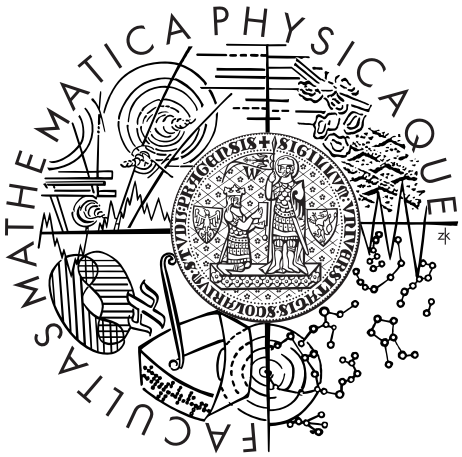
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Lukáš Ježek

C-language code generator for SOFA 2

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Tomáš Bureš, PhD.
Study programme: Computer Science
Specialization: Software Systems

Prague 2011

First of all, I would like to thank my supervisor Tomáš Bureš for his important comments and suggestions. I also thank my colleague Konrad Karczewski for his valuable observations and help with writing. Finally, my big thanks belong to my family for their support and patience.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 4, 2011

Název práce: Generátor C kódu pro SOFA 2
Autor: Lukáš Ježek
Katedra (ústav): Katedra spolehlivých a distribuovaných systémů
Vedoucí diplomové práce: RNDr. Tomáš Bureš, PhD.
E-mail vedoucího: tomas.bures@d3s.mff.cuni.cz

Abstrakt: SOFA 2 je komponentový systém založený na hierarchickém komponentovém modelu. K návrhu aplikace slouží jazyk ADL, chování komponent je popsáno behaviorálními protokoly, dále systém umožňuje dynamickou rekonfiguraci komponet a modeluje propojení mezi jednotlivými komponentami pomocí softwarových konektorů. Ty umožňují transparentní rozdistribuování vyvíjené aplikace mezi více počítačů. Implementace konektorů může být automaticky generována, SOFA 2 je primárně vyvíjena pro jazyk Java, proto obsahuje generátor Javovských konektorů. Cílem této magisterské práce je navrhnout generátor kódu pro jazyk C a zaintegrovat tento generátor do stávající struktury generátoru konektorů v systému SOFA 2. Automatické generování konektorů v jazyce C by mělo umožnit transparentní propojení komponent implementovaných v jazyce C.

Navržený generátor C kódu je založený na konceptu transformace šablon, kde je vstupní šablona, která obsahuje kombinaci cílového C kódu a speciálně vyvinutého skriptovacího jazyka, převedena na čistý C kód. Pro vyhodnocení šablon je použito strategické přepisování abstraktních syntaktických stromů poskytnuté frameworkem Stratego/XT .
Klíčová slova: komponenty, softwarové konektory, generátor kódu, abstraktní syntaktický strom, Stratego/XT

Title: C-language code generator for SOFA 2
Author: Lukáš Ježek
Department: Department of Distributed and Dependable Systems
Supervisor of the master thesis: RNDr. Tomáš Bureš, PhD.
Supervisor's e-mail address: tomas.bures@d3s.mff.cuni.cz

Abstract: SOFA 2 is a component system employing hierarchically composed components. It provides ADL-based design, behavior specification using behavior protocols, dynamic reconfiguration of the components, and modeling of the component communication by software connectors. This allows seamless and transparent distribution of component applications. The connectors can be automatically generated, SOFA 2 contains Java connector generator allowing to connect components with Java interfaces. The aim of this thesis is to implement C code generator and integrate it into the current SOFA 2 connector generator framework, so that C connectors can be automatically generated and thus components written in C language can be transparently connected in distributed environment.

The proposed C code generator is based on the concept of template transformation, where templates containing mixture of C code and a scripting Domain Specific Language are transformed to a pure C code. Strategic term rewriting method provided by Stratego/XT framework is used for evaluation of the scripts within the templates.
Keywords: components, software connectors, code generator, abstract syntax trees, Stratego/XT

Contents

Chapter 1

Introduction and motivation

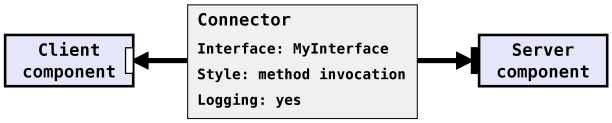
Various computer systems have become an inherent part of our everyday lives. Hardware engineers are trying to make all these systems, ranging from microwave owens to desktop computers, run faster, which enables them to perform very sophisticated computations and operations on the software level. This triggered the evolution of the software runnable on these machines from very simple computations (e.g. a calculator) to very complex programs like todays operating systems or vector graphics applications. The more complex software we have, the harder it is to maintain it. This led to a need for discovering methods of developing both scalable and maintainable software, and emergence of a separate engineering discipline focused on software design and the process of its development, we call it software engineering. One of the paradigms that the software engineering proposed is called component-based software engineering (CBSE). This fast evolving approach allows us to create large-scale software systems from smaller pieces called components. Each component has exactly specified functionality and the most important fact is that its communication with other components is restricted to well defined interfaces. This makes the components independent of each other and it also allows one component to be easily reused in several software applications or several places within the same application. The reusability of components allows for faster development of highly scalable applications and the strict separation of concerns between the components leads to more robust and error-free applications.

Tools that help the developers create applications consisting of components are called component systems. Many big companies have developed their own proprietary component systems both for their internal use and for the outside world, which enables the developers to create applications interacting with the companies' systems more easily. For example, Microsoft has a series of component systems starting from COM, which was extended for the use in distributed environments to DCOM [6] and finally it was deprecated by the introduction of .NET Framework [15]. Oracle offers the Enterprise JavaBeans (EJB [10]) component system developed by Sun Microsystems. Object Management Group used their CORBA middleware [5] as a base for CORBA Component model (CCM [7]). Each of these systems provide a wide set of development and runtime tools, but at the same time they lack some advanced features like hierarchical components or the ability to choose between various communication styles for the connections between the components. This provides space for another group of component

systems, mainly developed on the academic soil. For example SOFA 2 [22] and Fractal [11] component systems belong to this group.

All the component systems agree on seeing a component as a unit which only includes pure business logic. It should not care about other things like the communication with other components. This part is handled by the component system. And the handling of the communication is the process where the component systems vary significantly. The proprietary systems usually support only one predefined middleware, e.g. MSRPC [13] in DCOM, Java RMI or RMI-IIOP [12] in EJB or .NET Remoting [16] in .NET. On the other hand, the academic models usually define a new first-class entity called software connector, which stands on the same level as component. The connectors mediate all the communication between the components and the application designer is provided with the possibility to modify the parameters for each of the connectors. Mainly, he is allowed to choose from several communication styles (method invocation, messaging, streaming) and to define desired communication properties (logging, security, measurement) for each component interconnection. Figure 1.1 shows an example of connector as a mediator of communication between two components.

Figure 1.1: Modelling communication between components by a connector



The main purpose of the connectors is the separation of concerns, they take care of the communication logic of the application and the components only need to deal with the business logic. If the developer of a component application had to write the connectors by hand, then the component system would not save much work for him. This idea, and the fact that the code for connectors for one communication style usually varies only a little, gave rise to connector generators. Some component systems deploy only a simple stub/skeleton generators and the developer is then required to fill in the gaps in the generated code to call the component's code. Other component systems, like SOFA 2, contain a full connector generator, which takes a description of the connection (the interface used for the connection and other properties, e.g. the desired communication style) and produces a full connector code which implements the communication functionality (possible marshalling of data on the client end, sending the data to the server end, demarshalling the data and calling the appropriate code of the component).

There are several known and widely used approaches for code generation. The first and the most obvious one is direct generation of the target code by the generator. This method allows us to produce any text very easily, but the generated code has no structure, so we cannot be sure that it will be syntactically correct. One of the examples is generating HTML from PHP using `echo` or `print` commands. Another approach is to use a template in the target language with extension points denoted by special marks, the code for these special marks is then generated by

the generator and inserted into the template. This approach was used in the early version [37] of the connector generator for SOFA 2. This methods advantage is, that the template is kept completely separate from the generating code, so it is easier to maintain it and see its complete structure. Another approach to code generation uses templates consisting of a mixture of target language code and a DSL (Domain Specific Language) used for scripting. This template then goes through a transformation which removes all the scripting parts and we end up with the target code only. This method allows the most control about the target code, and with use of proper transformation tool (like Stratego/XT [25]) it can ensure that the syntax of the target code is correct. This method has been proposed for a newer version of the connector generator for SOFA 2 [47].

The main goal of this thesis is to design a C code generator suitable for generation of C connectors. The connector generator does not only consist of the code generator, it also has to deal with handling of the input arguments, running the code generator on the correct templates, preparing correct arguments for the generator including the business interfaces and providing a way to produce binaries from the generated code, so a second part of this thesis will focus on updating the current SOFA 2 connector generator to support the new C code generator. Finally, several sample code templates will be created so that the use of the generator can be shown and a sample application will be developed to show that this approach is feasible for larger use and that the generated code can be easily used by external code (e.g. later by components written in C).

1.1 Objectives

1. Choose fitting way for C code generation.
2. Design C code generator suitable for connector generation.
3. Modify the SOFA 2 connector generator so that it can use the newly proposed C code generator.
4. Create sample C templates that will allow generation of connectors for some of the communication styles.
5. Sample component application utilizing the connectors generated by the code generator will be created as a proof-of-usability.

1.2 Structure of the text

Chapter 2 describes SOFA 2 component system and the connector generator used in SOFA 2. Overview of Stratego/XT , the tool chosen for the code generation, follows in Chapter 3. After these introductory chapters, the tasks that need to be done are analysed and the goals of the thesis are reformulated to be more precise in Chapter 4.

Chapter 5 describes the created C grammar, which is used as a basis for template parsing and transformation. The problems of ambiguity of the C grammar are also discussed in that chapter. Proposed interface and structure of connector elements are introduced in Chapter 6 along with the DSL template scripting

language enhancements to support C code and target language specific transformations. The necessary modifications of the SOFA 2 connector generator to support both Java and C code generators are discussed in Chapter 7. This chapter also describes the connector generator parts dealing with C language structures (e.g. type information). Chapter 8 describes the created sample code templates and the choice of a suitable middleware for C language.

Then, a sample distributed application using the C connectors generated by the modified connector generator is presented, and the feasibility of the proposed solution is evaluated in Chapter 9. This chapter also contains overview of other code generation and transformation tools that were considered during the selection process. Finally, related work in area of C code generation is discussed in Chapter 10, and the work done on the solution of the C code generation problem is concluded in Chapter 11 along with the proposal of possible future work on that project.

Chapter 2

SOFA 2 component system

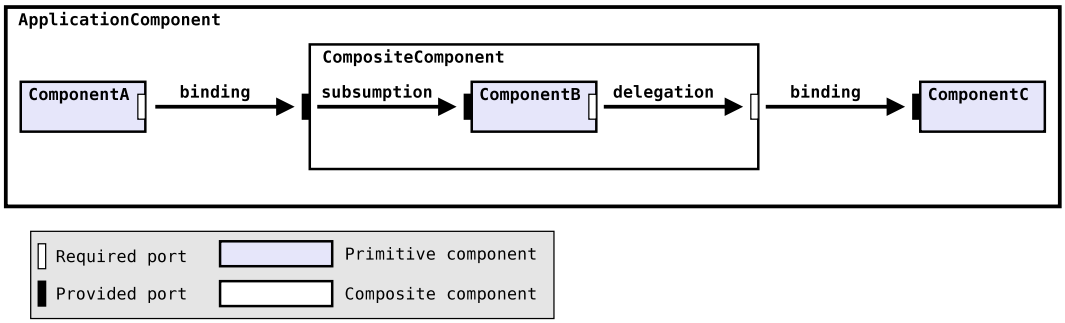
2.1 Component system overview

The core part of each component system is the component model it uses. SOFA 2 offers a hierarchical one, which distinguishes two types of components. *Primitive* components contain business logic implementation in some programming language (currently only Java is supported), whereas *composite* components are composed of other components (either primitive or composite ones). Each component has well-defined communication interfaces for communication with other components, of which we have two types - *provided* interfaces for the functionality that the component offers to other components, and *required* interfaces that the component uses for requesting service offered by other components, only bindings between the same types of interfaces are allowed. Each component also has a *controller* interface through which the component system communicates with the component (e.g. instantiates, starts, stops or configures it).

We can look at a component by two different views. The *black-box* view shows us the component's *frame*, which contains the definition of all the business interfaces that the component has, all the communication with other components goes through the interfaces in the component's frame. The *gray-box* view looks on the inner *architecture* of the component, it provides information about the contents of the component - which can either be a code implementing the business logic in case of primitive component, or definitions of frames of direct sub-components and their interconnections for a composite component. In the latter case also the bindings between the parent component's interfaces and the sub-components' interfaces are defined - these are called *subsumption* for (parent provided-child provided) binding and *delegation* for (child required-parent required) binding. The whole application is also described by a component, which has frame without any interface. Figure 2.1 shows an example of a simple component application.

The resulting component application does not need to run only on a single machine, it can be distributed to several computers, in SOFA 2 the computers capable of running the components are called *docks*, SOFA 2 *node* is a collection of docks. Each primitive component runs in one dock (one computer), but subcomponents of a composite component can be spread over several docks. This brings a new need to SOFA - to provide bindings for both local and remote communication. SOFA 2 uses the concept of *software connectors* as first-class entities

Figure 2.1: Sample component application architecture with all kinds of bindings



handling all the communication between components. Each binding between two components is represented by one instance of a connector. Component application designer is allowed to define properties of the bindings (desired communication style and other non-functional properties like logging or security). At application deployment time (when all the component architectures are known, and the programmer decided on which dock to run each of the components) the connectors can be generated, because at this time we know everything we need - whether to create local call connector or whether we need to employ some middleware, which communication style is required, and which interface is going to be communicated via the connector.

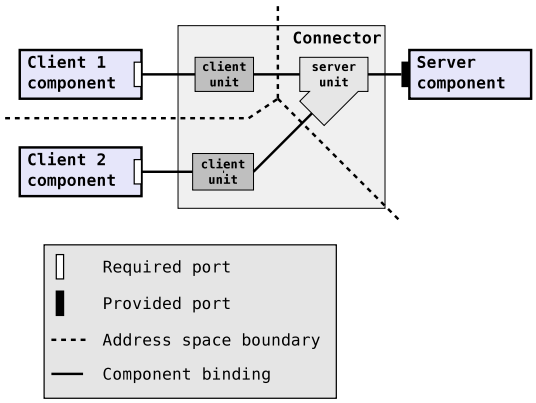
2.2 Connector generator

Adaptation of connectors in component systems has two reasons, firstly, they provide a way to *model* high-level requirements for component interactions (like communication style and other non-functional properties). Secondly, connectors are used to *implement* the prescribed interaction, they usually employ some middleware for remote communication and encapsulate it, so the communication is transparent to the component (the component does not have to deal directly with the middleware). Without automatical generation of the connector implementation they would only bring extra work into application development, which would lower their benefits (the biggest one being the separation of concerns). Thus there is a need to generate the connectors.

2.2.1 Connector model

One *connector* is usually spread over several address spaces (the communicating components might be placed on different deployment docks, thus on different computers) as seen on Figure 2.2, so it is necessary to be able to split it to smaller parts, *connector units*, which are capable of living in separate address spaces.

Figure 2.2: Remote binding between components realized by connector



2.2.2 Connector specification

The input of the connector generator comes from the parameters specified by the application designer, and additional requirements are set during deployment time (e.g. the location of the components or other restrictions given by the runtime environment). All these requirements are put together into a so-called *high-level connector specification*, a sample one is shown on Listing 2.1.

Listing 2.1: Sample high-level connector specification

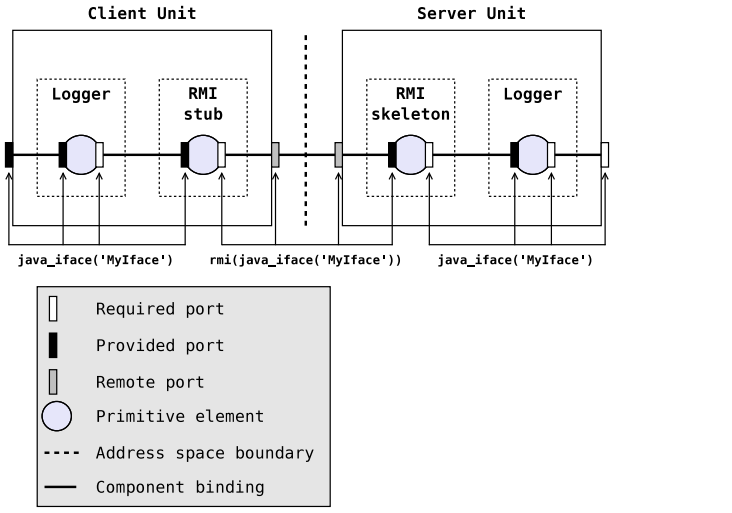
```
<specification>
  <unit name="client_unit" dock="dockA">
    <nfp-requirement predicate="nfp_mapping(Unit,'
        communication_style','method_invocation')"/>
    <nfp-requirement predicate="nfp_mapping(Unit,'logging',_)"/>
    <port name="call" type="provided" signature="java_iface('MyIface
        ')"/>
  </unit>

  <unit name="server_unit" dock="dockB">
    <nfp-requirement predicate="nfp_mapping(Unit,'
        communication_style','method_invocation')"/>
    <nfp-requirement predicate="nfp_mapping(Unit,'logging',_)"/>
    <port name="call" type="required" signature="java_iface('MyIface
        ')"/>
  </unit>
</specification>
```

However, this high-level specification is not directly used for generation of the connector source code because there is a huge semantic gap [39] between the high-level specification and the executable code. Thus *low-level connector configuration*, which is something close enough to both ends, is introduced. It views a connector in a component based fashion as a composition of *primitive connector elements* directly implementing some functionality, and *composite elements* which are composed of several sub-elements and describe their bindings. The low-level configuration thus sees each connector as a tree of elements where all leaves are primitive. The elements located in the first level of nesting are called *connector units* and each of them can be loaded in a different deployment dock.

Connector elements communicate only via designated *ports*, there are three types of ports in SOFA 2, *provided* where the element acts as a server, *required* where the element acts as a client, and *remote* ports which are used for modelling connections to other elements. Each element port has associated its actual signature (description of the interface used). The low-level specification contains also description of bindings, which are of two types. *Local bindings* are used between elements inside one connector unit and in communication with components and are always from required to provided or from provided to required port. The second type of bindings are *remote* ones, which are used for connecting different elements of one connector residing in different address spaces (different connector units), so they usually employ some kind of middleware. It is important to note, that a signature on two elements' ports can differ which provides the possibility to adapt the information transferred.

Sample low-level connector configuration for the high-level specification can be seen on Figure 2.3. We can see, that the connector has been split among two top-level composite elements (client and server unit) and four primitive elements (two loggers, rmi stub and rmi skeleton elements).

Figure 2.3: Sample low-level connector configuration



2.2.3 Architecture resolver

Architecture resolver transforms the high-level into the low-level connector specification. It has a database of all available element architectures (Listings 2.2 and 2.3 show sample element architectures) and tries to create a correct configuration of the whole connector from these elements so that it suits all the requirements in the high-level specification. If multiple possible configurations are found, the one with the lowest total *cost* is chosen and low-level connector configuration is created from it. The architecture resolver implements the resolving part in a Prolog program, which recursively searches through the space of possible connector architectures and locates the best one among them.

Listing 2.2: Element architecture of a composite element with two subelements

```
<element name="client_unit" type="rpc_client_unit" impl-class="
    ClientUnit">
  <architecture cost="0">
    <inst name="stub" type="stub"/>
    <binding port1="call" element2="stub" port2="call"/>
    <binding element1="stub" port1="line" port2="line"/>
  </architecture>

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

Listing 2.3: Primitive element architecture for RMI Stub

```
<element name="rmi_stub" type="stub" impl-class="RMIStub">
  <architecture cost="5">
    <port name="call" signature="I"/>
    <port name="line">
      <signature-entry ref-name="rmi" type="client"
          signature="rmi(I)"/>
    </port>
  </architecture>
  ...
</element>
```

2.2.4 Element generator

Having the low-level specification of the connector resolved, the identified elements can be created one after another. The input configuration specifies the subelements, signatures of their ports, and bindings among them, but it does not include the connector code - this part needs to be generated. For that purpose a new Domain Specific Language, ElLang, was designed and templates with a mixture of this scripting language and the target language code are created for each element type. In the process of code generation these templates are adapted to correspond to the configuration of the element (e.g. to reflect the interfaces on the element ports), and the code of the elements in the target language is created.

The element generator does not only generate the source code of the elements, but it is also used to adapt the communication interfaces if needed, to compile the generated code and to perform some other things that might be template specific - e.g. run a middleware stub generator. These tasks depend on the type of the element, so each element architecture descriptor contains a script with a list of actions to perform.

Modularity of the generator

Example action script can be seen on Listing 2.4, each of the commands actually invokes a corresponding Java class, which implement ActionInterface. The most important action is StrategoJGenerator which generates the code of the element

from its template, javac command compiles the generated source code, and delete action deletes the source code files (which are not necessary because they have already been compiled).

Listing 2.4: Sample action script associated to ServerUnit element

```
<element name="server_unit" type="rpc_server_unit"
    impl-class="ServerUnit">
...
  <script>
    <command action="StrategoJGenerator">
      <param name="class" value="ServerUnit" />
      <param name="template"
          value="ellang/compound_default.ellang" />
    </command>
    <command action="javac">
      <param name="class" value="ServerUnit" />
    </command>
    <command action="delete">
      <param name="source" value="ServerUnit" />
    </command>
  </script>
...
</element>
```

The actual code generation is based on transformation of file with code template to pure Java, the transformation is implemented in Stratego/XT framework, which will be introduced later in Chapter 3. The main goal of the action StrategoJGenerator is to prepare the connector element prerequisities, create an input XML file with element description (along with path to the template) and start the Stratego transformation.

ElLang language overview

The purpose of the template DSL is to enable scripting in the code generation. ElLang achieves this goal by introducing basic meta-statements (if, foreach, set), meta-variables (for numbers, strings and arrays of these) and some special constructs specific for the purpose of connector element generation. The transformation evaluates the template and gradually replaces the meta-statements in several stages until only code in the target language is left.

All the *meta-variables* can be referenced by their names in the scope of ElLang and by ${var} construct in the target language scope. Among the user defined meta-variables there are also predefined read-only meta-variables called *queries*, which provide access to the values in the input XML file and to a special dynamically created context information (e.g. function parameter types and names during generation of code for the given interface). The configuration stored in the XML file is hierarchical, thus the queries copy its structure and a dot symbol is used to navigate to lower levels. The queries offer also a counting operator ${ports.port#count}, which returns the number of subelements of a defined type in the parent element, here it would return the number of ports. Conditional queries are also allowed to perform a choice of an element, for example ${ports.port(name =call)} returns the subelement port under top-level element ports with sub-node name equal to call.

The assignment of a value to a meta-variable is the simplest *meta-statement* performed with a set command, e.g. $set cnt = elements.element#count$. Two basic control flow statements if and foreach, which work the same as in standard programming languages, are accompanied by a recursive version of foreach, which allows generation of nested structures, like series of Java-if commands in the following example:

```
$rforeach(PORT in ${ports.port(type=REMOTE)})$
  if ("${PORT.name}".equals(portName)) {
    /* some code */
  } else $recpoint$
$final$
  /* Code for the case when no
   * PORT.name matches the portName. */
$end$
```

The recpoint statement is replaced by the contents generated by the next loop of the rforeach, this proceeds until the last element of the array is processed and after that the remaining recpoint is replaced by the code in the final section. ElLang also allows splitting the template among several files (and thus have some generic parts in a single file) and including other templates by using $import ("included_template_path.ellang")$ meta-statement.

Among those generic constructs, special meta-statements designed for purposes of element generation are provided. The basic functionality that an element needs to perform is the mediation of the interface: for each method of the interface it needs to pack all the arguments and transfer them using the selected middleware in case of a stub element, or receive the packed parameters from the middleware and call some interface function in case of a skeleton element. This step usually involves implementation of interface-wrapper methods, and for this purpose a *method template* construct was created, its use is shown on Listing 2.5.

Listing 2.5: Using method template to implement an interface

```
implements interface ${ports.port(name=call).signature} {
  method template {
    ${method.declareReturnVar}
    /* generic stuff */
    $if (method.returnVar)$
      ${method.returnVar} =
        this.target.${method.name}(${method.variables});
    $else$
      this.target.${method.name}(${method.variables});
    $end$
    ${method.returnStm}
  }
}
```

ElLang also offers the possibility to create an element hierarchy, where simple elements implement just a basic functionality, and more complex elements can extend these simpler ones in specially marked extension points. Listing 2.6 shows both the code of a base element with definition of a single extension point and a more complex class extending the functionality of the basic one and redefining the extension point.

Listing 2.6: Extending basic elements in ElLang

```
element basic_element {
  public void ${classname}() {}
  public void printInfo() {
    System.println("Info for ${classname}:");

    $extPoint(ePrintInfo) $(
       System.println("Default implemention.");
    )$end$
  }
}

element complex_element extends basic_element {
  $defExtPoint(ePrintInfo)$
     System.println("Overriding functionality.");
  $end$
}
```

Stratego transformation

The flow of the STRATEGO transformation is depicted on Figure 2.4, which was based on work [48]. The element generator transformation, originally proposed by [47], first preprocesses the input XML element descriptor. At this stage, the ports and bindings are enhanced by the information about the elements on the other ends of the connections. The hierarchy of the created XML then denotes the hierarchy of the query meta-variables created for the element templates.

After this preprocessing stage, the template file of the element that is being generated is located and parsed.

The parsed template file is then evaluated in several steps:

1. Extends and import ElLang statements are evaluated. These statements might cause template expansion (the template of the extended element is also parsed and combined with the current one).

2. The extension points are replaced by their actual implementation (if a child template redefines an extension point of its parent template, then the new implementation is used and the old one is forgotten).

3. All the query meta-variables are evaluated and replaced by the values of the corresponding elements from the processed element descriptor.

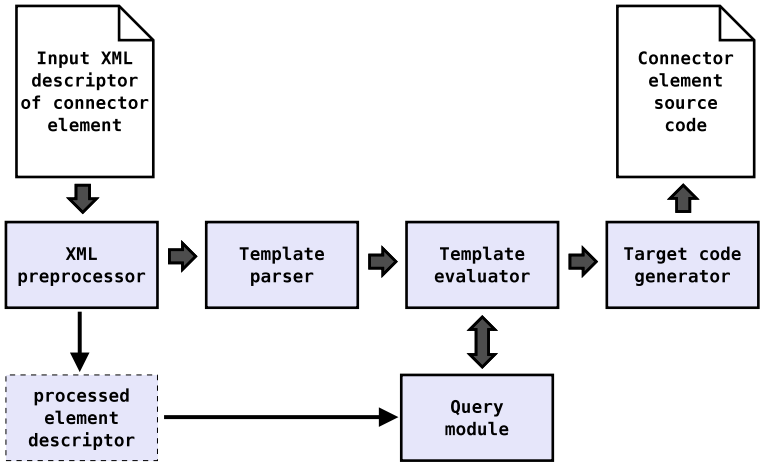
4. All the interface definitions are evaluated and the template methods defined inside them are expanded to the implementions of the corresponding business interface methods.

5. All the ElLang statements in the whole template are evaluated.

6. The rest of the ElLang nodes in the abstract syntax tree is processed and replaced by the target language nodes. Here the evaluation module also gets rid of the leftover parts from parsing (e.g. the constructors of rules for mixing ElLang and the target language).

Finally, the resulting abstract syntax tree is pretty-printed to a file denoted by the input XML element descriptor.

Figure 2.4: Workflow of Stratego element code generator



2.2.5 Type system

Connector generator used in SOFA introduced by [39] is designed to be as generic as possible, e.g. to support different component models and definition of interfaces in different formats. Thus a type system framework was introduced to shield the rest of the connector generator from the interfaces and to allow interface adaptation. The type system is used for specifying both generic types accessible by the rest of connector generator (e.g. Type, InterfaceDef or PrimitiveDefType) and specific interfaces (e.g. CORBA IDL, Java interface, C header file) and types native to particular programming languages, component systems and middlewares.

The main responsibilities of the type system are creation of a type description from different interface definitions (e.g. by reflection in Java), mapping such a type to a different type system (e.g. CDL interface to Java interface), modification of an interface (e.g. adding a parameter or an exception to all functions in it) and finally generating a source file containing the built interface for use in external tools and connector code (e.g. by Java compiler, CORBA IDL compiler). All the programming languages, middleware and component system specific features are brought into the type system in a form of plugins which deliver type factories.

The input description of a type is called symbolic type specifier and it is used to describe the desired interface type (e.g., rmi_iface (java_iface (" TestIface"))), this description is represented as a tree with functor name used as a name of a node and values as its subtrees. Type factories are then responsible for building types based on this symbolic description from bottom to top of the tree (thus always operating on resolved types), correct type factory for resolving the given subtree is chosen by the name of the root node of the subtree (the type factories know which types of nodes they support). Adaptation of the interface is also defined using a symbolic type specifier, type factory implementing the adaptation will then be used to adapt the type description (e.g. JavaTypeFactory also implements add_param(itf, type) specifier).

Interfaces are written back to files using type system specific interface writers, the type system also allows to define a compiler for the generated type (e.g. in Java we have rmi and javac compilers).

Chapter 3

Stratego/XT

3.1 Overview of Stratego/XT

Stratego/XT [25] is a framework that allows its users to define a grammar, parse a text according to the grammar, performs transformations on the parsed abstract syntax tree and pretty-printing the result back to a textual form. While XT is the set of transformation tools, Stratego is the name of the language for definition of the transformations. In Stratego/XT the grammar is defined in Syntax Definition Formalism format (SDF [29]), a parser takes a text (e.g. in a file) as an input and produces an Abstract Syntax Tree in ATerm format [23], this format is then used both as input and output of the Stratego transformations and the result can be finally pretty-printed to a textual form by a pretty-printer with use of output tables defined in Box format [45].

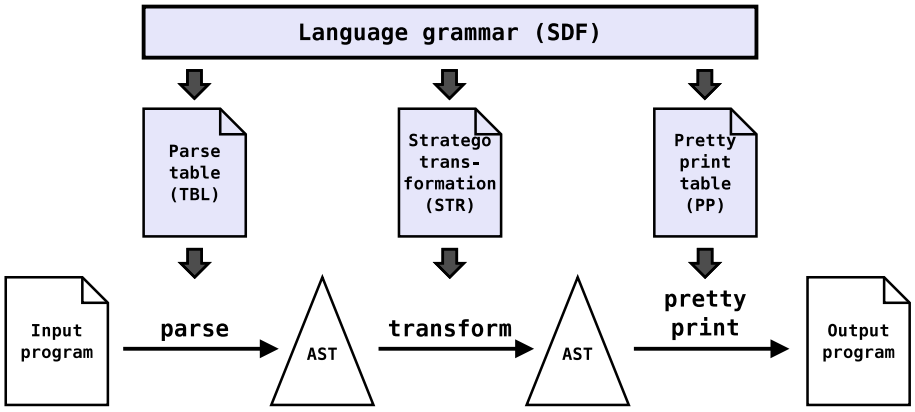
Stratego/XT delivers a collection of tools each dealing with a separate part of the above mentioned process, and there are also other support tools which help the developers use the framework effectively (e.g. grammar table generator, transformation compiler, pretty-print table generator). Each of them is implemented as a separate executable and their inputs and outputs are compatible, allowing for creation of a pipeline of a transformation system. Listing 3.1 presents an example of such a pipeline. Most of the tools also have their implementation in the Stratego language, so they can be easily used as transformations from user programs. Figure 3.1 shows a sketch of a typical Stratego/XT transformation process.

Listing 3.1: Example of pipelining of Stratego programs

```
$ strc -i mytransformation.str -la stratego-lib -m main
$ cat myprogram.txt | sglri -p mygrammar.tbl | ./mytransformation |
    ast2text -p mypptable.pp
```

User defined transformations can be compiled to C or Java code using `strc` and `strj` tools respectively. These programs can then be compiled into corresponding binaries and used in the transformation pipeline. This feature also allows for using of the generated code from external tools (e.g. to call the Stratego transformations directly from Java applications).

Figure 3.1: Pipeline of Stratego software transformation system



3.2 Syntax Definition Formalism

SDF is a declarative language for definition of a syntax of languages, a grammar for any context-free language can be defined in it. This grammar is then used as a basis for generation of a parse table, pretty print table and other definitions used later in the transformations (so that the transformation converts a valid AST to another valid AST). The parse table is then used for parsing of an input text by Scannerless Generalized Left-to-Right parser (SGLR [49]).

3.2.1 SGLR parser

SGLR parser is capable of parsing any context-free language (it is not restricted to any subset fo them like other parsers - e.g. to LALR or LL). Conventional parsers use lexical scanners to combine input letters into lexical tokens, these tokens are then passed to the syntactical parser. A *scannerless* parser on the other hand does not use a separate scanner, instead the lexical analysis is integrated in the context free analysis of the input string, which allows better interplay between these two parts. The *Left-to-Right* part indicates that the parser works in a bottom-up fashion, where it considers the consecutive input lexical tokens (characters) and tries to combine them into higher level notions defined by the grammar. However, this method only works when there is exactly one way to combine the tokens, a *generalized* parser solves this problem by starting new instances of itself whenever it encounters a collision (more possible ways of combining the input tokens). Some of the concurrently running parser instances then might stop in dead-ends, which means that this choice does not lead to a correct parse tree. Other instances might find a possible way to parse the rest of the input tokens[1], so we might end up with a parse tree with several possible choices for the ambigous nodes (and when the ambiguity is in the top level we have a parse forest).

There are several advantages of using such a powerfull parser, firstly we are

[1]The concurrently running parsers are actually merged back to a single one when they are on the same position in the input text and have the same non-terminal produced. This is well described in [49].

allowed to write any context-free grammar (even ambiguous one), secondly we can keep the lexical syntax along with the context-free one, which might be easier for reading and maintaining it, and the fact that a combination of two context-free grammars is another context-free grammar allows us to define modular grammars (bigger grammar based on several smaller ones).

The SDF language itself adds several useful features, including modularity of the files for a grammar and possibility to define disambiguation constructs (priorities of the rules, restrictions for the following tokens and rejections of productions).

3.2.2 SDF language basics

A SDF grammar consists of syntax rules, both lexical and context-free ones. It can be spread to several modules, which can import other modules and use the symbols exported by them.

Each syntax rule consists of symbols, which are similar to terminals and non-terminals in other grammars (e.g. Backus-Naur Form, BNF, used in several popular tools like ANTLR [2] or Yacc [32]). There are three elementary symbols, *sorts* in BNF called non-terminals (e.g. `Expression`), *literals* corresponding to terminals (e.g. `"=="`) and *character classes* used to define several matching characters (e.g. `[0-9]`). We can construct compound symbols from other symbols by applying operators to them: `?`, `+` and `*` are used for optional occurence, repetition and optional repetition respectively. Symbols can be grouped to sequences by parentheses, and choice operator `|` can then be applied to denote alternatives[2].

Syntax rules (also called productions or functions) are then written in a form `S1 S2 ... Sn -> S`, denoting that symbols `S1` to `Sn` can be rewritten (reduced) to symbol `S` (as opposed to derivation rules in BNF, which are written in exactly opposite order and the start symbol is only allowed to be non-terminal). See simple grammar sample on Listing 3.2. The target symbol can be followed by additional attributes enclosed in curly braces, these are used for setting the names of the nodes in the resulting abstract syntax tree (the {**cons**("Sum")} attribute) and for disambiguation, which is mentioned later in this chapter in Section 3.2.5.

Listing 3.2: SDF definition of expression module

```
module Expressions

exports
  sorts Expression

  lexical syntax
    "0"                 -> Number
    "-"? [1-9][0-9]*    -> Number

  context-free syntax
    Number                        -> Expression {cons("Number")}
    Expression "+" Expression     -> Expression {cons("Sum")}
    Expression "-" Expression     -> Expression {cons("Sub")}
    "(" Expression ")"            -> Expression {cons("Par")}
```

[2]There are more available operators described in the SDF documentation [29]

```
lexical restrictions
  Number -/- [0-9]
```

3.2.3 Lexical and context-free syntax

The above described form of production rules is used for the definition of both lexical and context-free syntax. However, the defined symbols are in different namespaces (LEX and CF), every user defined lexical symbol L is converted to namespaced symbol <L-LEX> and every context-free symbol C is converted to <C-CF>. And, for every lexical symbol L SDF automatically generates one rule in a form <L-LEX> -> <L-CF>, so we can easily use the lexical sorts within the context-free rules, and lexical and context-free rules with the same right-hand-side are automatically linked together as seen on Listing 3.3.

Listing 3.3: Lexical and context-free syntax defining the same sort

```
%% Grammar defined by the user
  lexical syntax
    "a"      -> A
  context-free syntax
    "b"      -> A

%% Rules automatically generated from the grammar
    "a"       -> <A-LEX>
    <A-LEX> -> <A-CF>
    "b"       -> <A-CF>
```

In the rules we can use both the symbolic names without namespaces and the fully qualified symbol names. Finally, SDF puts all the defined rules together into a combined syntax while preprocessing all the context-free rules. The preprocessing inserts special LAYOUT? symbol between each two symbols on the left side of the rules, this then serves for automatical parsing of the layout. This symbol is in the hands of the grammar developer, who is supposed to create lexical rules to define it.

SDF uses concept of *start symbols* to define the symbols that can be in the root of the parsed AST when the input string is processed. If there are no start symbols then no input string is accepted. There can be both lexical and context-free start symbols, the latter ones are preprocessed and allow LAYOUT in the beginning and at the end of the input string.

3.2.4 Combined syntax

As mentioned earlier, a combined syntax is generated after the preprocessing phase. Sometimes it is necessary to use context-free symbols, but it is desired to have fine-grained control of the layout (e.g. when a specific construct ends in the end of line which is otherwise allowed in the layout), so the grammar author is also allowed to write directly this combined syntax (where he needs to use the fully-qualified symbol names), as can be seen on the Listing 3.4.

Listing 3.4: Lexical and context-free syntax defining the same sort

```
module Macros
```

```
imports Literals
        Identifiers
exports
  lexical syntax
    [\ \t]+       -> WS
    [\ \t]* [\n] -> LineEnd
  syntax
    <"#define"-LEX> <WS-LEX> <Identifier-CF> <WS-LEX> <Literal-CF> <
        LineEnd-LEX>  -> Macro
```

3.2.5 Grammar disambiguation

When we parse an input string, we try to find the derivation tree that produces exactly that string. In many cases there is more than one derivation tree for the same string, such a situation is called ambiguity. It is quite common that other parsers do some disambiguation steps on their own (e.g. ANTLR applies the rule of the longest match when scanning for lexical tokens). But this is not the situation with SGLR, which offers the developer means to disambiguate the grammar himself according to his needs. There are several disambiguation constructs provided to the user, Listing 3.5 shows their usage in grammar for expressions.

Listing 3.5: Expression grammar with disambiguation constructs

```
1  %% file Literals.sdf
2  module Literals
3  exports
4    sorts Id Number Keyword
5
6    lexical syntax
7      [a-zA-Z] [a-zA-Z0-9]* -> Id
8      "begin" | "end"         -> Keyword
9      Keyword                 -> Id {reject}
10     "0"                     -> Number
11     [1-9][0-9]*             -> Number
12     [\ \t\n]+               -> LAYOUT
13
14   lexical restrictions
15     Id      -/- [a-zA-Z0-9]
16     Number  -/- [0-9]
17     LAYOUT  -/- [\ \t\n]
```

```
18 %% file Expressions.sdf
19 module Expressions
20 imports Literals
21 exports
22   sorts Expr Stm Code
23
24   context-free start-symbols Code
25
26   context-free syntax
27     Id              -> Expr {cons("Var")}
28     Number          -> Expr {cons("Number")}
29     Expr "+" Expr -> Expr {left ,cons("Sum")}
30     Expr "/" Expr -> Expr {left ,cons("Div")}
```

```
31      Expr "^" Expr -> Expr {right, cons("Pow")}
32      "(" Expr ")"  -> Expr {cons("Par")}
33      Id            -> Stm {cons("Decl")}
34      Expr          -> Stm {avoid, cons("Expr")}
35      Id "=" Expr   -> Stm {cons("Assign")}
36      "begin" (Stm ";")+ "end" -> Code {cons("Code")}
37
38    context-free priorities
39      Expr "^" Expr -> Expr >
40      Expr "/" Expr -> Expr >
41      Expr "+" Expr -> Expr
```

The grammar defined on the Listing 3.5 shows basic usage of the SDF disambiguation constructs. These are:

- *Priorities* work as parse forest filters removing certain trees, if A > B priority rule is defined, it means that production A binds stronger than production B, and thus a B node cannot be a direct child of an A node. The priorities are mostly used to define relative priorities between operators, which is shown on lines 38-41.

- *Associativity* attributes define another set of filters for the parse forest. The available associativity attributes {**left**}, {**right**} and {**non-assoc**} suggest that a production cannot be a direct right-most, left-most or any child of itself in that order. Example on lines 29-31 defines addition and division as left-associative operators (we evaluate them from left to right) and power as operator applied from right to left.

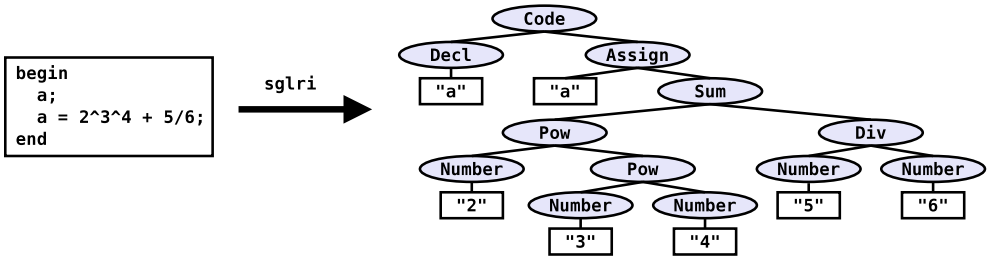
- *Reject* attribute mechanism is used when we want to forbid certain production, which is a subproduction of another one. This mechanism was designed to be used for lexical syntax, for example to forbid rewriting keywords to identifiers as seen on the listing on lines 8-9. Usage of reject attributes in context-free production rules is not fully implemented by SDF, if it was, it would effectively define the difference operator between two context-free languages, which would result in possibility to parse non context-free languages like $A^nB^nC^n$ [30].

- *Preference* attributes serve for the choice of a direct child in case of multiple possibilities, production rules with {**prefer**} attribute get precedence over ones with no attribute, which get precedence over rules with {**avoid**} attribute. This mechanism can be used to prevent some unnecessarily long rewrites as seen on lines 33-34 which both allow to rewrite Id to Stm, but we prefer to go directly than having a production path through Expr.

- *Restrictions* are mainly used to solve ambiguity on the lexical level, where we do not want some symbol to be followed by a set of characters (or even a sequence of characters). It is most commonly used as on lines 14-17 to define the longest match rule, but it can be used in the context-free meaning too, for example to define that an "else" word cannot follow the last statement in an "if-then" statement.

Figure 3.2 shows a sample input string in the grammar defined on Listing 3.5 and its corresponding abstract syntax tree constructed by SGLR parser.

Figure 3.2: Abstract Syntax Tree constructed by SGLR parser



3.2.6 Grammar mixing

As mentioned before, SDF supports separation of a large grammar into several files. This feature can be extended to grammar mixing, we can define separate grammars for two or more languages and then create a new SDF module, which imports all these grammar definitions and define rules for mixing the grammars (e.g. that expression in one language can be used on a place for expression of the other language). When mixing grammars we need to make sure that symbols defined by the separate grammars do not collide, they might both provide Stm sort which would lead to definition of just one sort with all the rules from the two languages put together and will automatically allow us to use statements of one language in place of the other one, which might not be wanted. The imported grammars might not be fully under our control, so SDF allows mechanism of sort renaming as seen on Listing 3.6. Both the **module** and **imports** statements can be parametrized by a variable, the sorts exported by the grammar being imported can then be easily renamed by the OldName => NewName construct.

Listing 3.6: Renaming sorts using parameter

```
module CMixture[Context]
imports languages/C
          [
            Expression => Expression[[Context]]
            Identifier => Identifier[[Context]]
            Statement  => Statement[[Context]]
            Literal    => Literal[[Context]]
            CUnit      => CUnit[[Context]]
          ]
```

```
module TemplateMixture[Context]
imports Template/Main
          [
            Expression => Expression[[Context]]
            Identifier => Identifier[[Context]]
            Statement  => Statement[[Context]]
          ]
```

```
module TemplatedC
imports TemplateMixture[Template]
        CMixture[C]
```

```
exports
 context-free start-symbols CUnit[[C]]

 context-free syntax
  Statement[[Template]] -> Statement[[C]]{cons("TemplateStatement")}
  Expression[[Template]] -> Literal[[C]]{cons("TemplateLiteral")}
```

By definition of separate grammars with renamed symbols we avoid the problems with automatical unification of rules with same sorts from different grammars and we can decide on our own which symbols can be rewritten. For example in the listing we allow usage of a statement defined by the template language in the place where C statement is expected and also use of a template language expression instead of C literal. The method for integration of two languages (usually a Domain Specific Language into a bigger host language) is called MetaBorg method and is described in [35] and [34].

3.3 Stratego programming language

Stratego is a programming language for transformation of abstract syntax trees in ATerm format. AST in ATerm format is produced from SGLR parser according to grammar in SDF form. Stratego is based on the concept of term rewriting with programmable rewrite rules and strategies. There are two general types of operations: *strategies* are used for locating the correct subtree within the AST and *rewrite rules* are applied to these subtrees to transform them (or their parts).

3.3.1 Terms and term patterns

In Stratego we refer to a subtree of AST as a *term*, the name of a term comes from the {**cons**("Name")} attribute of the syntax rule which produced that node and the parameters of the term are the subtrees for the left part of the syntax rule. For example when considering the grammar on Listing 3.5 we can have simple terms like Number("4") or complex terms like Sum(Id("a"), Number("2")). A term does not have to have all the parameters fully specified, there can be meta-variables instead of some of them, in that case we call it a *term pattern*, e.g. Number(x).

To not spoil the structure of the AST while transforming it, each Stratego program needs to know the possible structure of the terms within the AST, to ensure that only rewrite rules converting a valid term to another valid term are allowed. Program section called signatures is used to define the available *sorts* and the *constructors* of the terms. Names of the terms in the AST correspond to the {**cons**("Name")} attributes of the SDF grammar rules and the sorts in the constructor rules define the types of the subtrees of the nodes. These signatures can be automatically generated from the SDF definition of the grammar and then imported into the program file.

Stratego has a concept of *current term*. All the strategies and rules are applied to the current term, and there is no possibility to escape out of the current subtree (to get to a parent of the current node in the AST), strategies can only descend into subtrees of the current tree.

3.3.2 Strategies

Strategies are used for locating the correct position for application of rewrite rules. They are defined in a program section started by **strategies** keyword. Each strategy has the following form:

```
name = transformations
```

where name is the name of the strategy and transformations is a combination of transformations. The whole program tranformation starts in *main* strategy, its name is specified at the time of STRATEGO program compilation.

The simplest transformation is an application of a strategy or rule, which is performed by writing its name. There are also several predefined strategies, which can be combined to produce more complex transformations. The basic ones are the identity strategy id which succeeds without changing the current term, failure strategy fail which fails and also does not change the current term, term creation strategy !term which replaces the current term with the term, and term matching strategy ?term−pattern which potentially binds all the free meta-variables in the pattern and only succeeds if the current term has the same name and number of parameters as the term−pattern, and all the parameter terms also match. There is no possibility to rebind a variable to a different value within the same scope (by default the currently executing strategy), but smaller scopes can be created by the following construct: { localVar1, ..., localVarN: scoped−strategies }.

All these basic strategies can be combined into more complex ones by three operators

- s1 ; s2 - *sequential composition* operator first performs strategy s1 and if that succeeds it performs s2. It fails if any of the two strategies fail.
- s1 <+ s2 - *deterministic choice* operator tries to apply strategy s1, strategy s2 is only applied if s1 fails. The whole operation fails only if both strategies fail.
- s1 < s2 + s3 - *guarded choice* operator works like if (s1) then s2 else s3 statement in standard programming languages. It performs strategy s1, if it succeeds the program flow continues with s2, otherwise s3 is applied. This combined strategy succeeds if both s1 and s2 succeed or s1 fails and s3 succeeds.

Apart from these strategy operators, STRATEGO also allows to define a *block* of strategies by enclosing them into parentheses, e.g. (s1; s2; s3), this block can then be used in a position where a single strategy is expected.

Strategies can also be parametrized by other strategies and terms:

```
strategy(s1, ..., sm | t1, ..., tn) = transformations
```

where s1, ..., sm are strategies and t1, ..., tn are terms. The parametrized strategy can then use its parameters in the transformation it defines:

```
not(s) = s < fail + id
```

The defined not strategy succeeds when the strategy passed to it as a parameter fails and fails otherwise.

3.3.3 Rewrite rules

Rules define a transformation of one term pattern to a term (it can be a pattern but needs to have all the variables bound). The rewrite rules are defined in **rules** section of a STRATEGO program and have the following form:

```
RuleName: source -> target
```

where RuleName is the name of the rule, and source and target are term patterns. It tries to match the current term to source and replace it with target term. The rule is not applied if source does not match the current term.

It might seem that rules are only a syntactic sugar for the following strategy (?source ; !target), however, rules have another useful feature - there can be multiple rules with the same name and the first one (in the order of occurence in the source files) which matches the current term is applied.

Rewrite rules can also be made *conditional* by the following construct:

```
RuleName: source -> target where condition
```

The rule is only applied when condition succeeds. The code in condition contains transformations as defined before. This condition can be used to perform some additional computation on the matched meta-variables and bind the variables used in the target term.

Lambda rules can be created anywhere in the code where a rule name is expected, their definition lacks the rule name and is enclosed in backslash characters \ x -> y \.

STRATEGO also offers the possibility to create rules at runtime while transforming the AST, which allows for a dynamic behavior of the program. A *dynamic rule* can be created in the following way:

```
rules(DynamicName: source -> target where condition)
```

It can then be used as all the standard, statically defined, rules in the whole program until it is undefined by the call to **rules**(DynamicName:-). Dynamic rules are used for transferring the context of different parts of the abstract syntax tree to the transformation of another part of the tree, the most common uses are constant folding or function code inlining.

The programmer is allowed to create local scopes for dynamic rules by the following construct:

```
{| LocalName1, ..., LocalNameN: scoped-strategies |}
```

where LocalName1 through LocalNameN are names of dynamic rules only defined for the scoped-strategies.

3.3.4 Term traversal

Strategies and rewrite rules allow to descend through the AST based on the knowledge of its structure and to apply the rules where wanted. Sometimes we know, that we want to apply a rule to every term with constructor XYZ, but these nodes might be deep in the AST. In such cases it is useful to define traversal strategies that will be able to go through any AST. For this purpose STRATEGO offers

several basic traversal operators which can be combined to form more complex traversal strategies.

The *generic one-step descent operators* are:

- *all(s)* applies the strategy `s` to all direct subterms of the current term and succeeds only when the application to all the subterms succeeded.
- *some(s)* tries to apply the strategy `s` to all direct subterms and succeeds when the strategy succeded on at least one of them.
- *one(s)* tries to apply the strategy `s` to the direct subterms from left to right and succeeds after the first successful application, fails if the strategy failed on all direct subterms.

These generic operators are then used for definition of *standard traversal strategies* and other traversal strategies can be defined by the programmer. Listing 3.7 shows the definitions of the most common traversal strategies, we are using the fact that we can define the strategies recursively.

Listing 3.7: Standard traversal strategies

```
%% visiting all subterms
  bottomup(s) = all(bottomup(s)); s
  topdown(s) = s; all(topdown(s))
  innermost(s) = bottomup(try(s; innermost(s)))

%% visiting some subterms
  sometd(s) = s <+ some(sometd(s))
  somebu(s) = some(sometd(s)) <+ s

%% partial traversals
  oncetd(s) = s <+ one(oncetd(s))
  oncebu(s) = one(oncetd(s)) <+ s
  alltd(s) = s <+ all(alltd(s))

%% and many more ...
```

There is a special kind of term which is called a *list* (it is just an array of values). Lists are created by the repetition occurrence symbols (A* and A+) in SDF. It is possible to apply a strategy to all the items of the list by the `map` rule, for example:

```
  map(\ x -> <mul> (x, 2) \)
```

multiplies each item of the list by two. On this example we can see, that it is possible to use a strategy on a different term than the current term by enclosing the strategy name into angle brackets and passing the term after that:

```
  <strategy> term
```

3.4 Pretty printing

After completing transformation of the AST by a Stratego program we usually want to convert the tree back to the textual form of the language. This requires

a process opposite to parsing which is called *pretty printing* (even if the result is ugly). For this process we need to have another set of rules, which can reintroduce layout and print the fixed strings not needed in the AST. Stratego/XT contains tools for automatical creation of pretty print tables and it also offers a tool which takes this pretty print table and transforms an AST to a text according to it. The pretty print table has to contain information how to print all the different terms that can occur in the AST. The terms are identified by their constructors, so the pretty print table is responsible for defining a way to print every rule with a constructor back to text. Box Text Formatting Language is used for the description of these tables.

3.4.1 Box text format

Box language is used to describe the intended layout of the text. It formats input data into boxes according to the specified operators. Listing 3.8 shows a pretty-print table for a part of a grammar. The H[data] operator aligns given data parameters into horizontal boxes and allows us to specify the horizontal space size, whereas the V[data] operator aligns the data parameters below each other and we can specify both the indentation size and the vertical distance between the boxes.

Listing 3.8: Renaming sorts using parameter

```
%% SimpleLang.sdf
module SimpleLang
imports Layout
exports
  sorts Stat Expr Id Number
  context-free start-symbols Stat
  lexical syntax
    [a-z] -> Id
    [0-9] -> Number
  context-free syntax
    "if" Expr "then" Stat ("else" Stat)? "fi" -> Stat {cons("If")}
    "{" {Stat ";"}+ "}" -> Stat {cons("Block")}
    Id "=" Expr   -> Stat {cons("Assign")}
    Number        -> Expr {cons("Num")}
    Id            -> Expr {cons("Id")}
```

```
%% SimpleLang.pp
%% the pretty-print table for the code above
[
   If                -- V[V is=2[H hs=1[KW["if"] _1
       KW["then"]] _2] _3 KW["fi"]],
   If.3:opt          -- _1,
   If.3:opt.1:seq    -- V is=2[KW["else"] _1],
   Block             -- V[V is=2[KW["{"] _1] KW["}"]],
   Block.1:iter-sep  -- H hs=0[_1 KW[";"]],
   Assign            -- H hs=1[_1 KW["="] _2],
   Num               -- _1,
   Id                -- _1
]
```

Using this grammar and pretty-print table we can get ugly text formatted just by parsing it by `sglri` tool and printing it back by `ast2text` tool. This input source code:

```
if 1 then {a=1; b=2 } else a=0 fi
```

results into the following pretty-printed code:

```
if 1 then
  {
    a = 1;
    b = 2
  }
else
  a = 0
fi
```

More available operators, like the one for tabular formatting or horizontal-vertical formatting taking into account the line width, are described in [45].

3.5 Using XT from Stratego

STRATEGO/XT has most of its core tools implemented in its libraries, which can be imported to user programs. It also offers the strategies in the STRATEGO language, which call these library functions, so it is possible to write STRATEGO programs which can perform all the necessary steps starting from parsing, then transforming the parsed AST and finally pretty-printing the resulting AST back to a file, without the need to call any external tool or program.

Parsing transformations are available after importing library libstratego-sglr. It includes strategies like open-parse-table which opens the parse table referenced by the current, term and strategies parse-file-pt and parse-string-pt with variable number of parameters (ex. desired start symbol, path and failure strategies), for parsing text from a file or an input string into a parse tree. This tree then needs to be converted to ATerm format by implode-asfix. Listing 3.9 presents an example of parsing and pretty-printing a text in the SimpleLang grammar (as defined before) from STRATEGO.

The libstratego-gpp library offers access to several pretty-printing strategies. The parse-pp-table-file strategy parses a pretty-print table in the given file, ast2abox converts the current AST to box format according to the passed pretty-print table and box2text-string prints a box expression into a string with given line-width.

Listing 3.9: Parsing and pretty-printing from Stratego

```
module ParsePrintSimple
imports
  SimpleLang      %% contains the signatures
  libstratego-lib
  libstratego-gpp
  libstratego-sglr
  libstratego-xtc

strategies
  main = parse ; pretty-print
```

```
parse =
    import-term(SimpleLang.tbl)
  ; open-parse-table => parse-table
  ; !"input.txt"
  ; parse-file-pt(<echo> "Cannot open", <echo> "Parse error"
      | parse-table)
  ; implode-asfix

pretty-print =
    where (<parse-pptable-file> "SimpleLang.pp" => pp-table)
  ; ast2abox(|[pp-table])
  ; box2text-string(|80)
  ; print-to    %% print the string to a temporary file
  ; rename-to(!"output.txt")
```

Chapter 4

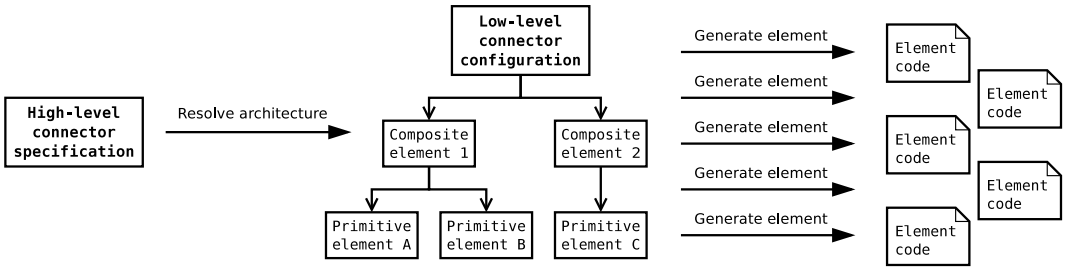
Analysis

As stated in the introduction, this thesis aims to create a C code generator and incorporate it into the current SOFA 2 connector generator, so that connectors for C language can be generated. This chapter contains an overview of the connector generator, and an analysis of steps that need to be performed to achieve the goal of the thesis.

4.1 Connector generation overview

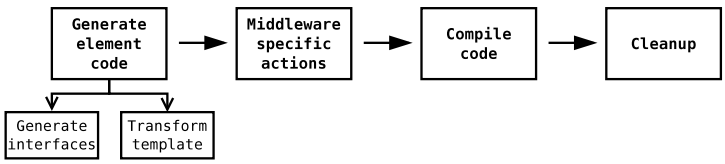
Figure 4.1 shows a general overview of the connector generator as proposed by [37]. First, the architecture of the connector (consisting of elements) is resolved. Then each element is generated.

Figure 4.1: General connector generator workflow



Each connector element has an action script associated with it, which describes the steps necessary for the element generation. Figure 4.2 depicts the usual process of element generation.

Figure 4.2: Typical element generator workflow



The element code generator is responsible for generation of all the code of the element. The first step involves preparation of the communication (business) interfaces, which may include adaptation for the selected middleware (e.g. adding parameter with middleware context to all of the methods). Afterwards the code of the element is generated by transformation of a template. The work [47] identified STRATEGO/XT as the best tool for these transformations. The script may also contain actions for running middleware specific tasks (e.g. an IDL compiler). Finally, the generated code is compiled to be usable in the application, and unnecessary unnecessary files are removed.

4.2 Proposed changes for C support

We want to create C connector generator in a way that would allow it to easily coexist with the current Java generator (so that the interface for using these two is the same). The connector generator is independent of the target language, all the target language specific tasks are concentrated in the element generators. We will retain the proposed action-oriented element generator architecture, and in the next sections we will analyse which parts of the element generator need to be modified for C code generation.

4.2.1 What cannot be reused

In general, the element generator consists of actions displayed on Figure 4.2.

The first thing that needs to be done is *generation of interfaces*. The input of this action is a symbolic type specifier describing the interface (where is the interface stored, and how should the generator modify it), the specifier is resolved by a type factory specifically created for Java (`JavaTypeFactory`). This factory uses Java introspection for getting information about the Java interface (by `Class.forName()`), all the modifications are performed on this object, and the interface code is then directly written to desired interface file. In C, we encounter several problems. First of all, there is no language structure designed for representation of an interface, this needs to be created. Secondly, we cannot use Java introspection for C code, thus we need to find a way to parse an interface code, modify the parsed information and print it back to a file. So we need to create a type factory specific for C which will develop its own approach to interface parsing, modification and printing, and it will understand the specific needs of C (e.g. the need of protection of type declarations in header files by **`#ifdef`** macros, so that the types declarations do not conflict).

We also cannot use the current *template transformation* module written in STRATEGO, because this module is directly tied to Java, so it can only parse and transform templates containing the mixture of Java and ElLang code. There are two possible ways of solving the actual code generation for C. Either we could create a completely new way of code generation (using either STRATEGO or another transformation language like XTEXT [31]), or we can base the C transformation module on the current one and rewrite only the target language specific transformations. The second approach has been chosen, since the main parts of the element generator introduced by [47] are independent of the target language, thus these can be used for C code generation too. This element generator is built on

top of STRATEGO/XT framework, which is the leading language transformation tool, so we will be extending it also using this framework.

To support C templates, we will need to design a C language grammar (this grammar is used both for parsing of the template and ensuring that the STRATEGO transformations produce valid abstract syntax trees), identify the target language specific transformations in the current element code generator and rewrite them for C, and create several sample C templates. For this task, the thesis will analyse the possible ways of interaction of connector element code with the application code and use the most appropriate one in the element templates.

Another distinguishing factor between C and Java are separate header and code files in C. To be able to export the functionality implemented in the code file, we will need to be able to generate header files. This could be done automatically from the code file by moving all the non-static declarations into a header file. This approach would require creation of quite complex transformations and might put unwanted declarations in the header files. Another possible approach would be to also support templates for header files, which allows for greater scalability (the template designer puts exactly what he wants into the header file) and is also easier to implement (since headers are written in the same language as standard code files). Taking the above mentioned factors into consideration it was decided that the second approach will be taken.

The STRATEGO template transformation needs to be called from the Java action (the element generator action). For this task we will follow the approach suggested by [48], and compile the transformation by `strj` tool into a JAR archive, then directly call the transformation from the element generator action.

The *middleware specific actions*, like calling `rpcgen`, shall be created separately for each middleware. These are usually tied to the middleware, thus contain very little things in common, so it does not make sense to create a framework for them.

The current actions for Java *code compilation* just find out the path to the generated file, construct a classpath and use javax.tools.JavaCompiler class to compile the generated sources. We cannot use this approach for C code, because this class can only compile Java code. This means that we need to determine a different way of compiling the generated code (and also of creating correct paths to directories to include - for example when compiling a composite element, its template may include the headers generated by its subelements). For this task, we will present one new action for generation of a Makefile for the element and another one for running the `make` program in the correct directory.

For the *cleanup*, the Java element generator only removed a single source file (since it only generated a single file), for C we may generate more than one source file (e.g. header and source). This information is available at the time of generation of the makefile, thus we can enhance the makefile by a target for deletion of the sources and intermediate object files.

4.2.2 Coexistence of Java and C element generators

The created C element generator can be inserted into the current infrastructure using various approaches, the most suitable and easy-to-use one should be chosen. There are two general approaches that can be taken: either new connector

generator supporting only C can be cloned from the existing one (org.objectweb.dsrg.congen.Generator), or this generator could be enhanced to support both Java and C. The second approach minimizes the amount of duplicated code and also allows to perform tasks that two coexisting generators would not allow, for example creation of adapter connectors from Java to C or the other way around (e.g. by using common middleware like CORBA).

To support connectors for both languages we need to allow the user to choose which connectors he wants to generate. The high-level specification is used for the definition of the input, so it is the right place for the choice of language (which is also an input parameter). For this selection, new NFP requirement language will be introduced. Its sample use in the high-level specification is shown on Listing 4.1. Specification of this NFP property on a unit granularity allows the possibility of already mentioned adapter connectors, where one unit could be possibly generated for C and the other one for Java.

Listing 4.1: Choice of language in the high-level specification

```
<specification>
  <unit name="client_unit" dock="dockA">
    <nfp-requirement predicate="nfp_mapping(Unit,'language','c')"/>
    <nfp-requirement predicate="nfp_mapping(Unit,
        'communication_style', 'method_invocation')"/>
    <port name="call" type="provided"
        signature="c_interface('test_myiface')"/>
  </unit>
  ...
</specification>
```

To support this feature, the configurations of already existing top-level unit elements will be modified to reflect this NFP. To ensure that the child elements of the top-level units will be created for the same language, separate element types will be defined for all languages (e.g. c_stub for a C stub and stub for Java stub).

4.2.3 Using the proposed element generator

This section covers the changes that have been brought to the users (both end-users and element developers) of the proposed C element generator in comparison to the Java element generator.

Connector generator user

The proposed element generator for C templates is usable in the same way as the original Java element code generator, these two can even coexist in the same connector generator. It is up to the user of the connector generator to pick, which generator should be used. This choice is made in the high-level specification of the connector by NFP requirement called language, where value 'c' denotes C code and 'java' denotes Java code. The architecture resolver then creates the low-level configuration (as shown on Figure 4.1) which either contains Java elements or C elements depending on the choice of the mentioned property. These elements are then generated by performing all the actions in their action scripts in their architecture definition files.

The output of the C connector generator is a set of object files (for each connector element there is one object file created) and corresponding header files with exported function definitions. Composite elements automatically create object archives composed of object files generated by its subelements. The usage of these object files or object archives is up to the user of the connector (which is the same as in case of Java connector generator, which creates compiled Java files).

Element designer

The major task in designing a new connector is the creation of the connector units architectures (along with the action scripts) and the definition of the corresponding template files. In case of C elements, the user is allowed to create two templates for each element (one for the code and the other one for the header file), which are then processed within one generator action, paths to both templates need to be specified in the action script of the element (so that the generator knows that two templates should be processed for the element).

The element designer can extend the current C type factory or create his own type factory if the middleware needs more complex modifications of the business interfaces. For middlewares requiring other files than just the interface to be prepared (or if the interface needs to have specific format), special element generator actions shall be created and inserted into the element action script.

4.2.4 Proposed element generator workflow

The main element generator task, the generation of code, will be implemented in a new action, StrategoCGenerator, which will first either generate the subelements (in case of composite element) or prepare the code of business interfaces with the use of new C type manager. The business interfaces can be modified by use of several predefined simple actions implemented in the C type factory, or more complex actions can be created in new type factories specifically tied to some middleware.

After these prerequisities are fulfilled, the element descriptor is prepared and a Stratego transformation specifically created for C code templates is started (the element descriptor is passed to it). The transformation proceeds as previously described in Section 2.2.4 and depicted on Figure 2.4. It first preprocesses the input element description, then it loads the main code template (and possibly the templates that this code template extends) and evaluates them (e.g. evaluates all the ElLang statements along with expanding template method bodies for the business interfaces) while creating C statements in places where Java statements were created in the previous template transformer. After the main template is processed, the Stratego transformation performs the same evaluation steps for the header template if the input element descriptor contains path to it.

The Makefile for the element is created after that, and `make` program is started in the correct directory to compile the generated files.

4.3 Goals revisited

4.3.1 Create C grammar for Stratego

The previous chapter explained that we need a full SDF grammar definition for the target language which we are generating. This grammar is used both for parsing of the templates and for the transformations, so that a valid target language code is generated. The java-front project [27] delivers Java grammar written in SDF, however, there is no project for C language of such a high quality. Several C grammars exist but they are usually too ambiguous, or they support only a limited subset of the language. For this reason a new C grammar will be created, based on one of the existing grammars.

There is also a need to support C macros in the grammar so that we can generate code with macros. A grammar for the C macros should also be created and combined with the C grammar, this mixed grammar will then be used as the target language grammar.

4.3.2 Enhancements of the code generator

The current Stratego code generator only supports Java. New version of the code generator should be created and adopted to support C as the target language. All the target language specific transformations should be identified and rewritten to successfully work with C templates.

A way of representing an interface in C language shall to be chosen. Convenient method for interaction of the generated code with external code (e.g. a component) should be found out, and proper transformations should be created to support this interoperability (e.g. declaration of the generated methods in a header file).

4.3.3 Modifications of the element generator

Support for calling of the C code generator should be added to the current SOFA 2 connector generator. New action for the action script which will invoke the C code generator should be created. This generator action will need to prepare input file for the Stratego code generator with information specific for C, e.g. it should name both the header and the code files that need to be generated.

Type system for the C language should be created in order to support scripted interface modifications and C version of language tools should be created so that the rest of the generator can easily find out the path to the generated interface header files.

Suitable way of automatical compilation of the generated C code should be developed and it should be incorporated into the connector generator.

4.3.4 Creation of sample C templates

Every template is directly bound to some middleware, so the template mainly deals with interaction with the chosen middleware technology. Unfortunately, there are not as many possibilities of opensource middleware technologies for C as there are for Java. The biggest difference between these two languages from

the middleware point of view is that no equivalent of the Java *serialize()* method exists in C. So every possible C middleware needs to perform the data encoding and decoding itself, and it needs to know the types of parameters of the methods in advance in order to generate the stubs and skeletons, which can then be used from the code template. Another problem is that there is no standard way of defining an interface in C, so each middleware technology uses its own description of the communication for the generation of stubs/skeletons. There are generally three possibilities for choice of middleware:

- Find a middleware which would take our description interfaces as input IDL and will generate easily usable stubs/skeletons.
- Choose a middleware taking a different form of IDL as input. This approach will require additional effort in form of creation of interface description to the input IDL transformation, which will need to take place before the stub/skeleton generation. The generated stubs will then need to be adapted to be easily called from client element code, and the skeletons' functions would need to be redefined to call the server elements' functions.
- Create a simple middleware from scrach to fulfill the requirements, e.g. taking our interface description as an input and generating an easy-to-use code. This approach requires the most effort, but allows us to adjust it to the special needs we might have in the future (e.g. so that it is usable in realtime environment).

The choices made will be discussed in Chapter 8.

4.3.5 Creation of sample application

Finally, a sample distributed application using the connectors generated by the new connector generator will be created to demonstrate the usage of the generated code from outside and to show the feasibility of the chosen interfaces of the generated elements.

Chapter 5

C language grammar

5.1 Differences between C and Java

It is important to realize that there are significant differences between Java and C when considering code generation, interface definition and availability of tools for these languages (middleware technologies specifically). These differences are shortly discussed at the beginning of the chapters affected by them.

The first problem is, that C language is ambiguous, which is not the case with Java. First of all, C supports pointers which makes the parsing harder (e.g. the same operator is used for pointer declaration, dereference and multiplication). Then in C we are coping with two grammars mixed together - there are macros, which might make the file look syntactically incorrect (e.g. macros containing semi-colon statement).

For Java the java-front project [27] delivers a well tested SDF grammar with many tools around it (tool implementing the parser with disambiguation, pretty-printer, many Stratego transformations for Java and concrete syntax definition, which makes the creation of transformations easier). There is no project of such extent for C, the available grammars usually support only some subset of C and never support macros inlined in the C code, which is something that we want to be able to use (see Section 5.3 for details).

5.1.1 Complex notation for declarations

C language contains much more complex type system than Java, we have pointers here and the way that variables, pointers and functions are declared requires higher complexity grammar to parse it. In Java, every declaration has the following form (potentially followed by an initializer):

```
TypeSpec varName;
```

whereas in C the type specification and variable name are somewhat mixed and declarations look like this:

```
TypeSpecifier Declarator;
```

where declarator can contain additional important type information, like pointer declaration or array declaration. This brings problems when we want to compare types, decide what to do based on the type and during other actions.

5.2 C syntax

5.2.1 Base for the grammar

The C grammar is based on a grammar provided within the SDF Library distributed in sdf2-bundle project [28], this grammar in comparison to other available SDF grammars (e.g. CTools [24]) uses at least some disambiguation constructs (e.g. preferences for expressions) and is partially based on the ANSI-C standard. However, it accepts more type-incorrect C code than the standard (especially in case of type definitions and expressions) and it lacks support for several important C constructs like function pointer declarations. The biggest problem of this grammar is that it is not used in any tool, thus not well tested, which leads to several bugs that need to be coped with.

5.2.2 Enhancing the grammar

The selected base grammar was enhanced to suit the needs of the connector generator, so that the upgraded grammar can describe any construct that is needed in templates and it is easily mixable with our scripting DSL, ElLang.

The first thing that had to be done was the introduction of constructor attributes to all of the production rules since the base grammar did not have them defined. Then declarations were redefined to be more restrictive. The original grammar had them defined in a way shown on Listing 5.1, which allowed meaningless constructs like **struct** a **typedef int** x;, the specifiers were separated into smaller groups as depicted on Listing 5.2. This new approach allows us to avoid parsing of non-declarations as declarations and also distinguishes declaration of new types, function declarations and function pointer declarations from standard variable declarations. This distinction is then used in template transformations.

Listing 5.1: Old loose grammar for declarations

```
Specifier+ {InitDeclarator ","}+ ";" -> Declaration
Specifier+ ";"                       -> Declaration {avoid}
```

Listing 5.2: New restrictive grammar for declarations

```
StorageSpecifiers QualifiedTypeName {InitDeclarator ","}+ ->
    VarDeclaration {cons("VarDeclaration")}

StorageSpecifiers FullTypeName Identifier "(" Parameters ")" ->
    FuncDeclaration {cons("FuncDeclaration")}
StructKW Identifier "{" StructDeclaration+ "}" ->
    NewTypeName {cons("StructDeclaration")}
NewTypeName      -> TypeDeclaration {cons("TypeDeclaration")}

TypeDeclaration  -> Declaration
VarDeclaration   -> Declaration
FuncDeclaration  -> Declaration
```

Rules for definition of pointers were also restricted from the generic rule:

```
("*" Specifier*)+ -> Pointer
```

to more restrictive ones:

```
"*" TypeQualifiers              -> PointerSingle {cons("Ptr")}
TypeQualifiers PointerSingle+ -> Pointer {cons("PointerSpec")}
```

5.3 C macros support

There are several ways of coping with C macros, first one is complete elimination of macros by working with templates already processed by C preprocessor, but this would lead to huge code expansion (include directives are also macros) and a need to know the macro values before template processing (to perform the preprocessing). Another possibility is to create completely separated grammar for the macros (with definition of all the symbols from scratch), this would allow the best control over the grammars but would lead to unnecessary duplication of some grammar rules (e.g. for expressions) and would not be easily mixable with the DSL grammar later. The last way is to create macro grammar based on some subparts of the C grammar (expressions, identifiers and constants), which solves all the problems of previously mentioned methods. This approach was selected for the implemtation. Thus a special grammar for C macros is created and it is mixed with the C grammar, this combined grammar then supports parsing of C code with macro definitions and applications.

5.3.1 Macro limitations

We do not support all the possible C macros in all the places they are allowed to be by the standard. Mostly we do not allow macro usage in a way the code would not look syntactically correct as in the Listing 5.3. Each macro application needs to look like a valid function call or variable name and the code around needs to be parsable as a statement, which needs to end with the ; character.

Listing 5.3: Invalid use of macro in C code

```
#define DPR(text) printf("%s: %s\n", __FUNCTION__, (text));

int compute(int x, int y)
{
  int res;
  DPR("starting computation")
  res = x + y;
  DPR("finished computation")
  return (res);
}
```

5.3.2 Using combined syntax for macros

In the grammar for macros we need to be able to parse the end-of-line as a symbol denoting an end of a macro definition, otherwise we could not tell the difference between an empty macro definition followed by an expression-like statement on the next line, and macro definition equal to the value of the expression (Listing 5.4 shows this ambiguity). The problem here is, that in the rest of the grammar we

want to allow the end-of-line character as a layout symbol, thus we need fine grained control of the layout while defining the macro grammar, which can be achieved by using combined syntax as introduced in Section 3.2.4.

Listing 5.4: Empty macro definition and a variable declaration?

```
#define FORGET(a, b)
a *b;
```

Example productions for definition of macros is shown on Listing 5.5.

Listing 5.5: Production rules for define macros

```
lexical syntax
  [\ \t]+          -> WS
  [\ \t]*  [\n]    -> LineEnd
syntax
  <"#define"-LEX> <WS-LEX> <Identifier-CF> <WS-LEX> <Expression-CF>
      <LineEnd-LEX> -> <StatementMacro-CF> {cons("CMDefine")}
  <"#define"-LEX> <WS-LEX> <Identifier-CF> <LineEnd-LEX> ->
      <StatementMacro-CF> {cons("CMDefineId")}
  <"#undef"-LEX> <WS-LEX> <Identifier-CF> <LineEnd-LEX>  ->
      <StatementMacro-CF> {cons("CMUndef")}
```

This method on one hand allows expansion of C expression within the macros, however, it also allows parsing of syntactically incorrect code (imagine a new-line inside the expression), but this is not a big problem for us - we generally want to allow parsing of more rather than less.

5.4 Solving grammar ambiguity

C code is ambiguous by design, most of the ambiguities are caused by the lack of full type information while parsing the input. Let us consider the input file on Listing 5.6.

Listing 5.6: Ambiguous C grammar - expression or variable declaration?

```
#include "my_header.h"

int main(int c, char **argv)
{
  a *b;
  return (0);
}
```

From that code we cannot tell whether the `a` is a typedef or a type macro defined in the included header and the line should be parsed as a variable declaration or perhaps both `a` and `b` are expressions (either variables or some macros) and the line is only a multiplication expression and we do not care about the result.

This problem could be solved by having all the system types predefined in the grammar and either doing the work of a preprocessor and including all header files or parsing C-preprocessed code (which we do not want to do as explained above). This process of creating the full type information has several problems. We would need to know in advance from where should we include the headers

(both the system paths and user include paths) and the values of externally defined macros, and processing of such files (with the **#include** directives expanded) would take considerably more time than processing the files without including the headers. And because we cannot create dynamic rules while parsing, we would need to do the processing of the type information in the transformation part of code processing.

Because of these problems it is not desired to include the headers and we want to retain the macros also in the generated code so the preprocessor approach cannot be taken.

5.4.1 Grammar ambiguity approach

We need to be able to parse any valid C code and not to end up with parse error because of a known ambiguity, so we want to have the grammar ambiguous.

We also allow mixing of declarations and statements within code blocks (as opposed to the standard which only allows declarations on the beginning of blocks), this leads to smaller ambiguous parts in the resulting tree, which is better in general and allows for easier integration with the DSL later.

5.4.2 Disambiguation in a transformation

After the parsing phase we may end up with an ambiguous parse tree (or a forest). If there are any ambiguities that need to be solved in a special way (where we prefer some possible trees over the others), then these should be solved in a special transformation, disamb−known. Currently, there is no known problem with that so it is just an identity transformation.

There are ambiguities which may remain in the parse tree even after the initial processing. It would not make sense to perform all the following transformations on the ambiguous subtrees in parallel (we need to end up with fully unambiguous tree before pretty printing it, so we would be making transformations on subtrees that we would finally throw away). Thus a unifying transformation picktree is created and applied after each parsing. We have a fully unambiguous parse tree after this transformation is applied on it.

Listing 5.7: Disambiguation in a transformation

```
signature
  constructors
    amb: List(Statement_CCX) -> Statement_CCX

rules
  UnifyAmbList: amb(List) -> amb(<nub> List)
  EliminateSingleAmbList: amb([Node]) -> Node
  PickFirstTree: amb([Head|Rest]) -> Head

strategies
  disambiguate-c = log-debug(|"Starting disambiguation");
    disamb-known; log-debug(|"Disambiguated known ambigueties");
    unify-lists; log-debug(|"Unified same ambiguous subtrees");
    pick-tree; log-debug(|"Picked first nodes in all amb(list)
        subtrees")
```

```
/* Try to get rid of known ambiguities - currently no */
disamb-known = id

/* Unify items of amb node with same contents */
unify-lists = bottomup(try(UnifyAmbList; EliminateSingleAmbList))

/* Deterministically pick the first node in all the amb(list)
 * nodes in the whole tree, thus fully eliminate the ambiguity. */
pick-tree = topdown(try(PickFirstTree))
```

5.5 Mixing C grammar with ElLang

ElLang is a DSL designed for the code generation, the templates contain a mixture of ElLang and target language. To be able to parse these templates, a mixed grammar for the two languages needs to be defined. We will proceed the same way as the original paper [47] that proposed ElLang did it, it means in a so-called MetaBorg method [35].

To achieve this we define a new module where we rename all the sorts exported by C grammar with macros, so that a context (CCX) part is incorporated into their names (by rules in a form Expression => Expression[[Ctx0]]). This way we avoid unwanted unification of equally named sorts from C and ElLang. Then we create a new grammar module for mixture of the two languages, called *ElLang-C*, we import both renamed grammars (with sorts with contexts) and define production rules for the allowed rewrites of symbols from one language to the other one (Listing 5.8 shows an example of that approach).

Listing 5.8: Part of the mixture of ElLang and C grammars

```
Stm[[ECX]]          -> Statement[[CCX]]  {avoid, cons("FromTL")}
Statement[[CCX]]    -> Stm[[ECX]]        {cons("ToTL")}
Expression[[CCX]]   -> TargetLanguageExpr[[ECX]]  {prefer}
```

To enable parsing and thus expansion of ElLang variables inside C string literals (e.g. for logging purposes to allow printing of the template method name) we need to define a way to compose C string literals from smaller parts. We want the string literals to capture all the layout, but we also need to use context-free symbols to define the parts allowed within a string (ElLang variable reference is defined this way). Therefore we define the string parts and the whole string literal in the combined syntax (we did the same with the C macros) as depicted on Listing 5.9.

Listing 5.9: C string literals composed from smaller parts

```
EStringLiteral          -> <StringConstant[[CCX]]-CF>  {prefer}
"\"" EStringPart* "\""  -> EStringLiteral  {cons("StringConst")}
EStringChars            -> EStringPart     {cons("StringChars")}
<VarRefL-CF>            -> EStringPart     {cons("IdFromTL")}
```

5.6 Pretty printing

Pretty print table for printing an AST parsed by the C grammar with added support of macros is also created. It does not include definitions for printing of any of the ElLang terms, because these are not supposed to be in the AST at the time of printing. If these were included it would mean that the transformation phase failed and left some of the DSL parts untransformed. The pretty-print table uses the Box format to produce easily readable code.

Besides the pretty-print table two Stratego transformations were created, pp−c−string uses the table to convert the current term into string with C code, and a wrapper transformation disamb−and−pp−c(|fileName) that disambiguates the current term, pretty prints it into a string and then outputs this string into file with the given name.

Chapter 6

Generating C code

6.1 C language issues

6.1.1 Separate source and headers files

The first thing that is more problematic in C are *separated header and source files*. The code generator thus needs to generate two types of files which are usually bound in pairs and contain interlinked information (headers contain function and type declarations, whereas source files define these functions - with the same signatures). Headers could be generated automatically from the resulting element code and contain declarations of all the non-static functions, variables and all the types defined by the corresponding source file, but this would lead to lack of control. This is solved by introducing header template files which are processed by STRATEGO transformations after the source file has been processed. Choice of the header template(s) for the source template is performed in the build script for the connector element as seen on Listing 6.1.

Listing 6.1: Script for generating header files

```
<script>
  <command action="StrategoCGenerator">
    <param name="class" value="TcpStub" />
    <param name="template" value="ellangc/tcp_stub.elc" />
    <param name="headertemplate"
         value="ellangc/element_header.elc" />
  </command>
  ...
</script>
```

6.1.2 Introducing interfaces into C

C programming language does not have any designated type for *interface definition* (as opposed to Java), so a convenient way of interface description needs to be established. The two possibilities which are easy to use from C code are C header file with function declarations and C struct containing function pointers. The latter one is chosen, because it is easier to handle by the parser and allows for definition of several interfaces in one header file (or directly in source file if this was needed). It also allows us to hold a pointer to an interface.

6.1.3 Interaction of components and connector elements

The hand-written component code needs to be able to communicate with the automatically generated connectors. The components have interfaces (required and provided) which represent the communication lines. A client component using functionality of a server component has a required interface and can call functions of that interface. The server component on the other hand implements some functionality and exposes it via a provided interface. The interfaces are thus the extension points where we would like to fit the generated connectors. The components would not need to know about the connectors in between, thus the communication logic would be transparent for them which is exactly what we want to achieve.

In Java where connector elements are classes implementing the component's business interface this approach is natural. There an element method can access element-specific data defined within the element class. In C we need to find a way to cope with the *lack of object-orientation*, so we workaround this feature by combining structs and function pointers, which is the way proposed by [44]. As mentioned before, an interface is just a structure with function pointers. We need to be able to call the interface function and pass the context to it (either some element-specific data or component-specific data). Thus each business interface method is obliged to contain **void** $*$ parameter as the first parameter (in addition to the business parameters that follow). This parameter is then used for passing the context.

To make the code of the components look cleaner and simpler, it would be nice if the components did not need to store these connector contexts as special parameters next to the interfaces they use. Therefore it is required to somehow store the context within the predefined business interface. This can be achieved even without modification of the interface struct by a simple trick - each component (and element) has its context data stored in a structure, when any of them provides an interface, then it contains the interface as a member of its context structure (not a pointer). Pointer to this interface is then passed to elements and components using this provided interface and is passed as a first argument of all interface functions. When a function gets a pointer to a provided interface, it can apply a bit of a pointer-magic to it and get to the context structure containing it, as displayed on Listings 6.2 and 6.3.

Listing 6.2: Common connector macros providing pointer arithetics

```
/* for easier use of the interface in component's code */
#define CALL(itf, func, args...) (itf)->func((itf), ##args)

/* cast the ptr (void *) to the given type */
#define DECLARE_SELF(ptr, type) \
  type *self __attribute__ ((unused)) = (type *)(ptr)

/* arithmetics - get from member ptr to 0th member in type */
#define INTERFACE_ENTRY(ptr, type, member) \
  ((type *)((char *)(ptr) - (unsigned long)(&((type *)0)->member)))

/* shortcut for declaring self variable from a member pointer */
#define GET_SELF(ptr, type, member) \
        DECLARE_SELF(INTERFACE_ENTRY(ptr, type, member), type)
```

Listing 6.3: Code passing context to interface function

```
struct business_itf {
  void (*set)(void *itf, int x);
  int (*get)(void *itf);
};

/* B component definition */
struct B_component {
  int value;
  struct business_itf provided;
};

void B_set(void *itf, int x)
{
  GET_SELF(itf, struct B_component, provided);
  self->value = x;
}

int B_get(void *itf)
{
  GET_SELF(itf, struct B_component, provided);
  return (self->value);
}

struct B_component compB = {
  .value = -1,
  .provided = {
    .set = B_set,
    .get = B_get
  }
};

/* A component definition */
struct A_component {
  struct business_itf *required;
};

struct A_component compA = {
  .required = &(compB.provided)
};

/* sample code */
int main()
{
  int x = 0;
  CALL(compA.required, set, 1);
  x = CALL(compA.required, get);
  return (x == 1 ? 0 : 1);
}
```

The interfaces for instantiation of connector elements, getting a required interface pointer from an element and setting a provided interface pointer to an element are discussed later in this work in Section 8.1.2.

6.2 Target language specific transformations

One of the supporting goals of this thesis is to identify the target language specific constructs used in the template transformations and separate them from the rest of the transformations, so that support for additional target languages can be brought more easily into the system. Most of these transformations have already been identified in the previous works and put into ElLang−J/src/java/ directory:

- *components.str* contains filename specific operations (resolution of classname to filename).
- *eval-iface.str* evaluates implementation of interface with expansion of method templates. It needs to define functions and their contents in the target language, thus it needs to be created separately for each target language.
- *generator.str* assimilates all the remaining DSL constructs into target language.
- *pp.str* contains transformations for pretty printing of the resulting AST to a file.

The following modules are identified by this thesis as target-language specific in addition to the above ones:

- *parser.str* contains transformations for parsing the mixture of ElLang and the target language.
- *disambiguate.str* contains target-language specific disambiguation transformations.
- *prepare-xml-interface.str* contains target-language specific transformations for the query module, this will be explained further in Section 6.6.1.
- *tls-rules.str* contains various target-language specific transformations dealing directly with code generation (e.g. generation of assign statement). This module is responsible for the creation of dynamic rules, which are then used from the evaluation modules common to all target languages.

All these Java-specific transformations were duplicated into src/C/ directory and modified to work with C code. The biggest problems arose with type information because, as we said in the previous chapter, it is stored both in the type specifiers and the declarator. Solution to this problem is discussed later in Section 6.5.

6.3 Parser and pretty printer

The parser and pretty print transformations use Stratego library functions in a way similar to the one shown on Listing 3.9 in Chapter 3. While the pretty printer transformations only allows printing of C code to a file or string, the parsing transformations allows parsing of both ElLang-C and pure C files (and strings). The pure C parser is used later for adaptation of C interfaces as described in Section 7.3.

While for Java the java−front project defines pp−java−string strategy for pretty-printing Java code into a string, we need to create our own strategy for C (with help of library functions), which is shown on Listing 6.4. This strategy uses C.pp pretty-print table, which was created by hand in the Box language introduced earlier in Section 3.4.1.

Listing 6.4: Strategy for pretty printing C code

```
pp-c-string =
  where (!"C.pp" => ppTemplateName);
  if is-file-exist(|ppTemplateName) then
      log-debug(|"Pretty printing C code using ", ppTemplateName)
    else
      fail-report-exit(|["File ", ppTemplateName,
          " does not exist"])
  end
  ; ast2abox(|[<parse-pptable-file> ppTemplateName])
  ; box2text-string(|80)
```

6.4 Interface evaluation

One of the most important parts of every template is the creation of the code specific to the business interface. ElLang construct implements interface SIGNATURE { /*interface code */}, which allows definition of a single template method code to be expanded to all methods in the interface, is used for this purpose. This template code is then transformed by the eval−iface transformation, which is responsible for parsing of the interface definition and calling transformation that generates the implementations of interface methods from the template method(s). This transformation dynamically creates additional meta-variables containing interface specific information, these variables can be used from the template code.

Four new meta-variables are created for the whole interface, ${interface.name} references the name of the interface, ${interface.definition} contains the definition of the interface structure type. New function for initialization of an interface structure is created for each template method defined within the interface implementation, ${interface.init.declarations} contains the declarations of these structure initializing functions and ${interface.init.definitions} variable expands to their implementations. The automatically created initialization methods can then be used from the template interface code, the names of the initialization methods start with the name of the interface, followed by index of the method template and _init string. For example the first template method for interface **struct** myiface would have the following signature **static void** myiface0_init(**struct** myiface * itf);.

6.4.1 Template methods

Current status

All the business interface methods are defined during method template evaluation, their implementation comes from the generic template method body which is adapted for the arguments of the methods. Most of the meta-variables available

inside the template method body are self-explanatory: ${method.name}, ${method.returnType}, ${method.declareReturnValue}, ${method.returnVar}. The return statement (if the method returns a value) is contained in the ${method.returnStm} meta-variable, the ${method.variables} contains a list of the names of all method arguments which can be used as a parameter list when calling other functions with the same arguments.

The work [48] created one more meta-variable ${method.args}, which is actually an array of structures and is modifiable from the template code. For each argument, its type is accessible and type.isPrimitive evaluates to true when the type of the argument is a Java primitive. This array can be then imploded into a comma separated list (for use as parameter list) by $implode(method.args)$ ElLang statement. There is also an expression bound to each of the variables, which is used for the implosion, by default it is set to the name of the variable, but it can be updated by $apply(expression for ARG in method.args where condition)$ statement. Additionally push, pop and peek statements can be used to append additional parameter, remove the last one or peek at its value.

Template method evaluation enhancements

First of all, the transformations producing the contents of all the meta-variables were redefined for C target language. Also the Stratego rules that are used to generate the declarations and definitions of expanded methods were redefined to generate C abstract syntax nodes, and a way of calling these methods was implemented as described before.

Secondly, for the purposes of C code generation, where we need to pass the context to functions (and we have chosen the first argument for that), we need to be able to also operate on the first argument of the arguments array. To enable this, new ElLang statements (and corresponding transformations) are defined:

```
"$" "peekfirst" "(" VarRefPart ")" "$" ->
    ArrayPeek {cons("ArrayPeekFirst")}
"$" "popfirst" "(" VarRefPart ")" "$" ->
    Stm {cons("ArrayPopFirst")}
"$" "prepend" "(" VarRefPart "," TargetLanguageExpr ")" "$" ->
    Stm {cons("ArrayPrepend")}
```

6.5 C type information

The type information in C declarations is spread between the type specifier and the declarator. More problems arise when considering parameter declarations, where the name of the parameter does not have to be defined, and function return types, where the name is not defined at all.

The C type is thus represented within the Stratego transformations as a tuple (TypeName, PointerDeclarator), and all the transformations dealing with types operate on this representation. Simple rules, get-return-type and get-param-pointer-part, were created for getting this representation from return type name and parameter declarator, where it might be a bit obfuscated. A sample rule dealing with types is shown on Listing 6.5, it transforms a type to a format string usable by the C printf(3) function.

Listing 6.5: Part of the rule resolving C type to a format string

```
/* The only supported pointer is char * */
get-type-formatter: (IntTypeName(constSpec, CharType()),
    PointerSpec(_, [Ptr([])])) -> "'%s'"
/* In other cases just print the pointer address */
get-type-formatter: (_, PointerSpec(_, _)) -> "%p"

get-type-formatter: (IntTypeName(None(), IntType()), _) -> "%d"
/* <snip> other formatters */
```

6.6 Other template language enhancements

6.6.1 Queries for interfaces

The template code can use special meta-variables, called queries, to get information about the input (information about the current element, its ports and bindings). The query module takes the information from the processed element descriptor which has XML format. The processing of the input element descriptor is performed as the first task of the element code generator. As of now, the query contains detailed information about the subelements for a composite element (their location within the repository, the names of the generated objects, etc.), the ports of the element and the signatures (names) of the interfaces.

For C we would like to have more details (like the names of the methods, names and types of their parameters) about the interfaces in the query module to be able to print element information or to use this data within template methods - as seen later in Section 8.6. Therefore we present a new module, C/prepare-xml-interface.str, which preparses the business interfaces and adds their description to the corresponding ports in the element descriptor. The description contains the names of all the interface methods, and for each of them it also contains the names and types of the parameters and the type of the return value.

6.6.2 Array creation

In addition to introduction of new operators that allow access to the first element of an array (thus converting a stack structure to a double-ended queue), new statement for custom array creation was added:

```
"$" "newarray" "(" VarRefPart ")" "$" -> Stm {cons("ArrayCreate")}
```

This allows for the creation of user defined lists, these can be populated in foreach statements from the queries (for example from the preparsed interfaces), and finally imploded into parameter lists for function calls.

6.6.3 Identifier concatenation

During creation of sample templates it turned out that there is a need for dynamical creation of new identifier names (variables, types or function names) based on the meta-variables (from the queries). For this, a new ElLang statement was defined:

```
%% varref concat
Id          -> IdCat
VarRef      -> IdCat
"$" "concat" "(" {IdCat ","}+ ")" "$" -> VarRef {cons("StrCat")}
```

This statement accepts any combination of variable references (meta-variables) and literal identifiers, and can be used in any place where identifier is expected.

Chapter 7

Modifications of SOFA 2 element generator

7.1 Running C code generator from connector generator

Section 2.2.4 presented the use of an action script for the generation of an element. For the purpose of element code generation, this thesis creates a new action, StrategoCGenerator, which maps to a Java class with the same name. This action is responsible for the generation of prerequisities of the element, preparation of element input descriptor, and starting the STRATEGO transformations defined in the previous Chapter.

7.1.1 Element generation prerequisities

Several tasks need to be performed before the element code generator can be started. Primitive elements need to have the business interface code prepared in the connector repository before they can be generated (the evaluation of interfaces in the templates starts with parsing of the interface definition). Composite elements need to have their subelements prepared in the repository, so that they know which header files to include and which methods to execute to instantiate the subelements.

After the template repository is filled in with the required code (either the interfaces or the subelements), the element descriptor, which contains location of the element template, the subelements, definition of element ports and the signature of the interfaces, can be constructed.

Primitive elements

Primitive elements need to have the source code of all the interfaces prepared before the element template transformation starts, because the template usually adapts the code to the interface and it needs its code for this task. Thus the interface needs to be fully resolved from symbolic type specifiers (which are our input now) to a type representation, and the code of the interface needs to be generated as explained further in the text in Section 7.3.2. All the elements defined so far have two interfaces, one on the input side where the element acts as a server and

one on the output side where the element acts as a client, thus code for both these interfaces is defined as shown on a simplified Listing 7.1.

Listing 7.1: Simplified code preparing interfaces for a primitive element

```
protected void prepareIfaces() throws ActionException {
  final PrimitiveElementPortsInfo primElPortInfo =
    new PrimitiveElementPortsInfo(this.elementDescriptor);
  final TypeFactory typeFactory =
    this.elementGenerator.getTypeFactory();
  final LanguageTools languageTools =
    this.elementGenerator.getLanguageTools();

  /* Simplified generation of interface for server port.
   * Not considering remote interfaces. */
  ResolvedElementPort serverPort = this.adaptationDescriptor.
    rei.getPort(primElPortInfo.clientPortName);

  SymbolicTypeSpecifier serverPortSig = serverPort.signature;

  CInterfaceDef serverCIface =
    (CInterfaceDef) typeFactory.getType(serverPortSig);
  String serverPackageName =
    languageTools.generateInterface(serverCIface);


  /* Simplified generation of interface for client port.
   * Not considering remote interfaces. */
  ResolvedElementPort clientPort = this.adaptationDescriptor.rei.
    getPort(primElPortInfo.clientPortName);

  SymbolicTypeSpecifier clientPortSig = clientPort.signature;

  CInterfaceDef clientCIface =
    (CInterfaceDef) typeFactory.getType(clientPortSig);
  String clientPackageName =
    languageTools.generateInterface(clientCIface);

  /* <snip> save the information about the generated interfaces */
}
```

The concept of a type factory for resolving symbolic type specifier to a type description and language tools used for generation of interface code is explained later in this Chapter in Section 7.3.

Composite elements

Composite elements on the other hand do not directly mediate any communication, instead they are built up from subelements (that can be either primitive or composite). Thus, the composite element does not need to generate any interface code (this will be handled by the subelements), but it needs to generate these subelements.

Listing 7.2: Simplified code generating subelements for a composite element

```
protected void prepareSubelements() throws ActionException {
  for (final ResolvedElementInstance inst:
```

```
        this.adaptationDescriptor.rei.subElementInst) {

      PackageDescriptor instPd = this.elementGenerator.generate(
        new AdaptationDescriptor(this.adaptationDescriptor, inst));

      this.packageDescriptor.runDependsOnPackage.add(instPd.
          packageName);

      /* <snip> save the information about the generated subelement */
    }
  }
```

The elementGenerator.generate() consecutively executes the whole action script for each of the subelements of the current element, thus the subelements are fully built before we proceed with the generation of the code for the composite element. This code usually includes the header files generated by the direct subelements and instantiates the subelements.

7.1.2 Element input descriptor

The low-level connector specification and the parameters of the StrategoCGenerator action, which are input for the element code generation, are converted to a format suitable for Stratego transformations, which was identified by work [47] to be XML. The information from which we generate this XML descriptor is gathered during the element preparation (e.g. the location of code of interfaces or subelements) and put together with other previously known data (template file location, destination folder, file name, etc.).

The element input descriptor contains the specification of all the ports of the element (including the signature and location of the business interfaces), and the connections between the ports. This information is used in the early stage of the transformation for construction of data for the query module.

7.1.3 Running Stratego transformations

The Stratego transformations introduced in the previous Chapter are compiled into a Java code and bundled into a Java JAR archive along with the element templates and other files necessary for ElLang template evaluation (e.g. a pretty print table), as was proposed by [48]. The main strategy from the archive can then be easily called by package_name.Main.mainNoExit(args); and command line arguments can be passed to it. This call is wrapped by the NativeStrategoCGeneratorBridge class.

The StrategoCGenerator only needs to pass the generated XML input descriptor to this class and ask it to generate the element code.

7.2 Compilation of C code

The purpose of the connector generator is not only to generate the source code of the elements, but also to produce binaries that will be usable from the rest of the component application (so that the component designer does not have to compile the connectors himself). This is achieved by introducing two new actions

into the element action script, one is responsible for generation of a Makefile and the second one for running the `make` program in the correct directory.

7.2.1 Makefile generation

Makefiles are created by the same action as the code files (StrategoCGenerator) because we need to perform similar kinds of tasks before the generation (we need to identify all the included headers, determine the directory for the sources, names of the generated files, list of the subelements). Action parameter with name makefile and value set to true is used to generate the Makefile instead of generating the code. We generate the Makefile directly from Java by printing its contents into a file once we have all the input information determined as shown by Listing 7.3.

Listing 7.3: Generation of makefiles by StrategoCGenerator

```
protected void generate() throws ActionException {
  this.isCompositeElement = !this.adaptationDescriptor.rei.
      subElementInst.isEmpty();

  /* prepareIfaces() for primitive element
   * prepareSubelements() for composite element */
  prepareElementClassInfo();

  if (actionType() == ACTION_TYPE.MAKEFILE)
    generateMakefile();
  else
    generateCode();
}
```

7.2.2 Running make

One new action, make, was presented for running the `make` program, it first determines the path to the previously generated Makefile from the AdaptationDescriptor associated with the action script and then runs the command `make -C DIR`.

7.2.3 Packaging the objects in archives

The goal of the compilation step is to make the usage of the generated binaries as easy as possible to the external code (e.g. a component application). When some external application uses a composite connector element, it does not only need to link against its object file, but also all the object files of all the subcomponents. This fact would make the deployment of applications using the connectors unnecessarily complex, and these applications would need to know the internal structure of the generated connectors to link with all the proper object files. This is not desired.

This thesis solves this problem by creation of object archives that contain all the object files of all the subcomponents for the composite component. Each element's Makefile thus contains a task for the generation of an object archive by the `ar` program. Primitive elements only put the single generated object file into the archive, while composite elements need to create an object archive that

contains the object file generated by the composite element and all the object files of all the subelements (which can also be composite). This is solved in two steps, the first step gathers all the object archives of all the subelements into a subdirectory and extracts all the archives, the second step constructs a new object archive with all the previously extracted object files and the object file of the composite element [1].

This approach allows the external application to be linked with the object archives created for the top-level connector units, thus only the top-level unit name is needed to be known for the generation of these applications.

7.3 C Type manager

The code generator needs to at least partially understand the business interfaces and to be able to adapt them to needs of a specific middleware, as was explained in Section 2.2.5.

This thesis introduces new type system for C language into the current type manager framework. Newly created type factory, CTypeFactory, supports two types: C interface and a parameter. In addition it supports several actions for modification of C interfaces (e.g. renaming interface or appending a parameter to all interface functions).

7.3.1 Understanding C code in Java

While for Java we use Class.forName() to parse an interface and to be able to work with it, there is no such thing as Java reflection for C code. Therefore we need to introduce a way to parse a C interface, possibly perform some modifications on it and print it back to a file. This kind of work is suitable for the Stratego/XT framework. We take advantage of having a C language grammar, a pretty print table and the corresponding Stratego transformations already defined, so we only need to bundle those in a way usable directly from Java.

For this purpose, a new Stratego module typemgr−c.str is defined. The transformations exported by this module are then compiled into Java and bundled into a Java JAR archive [2]. The compiled transformations are then directly usable from the Java code [26]. New Java class CStrategoTerm is introduced as a wrapper for calling the generated transformation functions (these have complex names and require parameteres wrapped into Stratego internal data structures). This class is then used as a Java representation of both C interfaces and parameters (and any other C type introduced in the C framework can be represented by this class).

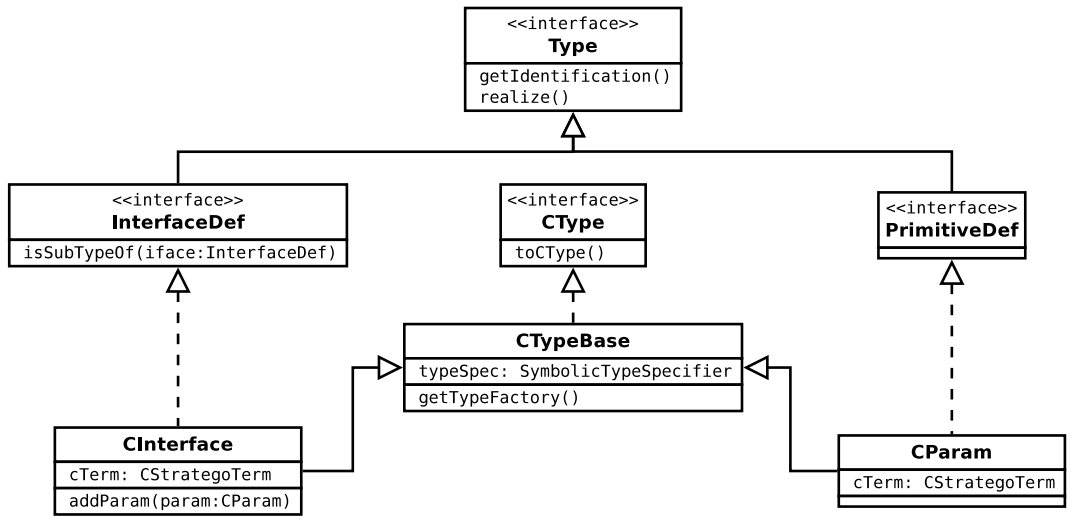
[1] Object archive cannot contain two files with the same name, so the object files that we are generating always have names based on their location in the connector repository which is unique (e.g. repository /connectors/A00000004/A00000004_ServerUnit.o).

[2] The STRJ compiler puts into the JAR archive only the transformations used (directly or indirectly) by the main strategy, but we want to have all the strategies defined by the typemgr−c module in there, so we create an artificial main strategy called all −transformations− list, which just calls all the exported strategies in a sequence (separated by semicolon).

7.3.2 C type system overview

C type system contains two types, CInterface and CParam, the class hierarchy is shown on Figure 7.1.

Figure 7.1: C type system class hierarchy



C interface

CInterface Java class represents a C interface. When it is being *realized* [3], it just creates a new CStrategoTerm and instructs it to parse an interface file as seen on Listing 7.4.

Listing 7.4: Parsing C interface by CStrategoTerm

```
public class CInterface extends CTypeBase implements InterfaceDef {
...
  public void realize() throws UnsupportedTypeException {
    TypeFactory rootTypeFactory =
        getTypeFactory().getRootTypeFactory();
    try {
      ifaceName = typeSpec.args[0].toString();
      String cIfacePathName = System.getProperty("user.dir") +
          File.separatorChar + ifaceName + ".h";

      /* parse the C interface to Stratego Aterm format */
      cTerm = new CStrategoTerm();
      cTerm.loadFromFile(cIfacePathName);
    } catch (CParserException e) {
      throw new UnsupportedTypeException (typeSpec, e);
    }
  }
...
}
```

[3] Realization of an interface means conversion of partial symbolic type specifier into a fully resolved type, which is understood by the type manager framework.

The CStrategoTerm class also implements a complementary function writeToFile(String fileName) that writes the current term representing the interface into a file, and function toCType() that constructs a FileNames Java class describing the name and the location of the code of the interface (further explained in Section 7.3.4).

For the support of actions implemented by C type factories, the interface can be cloned, renamed by renameInterface(), and parameters can be appended to all the functions by appendParam() function.

C parameter

The parameter is represented by a CParam Java class. It also uses a CStrategoTerm for parsing of a parameter from a string, storing its representation in ATerm format and printing it back to a string.

7.3.3 Interface adaptation

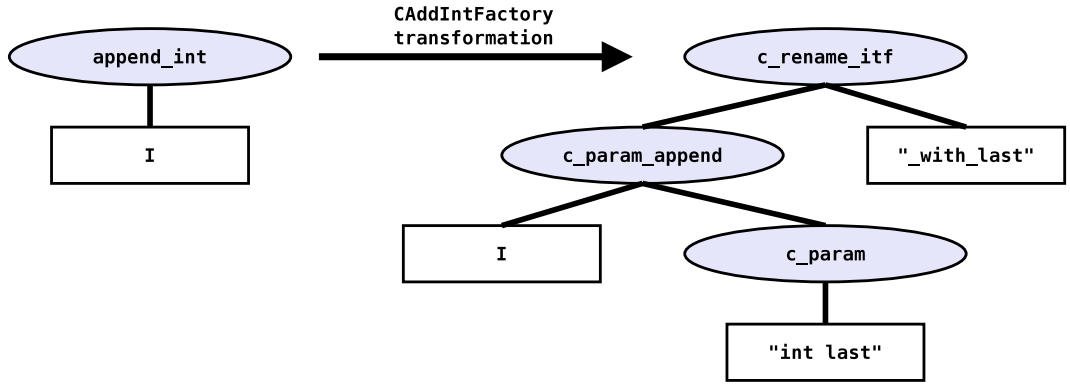
The CTypeFactory type factory implements actions for parsing of both C interface and C parameter, and it also supports three actions for adaptation of the interface. It is required that all the parameters of all the functions have assigned names for the purposes of template evaluation. The first adaptation action, c_name_params is used for this task. The other two interface operators, c_rename_itf and c_param_append, are used for renaming of an interface and for appending a parameter to all functions declared in the interface respectively. All these basic actions first clone the interface and then apply the desired modification to the cloned representation of the interface, as seen on the Listing 7.5. This approach is necessary to have correct type specifiers associated with correct interface objects.

Listing 7.5: CTypeFactory implementation of interface renaming

```
public Type getTypeDepthOne (SymbolicTypeSpecifier typeSpec)
    throws UnsupportedTypeException {

  if ("c_rename_itf".equals(typeSpec.funName)) {
    TypeFactory.checkTypeSpecArgs (typeSpec, ARGS_RENAME_ITF);
    CInterface iface = new CInterface((CInterface)typeSpec.args[0],
        typeSpec);
    String nameSuffix = typeSpec.args[1].toString();
    try {
      iface.renameInterface(iface.ifaceName + nameSuffix);
    } catch (CParserException e) {
      throw new UnsupportedTypeException("Cannot rename " +
          "interface!", e);
    }
    return iface;
  }
  /* <snip> other actions supported by the factory*/
}
```

Use of these actions from a middleware is illustrated in an artificial type factory CAddIntTypeFactory, which implements an action append_int by converting it to basic actions supported in the CTypeFactory, as depicted on Figure 7.2.

Figure 7.2: Expansion of symbolic type specifier by CAddIntTypeFactory



Symbolic type specifier evaluation

The symbolic type specifier is evaluated from the leaves to the root node. Let us have the following symbolic type specifier on the input:

```
append_int(c_interface(myitf))
```

At first, the interface type specifier (c_interface (iface_name)) is evaluated. This leads to expansion to c_name_params(c_parse_interface (iface), "param", " false"), because we want to enforce that all the parameters have assigned names. This modified specifier is then again evaluated from the leaves, c_parse_interface (iface) creates a corresponding CInterface object is and instructs it to load the interface with myitf name when being realized. Then the operator for assigning names to all parameters is evaluated by CTypeFactory, this clones the CInterface object and calls its nameParams() function to assign names to all the method parameters.

Then the append_int operator is processed by the CAddIntTypeFactory and expanded to the specifier on the Figure 7.2. The evaluation then proceeds by creating a CParam object from the **int** last string, the modified CInterface object is once again cloned, and the parameter is appended to this interface. This is then cloned for one more time, and the interface in the newest clone is renamed by appending " _with_last " string to its original name. And this is the final resolved type.

7.3.4 C language tools

The language tools concentrate target language specific tasks. Until now, only one class JavaTools has been used within the connector generator, this thesis introduces a new abstract generic class LanguageTools, which is extended by the tools specific for each language (e.g. JavaTools and CTools).

The tools are used for the generation of interface code from its description (the resolved type introduced above), and the Java tools are also used for the compilation of generated Java code and running RMI compiler on the code.

7.3.5 Interface file names

JavaNames class has been used for the transfer of information about a Java interface (it contained read-only attributes pointing to specific parts of its name, like packageName, className, fileName or relativeDirName).

This has been replaced by a generic abstract class FileNames, which provides the naming information about any source file in a generic way, and is subclassed by the target language specific name classes that set the initial values of these attributes (like the original JavaNames for Java and a newly created class CNames).

This allows the usage of the generic class in most cases in the whole connector generator, the specific classes are used only when new file name information needs to be created.

Chapter 8

Sample C connectors

One of the major goals of the thesis was to propose an interface for integration of the connectors with the whole component application and to create sample connectors, that could be used in component applications [1].

8.1 Integration of connector units

8.1.1 Intercomponent communication

The introduction of connectors that are mediating the communication is transparent to the components. In Section 6.1.2 we explained the represantation of the interfaces as C structs and we proposed the usage of additional **void** $*$ parameter (used for transfer of the context) prepended to argument list of every business method.

8.1.2 Connector units interfaces

Binding the ports

Each connector element is required to implement the interface depicted on Listing 8.1. By this we mean that each element implements two methods, lookupPort() which returns a port descriptor used for the communication on the element's port with the given portName, and setTarget() method which is used to initialize the binding of a port with name portName. The ports that are exposed to the components are always described by the business interface structure pointer, the representation of the ports for inter-element bindings fully depends on the element type (e.g. interface struct pointers for local calls, TCP address and port number for remote TCP connections). The type member of the interface identifies the type of the element, this field contains any combination of the following flags: local client, local server, remote client and remote server.

[1] Please note that the creation of runtime support for connectors (dynamic connector creation, automatic interconnection of the connector units, and creation of bindings between components and connector units) does not belong among the goals of this thesis. To be able to perform this task, we would need to have working support of C code from SOFA 2 docks, global connector manager and dock connector managers. This support would need to be usable from C code, so that the connectors can be dynamically created during runtime of the C application. None of the mentioned prerequisities is currently implemented, so the runtime connector support is left up to future works.

Listing 8.1: Public element interface

```
typedef struct connector_element {
  void *(*lookupPort)(void *element, const char *portName);
  int (*setTarget)(void *element, const char *portName,
      void *target);
  int type;
} connector_element_t;
```

Element creation and deletion

Each connector element also implements two functions with generated unique names, that are used for element creation and deletion. These methods are supposed to be exported by the header file generated for the element. The names of these methods are derived from the path of the element code in the repository and element unit name, for example element stored in A00000002 directory with name TcpStub would define the following two methods:

```
connector_element_t * new_element_A00000002_TcpStub(void );
void delete_element_A00000002_TcpStub(connector_element_t * el);
```

The first one creates a new element, initializes it and returns a pointer to its element interface. The second one can be used for stopping the service provided by the element and deallocating all the associated structures.

This approach has been chosen, because the names of these two methods can be easily found out from the element descriptor which is generated alongside the element code. In the future it should be possible to invoke these methods from some connector manager if the connector units are compiled as dynamic libraries.

Example use of the connector interfaces is illustrated in the case study in Section 9.1.

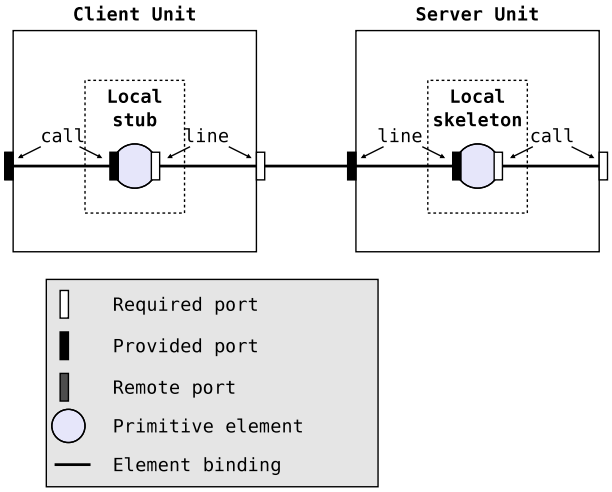
8.2 Local call

The simplest connector is the one for local method invocations. It does not employ any middleware for the communication.

8.2.1 Connector architecture

Local method call connector consists of two composite elements, ClientUnit which is on the side of the component with required interface, and ServerUnit which is on the side of the component providing the functionality. These two elements are independent of their subelements, so they can be used with different middleware technologies that work in a stub-skeleton fashion (e.g. with RPC as explained in the Section 8.5). The client unit requires a subelement that behaves as a stub, for local calls it is LocalStub. The server unit on the other hand supports subelements of skeleton type, thus local calls are handled by LocalSkeleton. The architecture of the local call connector is shown on Figure 8.1.

Figure 8.1: Local call connector architecture



8.2.2 Element templates

Compound default

compound_default.elc ElLang-C template is used for the generation of code for composite elements with any number of subelements. The template is written in a way that composite element initialization creates all its subelements and establishes bindings among them (binding description is one part of the query, so the code for connecting the ports is automatically created at the time of template processing). The element interface methods for port lookup and setting target are defined to delegate the call to the subelement that is providing the given port or requiring it in case of setting a target.

This template is used for the generation of both ClientUnit and ServerUnit code.

Local stub

For the client communication part of a local call connector, the local_stub.elc ElLang-C template is used. Listing 8.2 shows the definition of the context structure type used by this element.

Listing 8.2: Context structure for local stub element

```
#define MYITFTYPE $concat(local_stub_itf_, ${interface.name})$

typedef struct {
  connector_element_t element_info;
  struct ${interface.name} *target;
  struct ${interface.name} self;
} MYITFTYPE;
```

The element_info member contains the element's interface, the self member contains the provided interface (the element implementation of the interface business methods), and the target pointer is supposed to be set by setTarget() method to point to the interface provided by the local skeleton.

A template method is used for the generation of code for the business interface methods, these are all defined as a simple delegation of the call to the target's methods (if the return value of the function is not void, the value returned by the target's function is returned) as shown on Listing 8.3.

Listing 8.3: Template method definition of local stub element

```
element local_stub {
  implements interface ${ports.port(name=call).signature} {
    /* <snip> initialization and element methods left out */

    method template {
      ${method.declareReturnValue};
      GET_SELF($peekfirst(method.args)$, MYITFTYPE, self);

      if (self->target != NULL) {
        $popfirst(method.args)$
        $prepend(method.args, self->target)$

        $setReturnValue self->target->${method.name}(
            $implode(method.args)$)$;
      } else {
        $setReturnValue 0$;
      }

      ${method.returnStm};
    }
  }
}
```

Local skeleton

The server counterpart for the stub is defined in ElLang-C template local_skeleton.elc. This template looks almost the same as the local stub one, the only functional difference is in the type of the element stored in the element_info.type field, the local skeleton uses ELEMENT_TYPE_LOCAL_CLIENT | ELEMENT_TYPE_REMOTE_SERVER, which means that it acts locally as a client in its relationship to the component and as a server in its remote connection with the local stub. The stub uses ELEMENT_TYPE_LOCAL_SERVER | ELEMENT_TYPE_REMOTE_CLIENT combination, denoting that it is a local server for the component (waiting for its calls) and in the communication with the skeleton it behaves as a client.

8.3 Problems with C middlewares

The situation with remotely communicating elements is unfortunately a bit more problematic in C than in Java. Java has extensive support for remote communication built into the language itself (Serializable types, Remote interfaces) and a lot of support services (e.g. `rmiregistry`). In C on the other hand the developer needs to write all those features from scratch.

Another thing that is much harder in C, is context handling. In Java the objects transparently take care of context transfer, whereas in C we need to make huge effort to achieve that. Here we are mostly speaking about finding out

the context at the server (skeleton) side when new query arrives, the server application may provide many interfaces, i.e. services (even using the same middleware technology), so the context needs to be somehow provided by the middleware technology.

Due to the issues listed above we find out that some of the obvious middleware solutions turn out not to be easily usable in our scenario. Another problem is that there are not many middleware technologies for pure C, a lot of these are written in C++. It is possible to call C++ code from C, but we are creating generator of pure C code, thus we do not want to spoil it by introducing C++ middleware.

Almost every C middleware needs to generate some code, its input is some description of the communication interface, which is usually in different format for different technologies. The connector generator has an interface description in a form of C struct with function pointers, and both communication ends can easily generate functions with the same signature as those in the business interface (this is the role of the template method). So employment of a middleware usually brings the need to translate this interface description into some form of middleware-specific IDL (interface definition language) and to write some marshalling and demarshalling code in the element templates.

Another thing that needs to be considered when employing a middleware is whether it is self-contained or it needs other services running. The preferred option is the first one, because the entire application is independent and does not need to control some external service.

8.3.1 Existing middleware possibilities

In this section we will mention some of the possible C middlewares and explain, why they are not convenient for use in our scenario.

Sun RPC

The ONC RPC [18] (also known as Sun RPC) was considered as the first option because of its simplicity, but it has several problems.

First of all it requires running `portmap` application on the server to register the service provided (and to lookup the service required by the client).

Then in uses its own format for the definition of the communication, XDR which distantly resembles C. Utility `rpcgen` then generates a stub, a skeleton, a header and marshalling code, however, all the generated functions have modified return value type (it is a pointer to the returned value). And the function names on the servers have mangled names, they are not called the same as in the XDR, but the version of the interface is appended to each of them.

Finally, the RPC does not take care of the context at all.

To be able to employ this middleware, we would need to:

- Create generator of XDR files from interface structure definition, so that we have the interface representation in a format suitable for RPC and can generate the RPC stubs and skeletons from it.
- Create action for adaptation of the interface used for the server port, so that the function names are changed to the ones required by RPC and

the return types are the pointers to the original ones. This needs to be done, because the current template method implementation does not allow method renaming or modification of argument types.

- Find a way to transport context (or to somehow make it up on the server side), it cannot be in a global (or static global) variable, because an element code can be reused on the same dock for different purposes.

- The docks running the components would need to be able to run the `portmap` application.

Protocol Buffers

Protocol Buffers [19] is a Google project which offers a way of encoding structured data. It consist of a description of data interchange format, and a library that provides basic functions for construction of a buffer with a message from a structure holding the data according to a message descriptor. The opposite function constructs a data structure from a text buffer according to a message descriptor. The message descriptors and the structures for holding the data are automatically generated from message definition `.proto` files, which were specially designed to allow backwards compatibility when modifying the message format. There are three language mappings of Protocol Buffers for C++, Java and Python officially supported by Google, implementations for many other programming languages are provided by other groups.

The project interesting for us is the C implementation of Protocol Buffers [20]. C structures are used as a storage of the unpacked messages, their definitions, message descriptors and also functions for packing the data into a **char** $*$ buffer and unpacking it from the buffer are automatically generated by `proto-c` generator from the `.proto` message definition file.

The Protocol Buffers for C project also implements RPC functionality, where a service can be defined to provide several RPC functions. Each function accepts one message and it may also return a message. The communication is asynchronous, the client sends a request and specifies a callback to be called when the result message is received. The implementations of the service functions on the server side receive a pointer to the service (this could be used for context transfer), the decoded message structure, a callback function to be called when the return value is known and an argument to be passed to the callback.

To be able to employ this middleware, we would need to:

- Create generator of `.proto` files from interface structure definitions.

- Generate our own service routines for the server side which would call the element's implementation of the business functions with the arguments contained in the message structure, wait for the output and construct the return message.

- Cope with the generated names of the message structure types, macros and functions. These names have a common scheme, but the code using the code generated by the `proto-c` needs to understand the way these names are created and be able to generate the same names for the same things.

- On the client side we would also need to be able to construct a Protocol Buffer message from the parameters of a function, and call the appropriate generated service routine.

- To implement a method invocation connector which is synchronous, we would need to use some synchronization primitive (probably a semaphore) on the client side to turn the asynchronous response delivery into a synchronous method calls.

8.4 TCP RPC middleware

After discovering the troubles with existing middleware solutions, a decision was made to try to design simple middleware from scratch that would suit the needs of the connectors and that would be easily extendable or modifiable to fit additional criteria (e.g. to be used in real-time scenarios).

8.4.1 Design overview

The TCP middleware is designed as a very simple TCP service, that allows creation of a message, encoding several simple types into it, sending of a message and its reception on the other side, and finally decoding of the parameters from the message (based on the knowledge of the transported data types).

A server can be registered to listen on a defined port and wait for messages of a defined type, upon successful message reception a registered callback is called, the received message and an extra parameter used for context transfer are passed to it. There can be more servers registered within a single application.

A client allows to connect to a server. Messages can be sent in both directions, a special function for waiting for reception of single message allows the client to wait for an expected response from the server.

The middleware itself does not offer any code generator and it does not require any external service to be running on any of the hosts. The network communication is implemented in the Berkeley sockets API, thus it needs a UNIX-like system to run.

8.4.2 TCP server registry

In each application using the TCP middleware there is one global TCP registry that allows registration of TCP server services. This registry is not accessible from the application code, and it takes care of the following things:

- *Server registration* - server creation routine `tcp_server_create ()` automatically registers the created server in the registry. The registry either creates a new TCP socket for accepting client connections or ties the server to an already existing one (if there are more servers listening on the same port).

- *Accepting clients* - the registry listens on all the accepting sockets, and in case new connection is established (new client socket is accepted) it registers the new socket for data reception.

- *Message reception and delivery* - the registry monitors all the data reception sockets and reads the data from them to internal buffers. When it reads a full message, it delivers it to the server tied to the socket. This binding is established according to the message type at the time of reception of the first message on the data socket. The message delivery is performed by calling the callback function registered on the server with the context parameter and the received message.

- *Server deregistration* - when a server is destroyed by tcp_server_destroy () all the information about it is removed from the registry, this includes decreasing the reference count on the accepting sockets and also removing all the data reception sockets tied to the server.

The message type mentioned above, which is used for routing the data to the correct server, is a simple uint16_t value, which is encoded in the message header. The client and server communicating together need to agree on using the same message type.

The only registry routine, that is accessible from the application with some TCP servers registered, is tcp_service_poll (**int** mstimeout, **int** repeat). This function is used for handling all the data and new connections on all the registered sockets. For waiting for the data, the TCP registry uses a standard UNIX poll(2) function, the mstimeout parameter is used as a maximum waiting time of that function. The repeat parameter denotes how many times should we execute the poll(2) functions, a non-positive number is considered as infinite number of loops.

The fact that the registry allows to specify the maximum processing time makes the middleware usable even in single-threaded applications, this is not true about the Sun RPC middleware, where the svc_run() function never returns.

8.4.3 TCP message

The tcp_message_t, whose definition is shown on Listing 8.4, is the type for the only unit of information that is being transported by the middleware. An empty message is constructed by tcp_message_create() function, tcp_message_init () is used for its initialization, then encoding functions are used to encode values into the message, and a tcp_message_send() function sends the message to a file descriptor (a socket). The receiver of the message then uses the decoding functions to decode the information contained by it, finally, tcp_message_destroy() call destroys the message, or it can be reinitialized by tcp_message_init ().

Currently the middleware supports the following types: char, int16_t, int32_t, int64_t, uin16_t, uint32_t, uint64_t, float, double and char * for zero-delimited string, but the application is allowed to define its own encoding functions to encode different types into the message.

The encoding functions always need to make sure, that the data can fit to the buffer (that there is enough space between buf_pos and buf_size), then they can encode the data and move the buf_pos by the size of the encoded value. Finally, they return 0 in case of success and a negative value in case of failure. The decoding counterparts proceed exactly in opposite fashion, they return the decoded values and set the error code in the err parameter (if it was not NULL).

Listing 8.4: TCP Message structure definition and sample codec functions

```
typedef struct tcp_message {
  char * buf;          /* Buffer containing the data */
  uint32_t buf_pos;    /* Length of the valid data */
  uint32_t buf_size;   /* Size of the buffer */

  /* Fields for internal use: */
  uint32_t msg_len;    /* Expected message length */
  uint16_t msg_type;   /* Type of the message */
} tcp_message_t;

/* Sample encoding functions */
enum AllocationAction { NoAllocation, AllocateExact,
    AllocateExtra, AllocateMultiply };
int tcp_encode_int32(int32_t value, tcp_message_t *msg,
    enum AllocationAction allocate);
int tcp_encode_string(const char *value, tcp_message_t *msg,
    enum AllocationAction allocate);

/* Sample decoding functions */
int32_t tcp_decode_int32(tcp_message_t *msg, int *err);
char *tcp_decode_string(tcp_message_t *msg, int *err);

/* Wait for reception of a single message */
int tcp_message_receive(tcp_message_t *msg, int fd);
```

The message header, consisting of the message type and message length, is automatically encoded into the beginning of the message when tcp_send_message() function is invoked.

The tcp_message_receive() function can be used for waiting for reception of a single message on a socket.

8.4.4 TCP client

TCP client is used on the client side to establish a connection to a server. The basic client structure, tcp_client_t shown on Listing 8.5, can be created by tcp_client_create() function call, which initializes the message associated with the client, tcp_client_connect() then connects the client to a server with specified address. tcp_client_disconnect() and tcp_client_destroy() functions can be used to close the connection and deallocate the client structure.

Listing 8.5: TCP Client structure definition

```
typedef struct tcp_client {
  tcp_message_t *msg; /* Message associated with the client */
  int fd;             /* TCP socket of connected to the server */
} tcp_client_t;
```

8.4.5 TCP server

The server has already been partially described before, server creation automatically registers it within the registry, whereas destroying of the server also removes it from the registry. When a message is received by the registry, the callback function for the server associated to the data socket is called. The first parameter of

the callback function is defined by the creator of the server, this can be used for context transfer.

Listing 8.6: TCP Server structure definition

```
/* Callback function called by the~server upon message reception. */
typedef void (*tcp_callback)(void *param, tcp_message_t *message,
    uint32_t len, int fd);

typedef struct tcp_server {
  tcp_callback callback;    /* The callback function */
  void *param;              /* First parameter of the callback */
  uint16_t msg_type;        /* Supported message type */
  struct tcp_server *next;  /* Pointer to the next server */
} tcp_server_t;
```

8.4.6 Sample application

The Listings 8.7 and 8.8 show a part of an application using the middleware. Please note, that the destroy functions are written in a way, that they can be called even on a NULL pointer (to simplify the surrounding code).

Listing 8.7: Sample server code

```
#include <tcp_server.h>

int my_data = 12345;

void my_callback(void *param, tcp_message_t *message,
    uint32_t len, int fd)
{
  int err;
  int x = tcp_decode_int32(message, &err);
  char *text = tcp_decode_string(message, &err);

  if (err == 0)
    printf("Received value x=%d, text='%s'\n", x, text);
  else
    printf("Received corrupted message: %d!\n", err);
}

#define PORT 1234
#define MSG_TYPE 5678

void run_server()
{
  tcp_address_t address;
  tcp_server_t *server;
  tcp_address_init(&address, IPv4, "", PORT);

  server = tcp_server_create(&address, MSG_TYPE,
      my_callback, (void *)(&my_data));

  /* run forever */
  tcp_service_poll(100, 0);
}
```

Listing 8.8: Part of a sample client code

```
#include <tcp_client.h>

#define ADDR "127.0.0.1"
#define PORT 1234
#define MSG_TYPE 5678
#define BUF_LEN 128

#define VALUE 1000
#define TEXT "TCP RPC Middleware test message"

int connect_send_and_close()
{
  tcp_address_t srv_addr;
  tcp_client_t *client;
  int err;

  tcp_address_init(&srv_addr, IPv4, ADDR, PORT);
  client = tcp_client_create(MSG_TYPE, BUF_LEN);
  if (!client || tcp_client_connect(client, &srv_addr, 0) != 0) {
    tcp_client_destroy(client);
    return (-1);
  }

  err += tcp_encode_int32(VALUE, client->msg, AllocateExtra);
  err += tcp_encode_string(TEXT, client->msg, AllocateExtra);

  if (err != 0) {
    tcp_client_destroy(client);
    return (-1);
  }

  if (err = tcp_message_send(client->msg, client->fd) == 0)
    printf("Successfully sent the message!\n");
  else
    printf("Failed to send the message!\n");

  tcp_client_destroy(client);
  return (err);
}
```

8.5 Remote Procedure Call connector

For distributed calls we employ the middleware described above, which was designed to suit our needs.

8.5.1 Connector architecture

The architecture of the RPC middleware connector is very similar to the one for local calls that is displayed on Figure 8.1. Only the communicating primitive elements are changed to the ones specifically created for the middleware.

8.5.2 Middleware embedding

To enable easy use of the proposed middleware communication, we create a new action for the element's build script, TcpMwCodeGenerator. This action is used for the generation of the middleware stub and skeleton (depending on the value of a part parameter) from the interface description, this generated middleware code is then included by the element templates and used for the communication.

For the generation of TCP middleware stubs/skeletons we use STRATEGO transformations that are exported by the typemgr−c module (introduced in Section 7.3) and made accessible by CStrategoTerm Java class. The transformations convert an interface struct definition into several methods depending on the generated part, the use of the transformations is shown on Listing 8.9.

Listing 8.9: TCP Middleware stub and skeleton generator action

```
public class TcpMwCodeGenerator implements ActionInterface {
  /* <snip> constructor and members */

  public void perform(ElementDescriptor elementDescriptor,
      AdaptationDescriptor adaptationDescriptor,
      Collection<Property> actionParameters, PackageDescriptor pd)
      throws ActionException {

    String generatePart = Property.findFirst (actionParameters,
        "part");
    FileNames ifaceName = findIfaceName(elementDescriptor,
        adaptationDescriptor, generatePart.equals("stub"));

    CStrategoTerm cTerm = new CStrategoTerm();
    cTerm.loadFromFile(ifaceName.absoluteFileName);

    if (generatePart.equals("skeleton"))
      cTerm.generateTcpSkeleton(ifaceName.packageName + "/",
          ifaceName.className);
    else
      cTerm.generateTcpStub(ifaceName.packageName + "/",
          ifaceName.className);

    String destFolder = this.elementGenerator.
        getConnectorArtifactManager().
        getPackageDirectory(pd.packageName).
        getAbsolutePath() + "/";

    cTerm.saveToFile(destFolder + File.separator + "tcp_" +
        generatePart + "_" + ifaceName.className +  ".c");
  }

  /* <snip> other functions */
}
```

Middleware stub

The generated middleware stub contains definitions of all the interface methods, where each method constructs a TCP message, encodes the function identifier (the position of the function within the interface) and all the function parameters

into the message and sends the message using the tcp_client_t that is supposed to be passed to the stub function as the first argument (the one where we expect the context).

If the function has non-void return value, it instructs the client to wait for a response, decodes the return value from the received message and returns it. In case of no return value, the function just ends.

Interface structure variable with well-defined name (derived from the interface name) is also generated, it contains pointers to the generated functions and is supposed to be used from the stub element template. See Section A.1.2 for the example of the generated middleware stub code.

Middleware skeleton

The generated middleware skeleton defines a single callback function that is called by a TCP server when a new message is received. It first decodes the function identifier, according to that it proceeds with decoding of the parameters for the called function. The first parameter of the callback function contains user-defined parameter, in this case it contains pointer to the interface structure defined by the skeleton element. When all the parameters are decoded, corresponding function from the interface is called.

If the function has no return value, we are finished, otherwise a response message is constructed with encoded return value, sent back through the same socket and finally the response message is destroyed. See Section A.1.4 for the example of the generated middleware skeleton code.

8.5.3 Limited argument types

The middleware can only encode and decode scalar types and a zero-encoded strings, so these are the limitations for the parameter types of interfaces communicated via this connector.

8.5.4 Element templates

TCP stub

The TCP stub element template, tcp_stub.elc, includes the generated middleware stub code so that it can use the generated encoding functions. The generated element expects description of the TCP server (address and port number) when setTarget() is called for the remote port. At this time a TCP client is constructed, and a connection to the server (socket connected to the skeleton) is established.

The template method is defined in a way that it just calls the functions generated by the middleware stub generator, pointer to the TCP client is passed as the first argument. Section A.1.3 shows the generated code for a TCP stub element along with the interaction with the generated middleware stub code.

TCP skeleton

The TCP skeleton element template is very similar to the local skeleton template with several simple differences. TCP server is created when setTarget() is

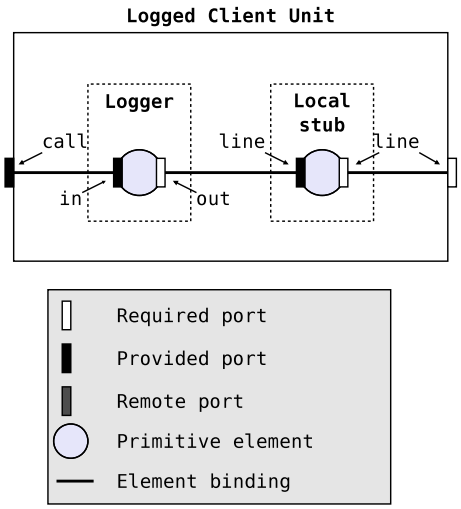
called for the remote port with the address and port description, the callback for the server is set to the function generated by the middleware skeleton generator and a pointer to the interface implemented by the skeleton element is set to be the argument of the callback function. After that, the lookupPort() function for the remote port returns server address description (this can then be used for initialization of the stub binding). The generated callback takes care of calling the right business interface function implemented by the skeleton element, so the template method looks exactly the same as for local skeleton.

8.6 Logging connectors

8.6.1 Connector architecture

When logging is required, it suffices to use different client unit and server unit composite elements, the communicating elements can stay the same as described above. The only two things that need to be done are to add logging subelement to the composite elements' architectures and to define a template for logging (delegating the call and printing the names of the methods and the values of arguments passed through). The new architecture of a logged client unit is shown on Figure 8.2.

Figure 8.2: Architecture of logged client unit composite element



8.6.2 Element templates

Client and server units

The compound default template defined before suits the needs of the logged composite elements, so it is used for both of them. A sample code generated for a logged server unit composite element is shown in Section A.2.1.

Logger element

The logger template is very similar to the local stub template, the only difference is that before the delegation of the call it needs to log the name of the method and the values of its parameters. Here we use the query data provided by preparsing the interfaces (as described earlier in Section 6.6.1). The query contains descriptions of all the methods including the names and types of the parameters (and a formatter for use in the printf(3) function). The logger's template method definition is shown on Listing 8.10 [2], which also shows a use of connector element inheritance. See Section A.2.2 for a sample code generated by this template.

Listing 8.10: Logging element template

```
element logger_console extends "primitive_default.elc" {
  $defExtPoint(primitiveMethodTemplate)$
    ${method.declareReturnValue};
    GET_SELF($peekfirst(method.args)$, MYITFTYPE, self);

    if (self->target != NULL) {
      printf("LOGGER: Called method ${method.name} with args\n(");

      $popfirst(method.args)$
      $prepend(method.args, self->target)$

      $foreach(DM in ${ports.port(name=in).description.method})$
        $if (DM.name == method.name)$
          $foreach(DPARAM in ${DM.params.param})$
            printf("${DPARAM.name} = ${DPARAM.formatter} ",
                ${DPARAM.name});
          $end$
        $end$
      $end$
      printf(")\n");

      $setReturnValue self->target->${method.name}(
          $implode(method.args)$)$;

      $if (method.returnVar) $
        printf("LOGGER: res = ${method.returnType.formatter}\n",
            ${method.returnVar});
      $else$
        printf("LOGGER: no return value\n", ${method.returnVar});
      $end$
    } else {
      printf(" !: NULL target -> no action\n");
      $setReturnValue 0$;
    }
    ${method.returnStm};
  $end$
}
```

[2] It might seem that much easier implementation of this functionality would be by extending the information provided by the method.args array. But this approach could not be used for this task because it is not possible to iterate through this array by ElLang foreach command.

Chapter 9

Evaluation

9.1 Case study - bank application

A simple component application was created to show, how the designed connector generator can be used by external applications. The application and the interface between the generated code and the hand-written code are described in this section, the sources of the application are attached on the enclosed CD in `src/codegen/case_study` directory.

9.1.1 Application design

The created component application simulates the tasks that could be performed in a bank. Our bank consists of a single central database, several cashpoints (ATMs) and a single branch with one accountant. The database is logging requests from ATMs to an external log, whereas all the communication between the database and the branch is logged automatically on the communication line. Thus we have four types of components in the system, bank representing the database, atm for the cashpoints, branch for the branch with the accountant and atm_log for the log into which the database is logging the ATM requests.

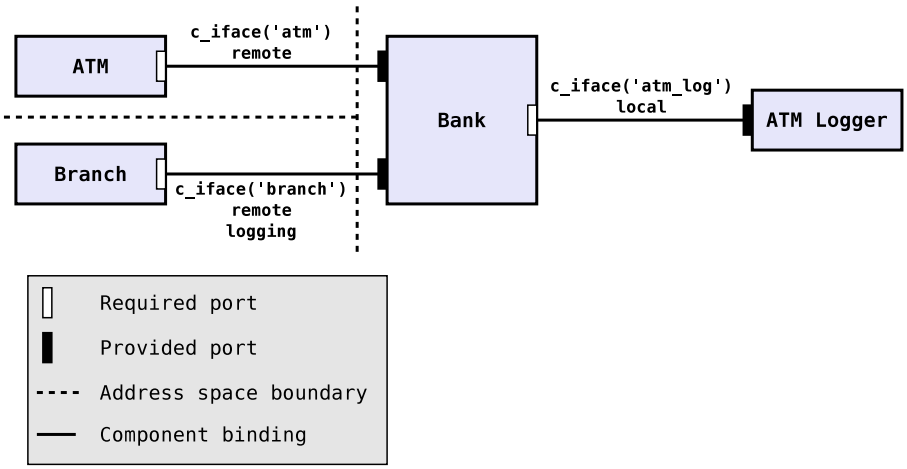
Figure 9.1 depicts the architecture of the bank application along with the desired distribution of the application.

9.1.2 Business interfaces

Creation of the communication interfaces is the first thing that needs to be done after designing the architecture of the application. These interfaces represent the service required by the client components and provided by the server ones. There are three communication interfaces in the application, one for each binding type, their definitions are shown on the Listings in this section. All the interfaces are represented by C structures, each saved in the file named same as the structures, and each of the functions contains an additional **void** $*$ argument as the first parameter. This parameter is used for context transfer, as was explained in Section 6.1.2.

The ATM can connect to the bank and obtain an account handle, this can then be used for querying the account balance or withdrawing money from the account. In such scenario the bank is informed about the withdrawal request, verifies it,

Figure 9.1: Architecture of the sample bank application



and in case of success subtracts the amount of money from the account and the ATM can then disburse the money.

Listing 9.1: ATM to bank interface

```
#ifndef ATM_H__
#define ATM_H__
struct atm {
  int (*login)(void *itf, unsigned long account,
    unsigned int pin);
  unsigned int (*remaining)(void *itf, int handle);
  int (* withdraw)(void *itf, int handle, unsigned int amount);
  void (* logout)(void *itf, int handle);
};
#endif
```

The branch is controlled by an accountant, thus the communication interface allows more complex operations like account creation, money deposit or transfer of money between accounts.

Listing 9.2: Branch to bank interface

```
#ifndef BRANCH_H__
#define BRANCH_H__
struct branch {
  int (* choose)(void *itf, char *name);
  unsigned long (* number)(void *itf, int handle);
  char *(* mail)(void *itf, int handle);
  unsigned int (* remaining)(void *itf, int handle);
  int (* create)(void *itf, char *name);
  int (* setPin)(void *itf, int handle, unsigned int pin);
  int (* setMail)(void *itf, int handle, char *mail);
  int (* transfer)(void *itf, int handle, unsigned int amount,
    unsigned long target);
  int (* deposit)(void *itf, int handle, unsigned int amount);
};
#endif
```

The interface for logging ATM requests by the bank database is very simple, it only contains a single method with message severity and message text as parameters. The example interface intentionally does not contain the names of all the arguments, so that we can show the automatic parameter name assignment functionality of the connector generator.

Listing 9.3: ATM requests logging interface

```
#ifndef ATM_LOG_H__
#define ATM_LOG_H__
enum LogSeverity { LOG_INFO = 0, LOG_WARN, LOG_ERROR };
struct atm_log {
  void (* log)(void *, enum LogSeverity severity, char *);
};
#endif
```

The code of all the business interfaces can be found in the `ifaces` subdirectory of the case study directory.

9.1.3 Creation of components

Two of the components, atm and branch are interactive components, awaiting user input (either from clients at the cashpoints or from the accountant at the branch), the bank and atm_log components wait for events from the previously named components.

Each component needs to be able to do four things: handle the storage of component data, return its provided interfaces, set its required interfaces and perform the business logic functionality. The specific interface for the first three tasks should be in the future designed by C connector runtime. Without it, the thesis proposes two types of functions:

```
void COMPONENT_init(REQUIRED_INTERFACE_POINTERS);
PROVIDED_INTERFACE * COMPONENT_get_itf_NAME();
```

The first one is used for the initialization of the component's required interfaces, and the second one should return the pointer to the provided interface structure of the given name. Each of the designed components is supposed to exist only once in each application, so they use a static global variable for storing their data. Each component implements all the business interface methods.

Listing 9.4: Bank component storage and interface initialization methods

```
/* storage for the component data */
struct BANK {
  struct account *accounts; /* List of accounts */
  struct atm machines;      /* ATM provided interface */
  struct branch branches;   /* Branch provided interface */
  struct atm_log *log;      /* ATM log required interface */
  int base_handle;          /* Internal implementation field */
};

static struct BANK mybank;

/* Component required functions initialization */
void bank_init(struct atm_log *log)
```

```
{
  /* Initialize the required interface */
  mybank.log = log;
  /* Initialize the provided interfaces */
  mybank.machines.login = bank_atm_login;
  mybank.machines.remaining = bank_atm_remaining;
  mybank.machines.withdraw = bank_atm_withdraw;
  mybank.machines.logout = bank_atm_logout;
  /* ... and branches interface ... and other data */
}

/* Return the pointers to the provided interfaces */
struct atm * bank_get_itf_atm()
{
  return (&mybank.machines);
}
struct branch * bank_get_itf_branch()
{
  return (&mybank.branches);
}
```

Then the components need to implement all the business methods of their provided interfaces, a sample implementation of bank's method for money withdrawal from ATM is shown on the Listing 9.5. It shows both how to get the component data structure from the context passed in the first argument and how to call a different component's provided method.

Listing 9.5: Implementation of business method by bank component

```
int bank_atm_withdraw(void *itf, int handle, unsigned int amount)
{
  GET_SELF(itf, struct BANK, machines);
  struct account * acc = find_account(self, handle, 1);
  if (acc == NULL)
    return (-2);
  if (acc->balance >= amount) {
    acc->balance -= amount;
    snprintf(buf, BUFLEN, "Withdraw from %lu %u units, left"
        " %u units", acc->number, amount, acc->balance);
    CALL(self->log, log, LOG_INFO, buf);
    return (0);
  } else {
    snprintf(buf, BUFLEN, "Cannot withdraw %u units from %lu,"
        " left %u units", amount, acc->number, acc->balance);
    CALL(self->log, log, LOG_WARN, buf);
    return (-1);
  }
}
```

The GET_SELF and CALL macros are defined in the connector_common.h header delivered by the C connector generator project. The implementation of all the components is saved in the components subdirectory in the case study directory.

9.1.4 Generation of connectors

The generation of connectors is independent of the implementation of the components. These two steps can be done separately, even by different developers.

The connection points of components and connectors are the business interfaces.

Connector high-level specifications

The input of the connector generator is the high-level specification of the connector, which was previously described in Section 2.2.2, and the definition of the business interfaces. The specification defines the communication units and the docks on which they will be located, the interface used and other non-functional properties (like language of the generated code or required communication style). The sample high-level specification of the connector used for connecting branch to a bank, where each resides in a different address space, is shown on Listing 9.6. Here we also specify that we want the server unit to log all the communication. This then results in a bit more complicated server connector unit architecture described previously in Section 8.6.

Listing 9.6: Branch to bank connector specification

```
<!-- Branch to bank connector -->
<specification>
  <unit name="client_unit" dock="dockA">
    <nfp-requirement predicate="nfp_mapping(Unit,'language','c')" />
    <nfp-requirement predicate="nfp_mapping(Unit,
      'communication_style','method_invocation')" />
    <port name="call" type="provided"
      signature="c_interface('branch')" />
  </unit>

  <unit name="server_unit" dock="dockB">
    <nfp-requirement predicate="nfp_mapping(Unit,'language','c')" />
    <nfp-requirement predicate="nfp_mapping(Unit,
      'communication_style','method_invocation')" />
    <nfp-requirement predicate="nfp_mapping(Unit,'logging', _)" />
    <port name="call" type="required"
      signature="c_interface('branch')" />
  </unit>
</specification>
```

Generating connectors

The connector generator needs to be invoked and provided with the high-level connector specification, it then performs the tasks described in Section 4.1.

Since starting the connector generator is not a straightforward task, an Ant `build.xml` script is created in the case study directory. The tasks implemented in the build script are capable of running the connector generator from the JAR in which it is distributed, and constructing the correct CLASSPATH and arguments for it.

In fact, generating the connectors by the `generate-connectors` Ant target is the only supported way for the sample application, because there is no C connector runtime, so the hand-written application using the connectors needs to have hard-coded paths to the generated connector units (and generating the connectors in different order would result in different unit names).

Generated code

Invoking the `ant generate-connectors` command starts the connector generator for all three connector specifications. This generates the connector elements' code in the connector repository, compiles it and packages it in object archives as previously described in Section 7.2, the list of generated object archives is shown on Listing 9.7.

Listing 9.7: Generated object archives in connector repository

```
case_study$ find repository/connectors/ -name '*.a' | sort
repository/connectors/A00000001/ClientUnit.a
repository/connectors/A00000002/TcpStub.a
repository/connectors/A00000004/ServerUnit.a
repository/connectors/A00000005/TcpSkeleton.a
repository/connectors/A00000006/ClientUnit.a
repository/connectors/A00000007/TcpStub.a
repository/connectors/A00000009/LoggedServerUnit.a
repository/connectors/A0000000A/LoggerConsole.a
repository/connectors/A0000000B/TcpSkeleton.a
repository/connectors/A0000000C/ClientUnit.a
repository/connectors/A0000000D/LocalStub.a
repository/connectors/A0000000F/ServerUnit.a
repository/connectors/A0000000G/LocalSkeleton.a
```

The directories also contain the generated source and header files of the elements and special entries in the repository also represent the generated business interfaces. For example, the atm_log generated interface is shown on the Listing 9.8, where we can see that the first and the third arguments already have assigned names (the interface definition given by the user previously shown on Listing 9.3 did not have these names set).

Listing 9.8: Generated ATM request logging interface

```
#ifndef ATM_LOG_H__
#define ATM_LOG_H__
enum LogSeverity { LOG_INFO = 0, LOG_WARN, LOG_ERROR };
struct atm_log
{
  void (*log)(void * param1,  enum LogSeverity severity,
      char * param3);
};
#endif
```

The connector generator task does not only generate the connector code (and objects), but it also saves the information about the architecture of the elements created for each high-level specification. There are two files created for every connector, a text file containing description of the selected architecture of the connector (with subelements, their ports, the interfaces used and the bindings), and a XML file describing only the top-level connector units, the connections between these two units and the directory where the connector unit code was generated. Both these file types are saved along the input high-level specifications in the `spec` directories. A sample XML file with connector description is shown on Listing 9.9. These files are used by the Java connector runtime for instantiation of the connectors and linking their ports, and they could also be used for a future project

delivering C connector runtime.

Listing 9.9: XML description of generated connector for branch to bank binding

```
<?xml version="1.0" encoding="UTF-8"?>
<output-descriptor>
  <unit package="A00000006">
    <exec-param name="implClass" value="ClientUnit" />
  </unit>
  <unit package="A00000009">
    <exec-param name="implClass" value="LoggedServerUnit" />
  </unit>
  <rbm>
    <architecture connectorArchName="method_invocation">
      <binding>
        <port unitName="server_unit" portName="line" />
        <port unitName="client_unit" portName="line" />
      </binding>
    </architecture>
  </rbm>
</output-descriptor>
```

9.1.5 Creation of applications

The hand-written components that were presented in this Chapter are not runnable binaries, they only contain the business logic implementation. To be able to test the connectors and the components, we need to create runnable applications using them.

There is no C connector runtime, which would ease the incorporation of the generated connector units within the application and would be able to create the connections between the connector units automatically.

So the applications using the components need to instantiate the corresponding connector units and connect them by hand. For the case study, three applications were created, one for the ATM, second for the branch and the third one for the bank (this also contains the logger for the ATM requests). Section 8.1.2 presented the interface for communication with the connector units from external applications. The declarations of the functions exported by the units are saved in the generated header files.

The relevant parts of code dealing with connector creation and establishing of bindings between the connector units and the component for the branch application are shown on Listing 9.10.

Listing 9.10: Creating and binding connector units for the branch application

```
#include <connector_common.h> /* the common connector declarations
    */

#include "../ifaces/atm.h" /* the business interface declaration */
#include "../components/atm.h" /* the component declarations */

/* Include the generated header files of the connector unit. */
#include "../repository/connectors/A00000006/ClientUnit.h"
connector_element_t *client_connector;
```

```
int create_connectors(char *srv_addr)
{
  /* Create the client connector - the subelements are created
   * automatically. */
  client_connector = new_element_A00000006_ClientUnit();

  /* Initialize the "line" port of the connector by the
   * given connection string. This port serves for communication
   * with the server connector unit. */
  return (client_connector->setTarget(client_connector, "line",
      (void *)srv_addr));
}

int main (int argc, char **argv)
{
  if (argc < 2) {
    return (1);
  }

  /* We expect to have the connection string for initialization of
   * the binding of the TCP client unit in the first argument. */
  if (create_connectors(argv[1]) != 0) {
    printf("Failed to connect to server: %s!\n", argv[1]);
    return (1);
  }

  /* Initialize the branch component's required interface. This
   * interface is provided by a component behind the connector.
   * "call" is the name of the port of the connector exposed to
   * the local component. */
  branch_init((struct branch *)client_connector->lookupPort(
      client_connector, "call"));

  /* Run the interactive component. */
  branch_run();

  /* Cleanup the connector data. */
  delete_element_A00000006_ClientUnit(client_connector);
  return (0);
}
```

The code sample shows how to instantiate a connector unit, query it for a pointer to an interface, set its port and finally delete the element. For example of creation of other types of bindings (e.g. from a connector to another connector or from a connector to a component) please see the `apps/bank_app.c` file in the case study directory containing the code of the bank application.

9.1.6 Compiling and running the applications

Prerequisities for running connector generator

The case study application requires the connector generator to be fully built and it also needs the C library for the connector generator to be compiled. These steps can be done by copying the contents of the enclosed CD [1] to a writable

[1] The explanation of the directory structure of the attached CD can be found in Chapter B.

disk and following the `README` file in the root directory of the copied structure. The steps that need to be done are:

1. Installation of Stratego/XT binaries and configuration of the path to the installed binaries in the connector generator configuration files.
2. Running `ant` in the directory of the connector generator. This automatically performs the following steps:
   (a) Generation of Java sources from the Stratego sources dealing with code transformation. This is followed by compilation of the generated sources and bundling the objects into a JAR archive.
   (b) Compilation of the Java sources of the connector generator.
   (c) Compilation of the common C connector code along with the TCP middleware code.

Compiling the case study applications

To compile the case study application, just enter the case study directory and execute `ant` program. The script performs the following actions:

1. Generation of the connectors from the high-level specifications.
2. Compilation of the code of the hand-written components.
3. Compilation of the code of the sample applications and linking it with the generated connector object archives.

Running the applications

After successful compilation of the sample applications, three binaries are created in the `apps` directory, `bank_app` representing the bank application, `atm_app` as interactive application for the cashpoints and `branch_app` as the application to be used at the branch.

The bank application needs to be started first, as it creates the servers, after that the atm and branch applications can be started. The information for establishing the connections from the client applications to the bank application needs to be passed from the bank. For this purpose, the bank prints two strings containing encoding of the connection parameters as displayed on Listing 9.11. The first connection string is supposed to be given to the `atm_app` as the first parameter on the command line, the second one is intended for the branch application, as shown on Listings 9.12 and 9.13.

Listing 9.11: Running the bank application

```
case_study$ apps/bank_app 127.0.0.1
ATM clients connection string:   '430390462127.0.0.1'
BRANCH client connection string: '430390d10127.0.0.1'
List of pre-created accounts:
 Account 123456, name John Doe, mail john@doe.com, handle 1383,
pin 1111, balance 10000
```

Listing 9.12: Running the ATM application

```
case_study$ apps/atm_app 430390462127.0.0.1
Pick your action ([L]ogin, [Q]uit):
```

Listing 9.13: Running the branch application

```
case_study$ apps/branch_app 430390d10127.0.0.1
Pick your action ([S]elect account, [C]reate, [Q]uit):
```

Users can interact with the two client applications, all the requests made locally are sent to the bank (the branch and atm components use required interfaces for using the functionality provided by the bank). As specified during the connector specification, all the function calls from branch to bank are logged on the bank application console. The bank logger component logs all the ATM requests into the `apps/atm_log_output.txt` file.

9.2 Lessons learned

Writing of the example application showed, that the proposed connector code generator is easily usable. The generated code and its structure is quite convenient for the use in hand-written application. However, this should be further improved by a separate project introducing C connector runtime. The important thing is, that from the component point of view, the communication via local call connector looks exactly the same as communication via remote call connector.

9.2.1 Middleware

Design of a custom middleware, which was previously described in Section 8.4, brought both advantages and disadvantages to the project.

Most of the good things were based on the fact, that the requirements for the use of the middleware were known in advance. Thus the important parts like construction of a message, data encoding and sending are easy to use. Then we knew that the server might be used in an application where other active components are running, so the function waiting for data on server sockets allows specification of parameters saying whether it can block forever or whether it should return after some time (so that other processing can be done). The environment where this middleware is being run also does not depend on any external tool (like `portmap`), which is a good thing. Designing a custom middleware also allowed us to design the stub and skeleton generator and their input and output formats. This way, we can use C structures for the input of the generator, and the generated methods are easily usable from connector elements. The stub and skeleton generator is based on some parts of the element code generator (the C grammar, parser, disambiguation, and some C specific transformations), thus the rest of the code generator does not depend on any external code generator tool, which would need to be delivered with it (e.g. `rpcgen`).

However, on the other hand we had to resolve all the problems that are already solved in other middleware technologies. This was done both within the middleware code and in the stub and skeleton generator. To name a few, we needed to implement the marshalling and unmarshalling code (considering both big-endian

and little-endian architectures), all the code for handling the work with TCP sockets (mainly registering several servers within one application), and the whole stub and skeleton generator.

9.2.2 Stratego/XT

Stratego/XT proved to be a powerful tool that solves all the pieces of program transformation process, starting from parsing, creation of transformation rules, up to printing the resulting code back to files. Due to its modularity, it was possible to reuse some parts from the previous SOFA 2 Java connector generator, mainly the ElLang grammar for the scripting language and the common transformations did not need to be touched at all. For the introduction of a new language only its SDF grammar needed to be created and the target language specific transformations (e.g. evaluation of interfaces) had to be rewritten.

Downsides of using Stratego

Extending the work of [48], which among others switched the implementation of the Stratego transformations from C code to Java, brought additional obstacles to debugging and development of the transformations, while making the resulting generator more easily deployable. The majority of the problems result from the long time that it takes to the `strj` compiler to create Java files from the Stratego transformations, and the Java compiler to compile these files. In addition the generated Java code is quite slow. Considering all these necessary steps together, it takes about three minutes to test even a simple change in the transformation. This makes the debugging and testing very problematic and lengthy.

Much bigger problems are caused by the differences between the quality of Java and C Stratego standard libraries. The biggest problems were encountered with the parsing SGLR library, where the older Java libraries did not parse all the tokens correctly (the ElLang variable references within strings looking like **#include** "${ports.port(name=call).signature}.h" were not recognized), and the newer ones parsing these tokens correctly took unreasonably long time - tens of seconds in comparison to less than a second that it takes to the `sglri` binary to parse the same file with the same parse table. We have chosen to use the newer library, which is the main reason of the slowness of the connector generation. It is important to bear in mind, that the Stratego/XT toolset is still in development and that it sometimes contains problematic bugs.

Although the disambiguation techniques for SDF are quite good, they unfortunately do not provide any means of generating dynamic disambiguation rules (e.g. when the parser encounters variable declaration, it immediately loses the information about the variable being declared - so we cannot use it further in the parsing). To not cope with ambiguous parse trees, we created quite restrictive grammar and presented a very simple disambiguation transformation to get rid of the rest of the ambiguities. This problem can be solved differently, for example the C-Transformers project [33] uses the SDF annotations (each reduction rule can be annotated) for generation of attributes for the tree nodes. They developed their own simple Attribute Grammar for this purpose and their C grammar does not

have any of the SDF disambigaution constructs. After the SGLR parser produces fully ambiguous parse tree, their disambiguation transformation is applied and it processes all the attributes of all the nodes. This is already performed in a transformation, so they can create dynamic rules and thus understand the disambiguated code much more deeply. However, their approach is not usable for us, because they need to preprocess all the macros and include all the headers, thus the generated code cannot be platform independent, and it is quite big and unreadable. By not having any disambiguation constructs directly in the grammar, it also happens that the SGLR parser hits its hard limit for maximum number of ambiguous nodes and stops working. They are working around this problem by providing the parser with smaller code pieces and then merging the resulting parse forests.

9.2.3 Connector generator framework

After putting together the connector architectures for C and for Java, the generation of the connectors started to be very slow. It was discovered that the major slowdown was caused by the architecture resolver, which is trying to generate every single possible combination of elements to fulfill the requirements in the high-level specification and choose the best one among them.

We did not want to make the generation of the connectors even slower, so a new command line argument `"congen.language"` for the connector generator was created to choose the subset of allowed connector architectures, its values can be `"C"`, `"Java"` or `"Both"`.

9.3 Different code transformation tools

9.3.1 Naive code generation techniques

One of the options for code generation is to use a general purpose scripting language like Python [21]. For many of these scripting languages there already exist extensions for easier integration with the text being generated, for example Mako project [14] for Python. The advantage of using these frameworks is having a scripting part which is usually well designed and thoroughly tested for free. But all of these frameworks have one major disadvantage, which makes them unusable for our purpose. They do not allow to control the grammar of the generated code (usually text or HTML code), thus syntactically incorrect code can be produced without any warning from the tools. The error is discovered at the time of usage of the generated code, which is very late.

The same problems with no control over the syntax of the produced code arise when using other template engines, like Apache Velocity [3] engine for processing of templates from Java.

9.3.2 Tools generating syntactically correct code

Beside the above mentioned tools, there exists quite a lot of frameworks for code generation or transformation that respect the grammar of the generated code.

XText [31] was the first framework considered when starting with the C code generator. It is a framework tightly integrated with the Eclipse Modeling Framework (EMF [9]) for the development of domain-specific languages. Its grammar is easily readable, it produces ECore models, that can be further processed and transformed by various Eclipse tools (ATL, Declarative and Procedural QVT), and then pretty-printed source code can be produced from these models by XTend or XPand tools. The biggest problem of this tool is its design, it is meant for small and simple DSL languages and generators of code for real languages (C, Java, C++) from definition of these simple languages (e.g. automatical generation of configuration parser from configuration language description), so it is not intended for parsing of large and complex languages, like C. It is based on ANTLR3 [2] LL(*) parser, the XText grammar does not allow left recursion in the production rules, it also does not allow handling of whitespace, and the grammar cannot be ambiguous. Due to these problems it turned out, that this tool is not usable for our purposes.

There are also tools that are specifically created for transformations of C programs like CIL (C Intermediate Language [4]), they usually contain their own parser and their own representation of syntax trees. Their biggest common problem is that they are not easily (or even at all) extensible, there is no way of defining a broader grammar which would allow us to mix some scripting language with the C code. This would result in a need of implementing all the templates for element codes as transformations in the programming language used by the respective tools (e.g. in Ocaml for CIL). This is not convenient for our purposes, where we want to have easily extensible set of connector elements.

All of the systems mentioned so far are either University research projects or tools developed by open community, they are all meant for free use and further research. There are also several commercial applications aimed at code optimalization, transformation and generation. One of the most advanced is the DMS (Design Maintenance System, [8]). It is very similar to Stratego/XT framework but it provides much more advanced tools and features designed for better understanding of the source code. It also uses a GLR parser, thus supports all the context-free languages and also their combinations. A huge set of front-ends for parsing is shipped with the tool, each front-end is capable of performing tasks specific for the language. For example the C front-end allows parsing of included files, handling preprocessor macros, constructing symbol tables for defined types and symbol names, call graph and data flow analysis are automatically created and accessible from transformations. These features allow faster generation of user defined transformations and deeper analysis and understanding the parsed code. But this toolset is unusable for our project, because it is commercial and not open-source.

Chapter 10

Related work

Generation of C code is not a new topic. There is quite a lot of existing projects dealing with C code generation even in the component system field. This chapter looks at the approaches to code generation used by some of these projects.

10.1 Ocarina and PolyORB-HI

Ocarina [17] is a toolsuite for AADL (Architecture Analysis and Design Language [1]) model manipulation, syntactic/semantic analysis, scheduling analysis, verification and code generation. It is designed as a compiler with a front-end used for parsing and analysis of the AADL model, it is modular and allows several code generation backends (currently Ada, C and AADL).

AADL has been designed by SAE (Society of Automotive Engineers), it allows to describe architecture of systems (both hardware and software) and, among others, it is being used for description of distributed real-time embedded systems. Developing the code of these systems by hand tends to be very tedious and error-prone, thus Ocarina aims at generation of the code of these systems from their AADL description and thus ensures that the code complies with the standards and other strong requirements (e.g. low memory footprint).

AADL is an extensible modelling language, which defines several components, each of them represents a hardware or software entity. Hardware components are a processor, a memory, a bus and a device. Software components can be a process, a thread, a subprogram and data. The components can contain subcomponents (e.g. process contains several threads, which contain subprograms). AADL components can declare features, which are their interfaces for communication with other components (connections between features are used to model the communication).

Ocarina is fully written in Ada, this means that also all the code generating the target language code from the AADL model is written by hand in Ada. The code generation has several steps, Ocarina first parses the AADL model and creates an abstract syntax tree from it. Then it expands it, so that the AST is closer to the structure of the generated code (e.g. the tree is separated into smaller ones according to the target node - application). At this time, the backend applies language translation transformation to the forest of AADL ASTs, and several new ASTs with target language code are generated in several steps (first the trees representing the common header files and common functions are generated, after

that the trees with code specific for the nodes are generated). Finally, these ASTs are traversed by a code generator and text files are produced. To compile the code, Makefiles for the code of each node are generated (they contain target architecture specific instructions) and a single Makefile for the whole application is created and the build process started.

The rules for mapping the AADL to C are introduced in [41]. These try to respect the High Integrity of the generated application by generating Ravenscar-like [36] code, which means that there is no dynamic allocation in the whole application, every possible initialization is performed statically at the time of code generation, the rest of them are performed in early initialization phase called before the application is fully started. Also every remote binding (to a remote node) is defined statically within the generated code. The generated code can be platform independent because of use of PolyORB-HI middleware very closely bound to the Ocarina project (in fact these are distributed together), this middleware shades the platform specifics. To reduce the memory footprint of the generated binaries, only small part of the middleware is always linked to the application, the other parts (like marshalling code) are automatically generated during the code generation phase (here only the parts actually needed by the application are generated).

10.2 THINK ConGen

THINK [42] is C implementation of Fractal component system, it allows to design component applications in ADL (Architecture Description Language), the communication interfaces in IDL (Interface Description language) and the implementation of the components in C with special annotations in comments (this modified C is called NuptC). Fractal is based on a hierarchical component model, and serves both for designing of component applications and as the runtime environment and supports advanced component features like dynamic reconfiguration.

Thanks to the full specification of the components, their bindings and deployment in ADL, and the NuptC annotations in the implementation code, the code generator is able to generate and build code for the whole application, which is specially optimized for the current deployment (e.g. local calls for bindings that are not supposed to be dynamically reconfigured can be inlined).

The annotations in NuptC specify the mapping between C symbol names and the architectural entities described in the ADL (e.g. mapping of function names to interface functions or variables to component attributes).

The code generator [46] is implemented in Java and works in two steps, first it loads and expands an AST and then it builds the code from the tree. Both these phases are implemented as component applications, thus can be easily extended by the developer by modifying the architecture of the loader or the builder composite components.

The *loader* is responsible for the generation of the abstract syntax tree from the input information. Each of the inputs (application architecture in ADL, interface description in IDL, and component implementation in NuptC) is handled by different sub-component, where each of them adds some information to the produced AST (e.g. the component implementation) or transforms it (e.g. by adding stub components between client and server components). This tree

contains a mixture of ADL and C, and as it is being expanded it starts to contain nodes with C abstract syntax trees.

After the tree is fully prepared, it is passed to the *builder* component application, which is responsible for the generation of the code of the whole application and its compilation. The whole generation is done in two phases, first an organizer identifies the tasks that need to be perfomed (by walking the AST) and constructs a task graph, after that the tasks are executed. Each task starts a specific builder for the given task type. The builder's input arguments are either parts of the input AST or outputs of other builders, the builder's output can be a source code, file or AST in the target language. This way, the generation of component definition file can be decomposed to generation of the component's source code, which consists of generation of server interfaces and generation of client interfaces.

This approach allows the builders to make decisions, ex. whether to optimize or not, independently of other builders. For example, a client interface builder could create a direct server interface function implementation function call in place of client interface function call, or even inline the function implementation.

Chapter 11

Conclusion and future work

11.1 Conclusion

The thesis fulfilled its main goal, which was the creation of C code generator suitable for generation of connectors. The work was based on the existing Java element code generator written in Stratego/XT toolset, that is based on template transformation. To achieve this task, several things had to be done.

First of all, a SDF grammar suitable for parsing of C code was created. This grammar was then mixed with the already existing ElLang template language grammar, and a combined ElLang-C grammar was created. It is used for parsing of C element template files, and also as the basis for the definition of the transformations (in order to ensure that correct abstract syntax tree is produced). C pretty-print table, which is a parse table counterpart for generation of textual version of code from its abstract syntax tree definition, was also created.

Before the transformation generating the C code from the template could be created, several issues brought by differences between Java and C needed to be solved. C structure with function pointers was identified as a suitable way of representation of C interfaces for connector generation. The problem of context transfer in non-objective C code was solved by a mandatory **void** $*$ parameter for all the business interface functions. Then basic structures for the representation of C connector elements were designed, as well as a simple library, with code common to composite elements and definitions making the use of connectors easier for applications, was created.

In the next spet the target language specific Stratego transformations were identified and rewritten to generate C code. During this phase, the ElLang template language was extended a bit (for example we needed to be able to both prepend and append parameters to the parameter list). The generated input XML used for queries was extended by more detailed information about the business interfaces (these were preparsed and information about the methods and their parameters was saved in the XML). This data can be used in the templates for creation of informative outputs about the elements as well as for gathering more information about the arguments of current method (or even other methods) in the template method evaluation.

The existing code generator, which uses the Stratego element generator, was extended to support generation of both Java and C connectors. A type system for representation and modification of C interfaces was created and incor-

porated into the type system framework. A way of compilation of the generated element C code by first generation of Makefiles for each generated connector element and then executing `make` in the correct directories was established and implemented. The composite elements create object archives consisting of object files of their subelements. This allows easier integration of the connectors within target applications.

Then several simple C connector architectures were designed and templates for their elements were created. During this phase, a new simple TCP middleware for C language was introduced, and a stub and skeleton code generator was created as a Stratego transformation module based on the grammar and transformations used for C element generation.

Finally, a sample application demonstrating the use of the generated connectors within a distributed component application was presented.

11.2 Future work

11.2.1 Creation of C connector runtime

The usage of the generated code (which is represented by header files and object archives within the connector repository) in external applications (e.g. a component application) is quite problematic, because the paths and names of the connectors units cannot be anticipated before the generation.

This interaction between the components and the connectors should be solved by a project introducing C connector runtime that would work similarily to the Java runtime introduced by [39]. The connector generator already generates a XML output descriptor of the connector (example is shown on Listing 9.9), which contains the location and names of the top-level units (that are the only ones that need to dealt with from the application) and the information about their connections. The C connector runtime could parse this file and generate the code of the main applications from these descriptions. Another possibility would be to parse this information from the component applications and dynamically load the connectors (this would mean that the connectors would need to be compiled into dynamically linked libraries).

11.2.2 Extending the template repository

One of the obvious future tasks is the extension of the current set of templates by introduction of templates for other communication styles (e.g. messaging) or elements providing other non-functional properties (e.g. encryption of the communication for ensuring greater security on the communication channel).

11.2.3 Concrete syntax for C and ElLang

Stratego/XT toolset allows to extend a SDF grammar by definition of special production rules, that then allow the usage of concrete syntax for transformations. This Stratego feature allows the use of parts of the target language code in the transformation rules instead of pure abstract syntax trees that are sometimes hard to visualize and map to the code. This usually results in simpler and

more readable transformations, thus extending the C and ElLang grammars to support concrete syntax would allow to simplify the current quite complex and nasty transformation rules (mainly those in the evaluation of the interfaces or the generation of stubs and skeletons of the TCP middleware).

Bibliography

[1] SAE Aerospace. Architecture Analysis & Design Language (AADL). AS-5506, SAE International, 2004.

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

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

[4] C Intermediate Language, http://cil.sourceforge.net/.

[5] Object Management Group, Common Object Request Broker Architecture (CORBA), http://www.omg.org/spec/CORBA/.

[6] Microsoft Corporation, Component Object Model, http://www.microsoft.com/com/.

[7] Object Management Group, CORBA Component Model, http://www.omg.org/spec/CCM/.

[8] The Design Maintenance System (DMS), A Tool for Automating Software Quality Enhancement, http://www.semdesigns.com.

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

[10] Oracle, Enterprise JavaBeans, http://www.oracle.com/technetwork/java/ejb-141389.html.

[11] Fractal Component System, http://fractal.ow2.org.

[12] Oracle, Jara RMI over IIOP, http://java.sun.com/products/rmi-iiop/.

[13] Microsoft Corporation, Microsoft Remote Procedure Call, http://technet.microsoft.com/en-us/library/cc759499%28WS.10%29.aspx.

[14] Mako Templates for Python, http://www.makotemplates.org/.

[15] Microsoft Corporation, .NET Framework, http://www.microsoft.com/net/.

[16] Microsoft Corporation, .NET Remoting, http://msdn.microsoft.com/en-us/library/kwdt6w2k%28v=VS.71%29.aspx.

[17] Ocarina Code Generation toolsuite, http://penelope.enst.fr/aadl/.

[18] ONC Remote Procedure Call Protocol, http://tools.ietf.org/html/rfc5531.

[19] Google, Protocol Buffers, http://code.google.com/p/protobuf/.

[20] Protocol Buffers C binding, http://code.google.com/p/protobuf-c/.

[21] Python Programming Language, http://www.python.org/.

[22] SOFA 2 Component System, http://sofa.ow2.org.

[23] Stratego Annotated Term Format, http://strategoxt.org/Tools/ATermFormat.

[24] Stratego c-tools Project, http://www.program-transformation.org/Stratego/CTools.

[25] Stratego/XT Framework, http://www.strategoxt.org/.

[26] Stratego integration with Java, http://strategoxt.org/Stratego/STRJ#Java_Integration.

[27] Stratego Java-front Project, http://www.strategoxt.org/Stratego/JavaFront.

[28] Stratego SDF2 Bundle Project, http://strategoxt.org/Sdf/SdfBundle

[29] Syntax Definition Formalism, http://www.syntax-definition.org/.

[30] Syntax Definition Formalism Manual, http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html.

[31] XText Language Development Frameworks, http://www.xtext.org.

[32] Yacc: Yet Another Compiler-Compiler, http://dinosaur.compilertools.net/yacc/.

[33] Borghi, A., David, V., Demaille, A. C-Transformers - A Framework to Write C Program Transformations. *ACM Crossroads* Volume 12 Issue 3, March 2006.

[34] Bravenboer, M., Visser, E. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'04)* (Vancouver, Canada, October 2004), D. C. Schmidt, Ed., ACM Press, pp. 365–383.

[35] Bravenboer, M., de Groot, R., 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.

[36] Bums, A., Dobbing, B., Romanski, G. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies, Proceedings of the Ada Europe Conference*, Uppsala, pp. 263-275. Springer Verlag, 1998.

[37] Bures, T. Generator of Connectors for SOFA/DCUP. Master thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2002.

[38] Bures, T., Hnetynka, P., Plasil, F. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proceedings of SERA 2006*, Seattle, USA, IEEE CS, pp. 40-48, August 2006.

[39] 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.

[40] Bures, T., Hnetynka, P., Plasil, F. Runtime Concepts of Hierarchical Software Components. In *International Journal of Computer & Information Science*, Vol. 8, No. S, pp. 454-463, September 2007.

[41] Delange, J., Hugues, J., Pautet, L., Zalila. B. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. In *4th European Congress ERTS*, Toulouse, 2008.

[42] Fassino, J.-P., Stefani, J.-B., Lawall, J., Muller, G. THINK: A Software Framework for Component-based Operating System Kernels. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, California, USA, June 2002.

[43] Hemel, Z., Kats, L., Visser, E. Code generation by model transformation. A case study in transformation modularity. In *International Conference on Model Transformation (ICMT 2008)*, Lecture Notes in Computer Science. Springer, June 2008.

[44] Hosek, P. Supporting real-time features in a hierarchical component system. Master thesis, Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University in Prague, 2010.

[45] de Jonge, M. To Reuse or To Be Reused: Techniques for Component Composition and Construction. PhD. Thesis, Faculty of Natural Sciences, Math., and Computer Science, Univ. of Amsterdam, January 2003.

[46] Lobry, O., Polakovic, J. Controlling the Performance Overhead of Component-Based Systems. In *Software Composition, 7th International Symposium*, pp. 149–156. Springer, 2008.

[47] Malohlava, M. Using Stratego/XT for Generation of Software Connectors. Master thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2006.

[48] Raszyk, J. Connector Generation Process Enhancement. Master thesis, Department of Distributed and Dependable Systems, Faculty of Mathematics and Physics, Charles University in Prague, 2010.

[49] Visser, E. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[50] Zalila, B., Hamid, J., Hugues, J, Pautet, L. Generating distributed high integrity applications from their architectural description. In *Ada-Europe'07: Proceedings of the 12th international conference on Reliable software technologies*, pp. 155-167, Berlin, Springer-Verlag, 2007.

Appendix A

Additional resources and examples

This Chapter contains listings with source files generated by the proposed C code generator. The role of the presented connector elements was explained earlier in Chapter 8.

A.1 Generated code for TCP middleware connector

A.1.1 Business interface

Listing A.1: Business interface used in this sample

```
/* A00000003/sample_iface.h file */
#ifndef SAMPLE_IFACE__
#define SAMPLE_IFACE__

struct sample_iface {
  long (*max)(void *iface, short x, int y, long z);
  char *(*repeat)(void *iface, char *input, unsigned int count);
  void (*store)(void *iface, unsigned long);
};

#endif
```

A.1.2 TCP middleware stub

Listing A.2: Generated stub code for the TCP middleware

```
/* A00000002/tcp_stub_sample_iface.c file */
#include <tcp_comm.h>
#include <tcp_client.h>
#include <strings.h>

#include "A00000003/sample_iface.h"
```

```
static long marshall_max(void * param_1,  short param_2,  int
    param_3,  long param_4)
{
  long res;
  int err = 0;
  tcp_client_t * client = (tcp_client_t * ) param_1;
  tcp_message_init(client->msg);
  err = tcp_encode_int16(1, client->msg, AllocateExtra);
  err += tcp_encode_int16(param_2, client->msg, AllocateExtra);
  err += tcp_encode_int32(param_3, client->msg, AllocateExtra);
  err += tcp_encode_int64(param_4, client->msg, AllocateExtra);
  if (err == 0)
    {
      err = tcp_message_send(client->msg, client->fd);
      if (err != 0)
        printf("TCP message could not be sent - function %s, error %
           d!\n", "max", err);
    }
  else
    printf("Params for function %s could not be encoded - %d!\n",
        "max", err);
  if (err == 0)
    {
      err = tcp_message_receive(client->msg, client->fd);
      res = tcp_decode_int64(client->msg, &err);
      if (err != 0)
        printf("TCP failed to read the return value for function %s
           from the socket\n", "max");
    }
  else
    bzero(&res, sizeof (res));
  return (res);
}
static char * marshall_repeat(void * param_1,  char * param_2,
    unsigned int param_3)
{
  char * res;
  int err = 0;
  tcp_client_t * client = (tcp_client_t * ) param_1;
  tcp_message_init(client->msg);
  err = tcp_encode_int16(2, client->msg, AllocateExtra);
  err += tcp_encode_string(param_2, client->msg, AllocateExtra);
  err += tcp_encode_uint32(param_3, client->msg, AllocateExtra);
  if (err == 0)
    {
      err = tcp_message_send(client->msg, client->fd);
      if (err != 0)
        printf("TCP message could not be sent - function %s, error %
           d!\n", "repeat", err);
    }
  else
    printf("Params for function %s could not be encoded - %d!\n",
        "repeat", err);
  if (err == 0)
    {
      err = tcp_message_receive(client->msg, client->fd);
      res = tcp_decode_string(client->msg, &err);
      if (err != 0)
```

```
        printf("TCP failed to read the return value for function %s
            from the socket\n", "repeat");
    }
  else
    bzero(&res, sizeof (res));
  return (res);
}
static void marshall_store(void * param_1,  unsigned long param_2)
{
  int err = 0;
  tcp_client_t * client = (tcp_client_t * ) param_1;
  tcp_message_init(client->msg);
  err = tcp_encode_int16(3, client->msg, AllocateExtra);
  err += tcp_encode_uint64(param_2, client->msg, AllocateExtra);
  if (err == 0)
    {
      err = tcp_message_send(client->msg, client->fd);
      if (err != 0)
        printf("TCP message could not be sent - function %s, error %
            d!\n", "store", err);
    }
  else
    printf("Params for function %s could not be encoded - %d!\n",
        "store", err);
}
struct sample_iface tcp_stub_itf_sample_iface = {
                                                   marshall_max,
                                                   marshall_repeat,
                                                   marshall_store
                                                 };
```

A.1.3 TCP element stub

Listing A.3: Generated code for the stub element using the middleware code

```
/* A00000002/TcpStub file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <connector_common.h>
#include <tcp_client.h>

#include "A00000002/TcpStub.h"
#include "A00000003/sample_iface.h"
#include "A00000002/tcp_stub_sample_iface.c"
#define MYITFTYPE tcp_stub_itf_type_sample_iface

typedef struct
        {
          connector_element_t element_info;
          struct sample_iface self;
          tcp_client_t * client;
          int myint;
        } MYITFTYPE;
static void sample_iface0_init(struct sample_iface * itf_);;
static void * lookupPort(void * element,  const char * portName)
```

```
{
  DECLARE_SELF(INTERFACE_ENTRY(element, MYITFTYPE, element_info),
      MYITFTYPE);
  if (strcmp("call", portName) == 0)
    return (&self->self);
  else
    return (NULL);
}
static int setTarget(void * element,  const char * portName,  void *
    target)
{
  DECLARE_SELF(INTERFACE_ENTRY(element, MYITFTYPE, element_info),
      MYITFTYPE);
  if (strcmp("line", portName) == 0)
    {
      tcp_address_t address;
      tcp_address_decode(&address, (char * ) target);
      tcp_client_destroy(self->client);
      self->client = tcp_client_create(64);
      if (self->client == NULL || tcp_client_connect(self->client, &
          address, 0) != 0)
        {
          printf("TCP Stub Error occured while connecting to %s:%u\n
              ", address.address, address.port);
          tcp_client_destroy(self->client);
          return (-1);
        }
      return (0);
    }
  else
    return (-1);
}
static long sample_iface0_max(void * iface,  short x,  int y,  long
    z)
{
  long b_0;
  DECLARE_SELF(INTERFACE_ENTRY(iface, MYITFTYPE, self), MYITFTYPE);
  printf(" C: called TcpStub 0x%p method %s\n", self, "max");
  if (self->client != NULL)
    {
      b_0 = tcp_stub_itf_sample_iface.max(self->client, x, y, z);
    }
  else
    {
      printf(" !: NULL target -> no action\n");
      b_0 = 0;
    }
  return (b_0);
}
static char * sample_iface0_repeat(void * iface,  char * input,
    unsigned int count)
{
  char * d_0;
  DECLARE_SELF(INTERFACE_ENTRY(iface, MYITFTYPE, self), MYITFTYPE);
  printf(" C: called TcpStub 0x%p method %s\n", self, "repeat");
  if (self->client != NULL)
    {
```

```
      d_0 = tcp_stub_itf_sample_iface.repeat(self->client, input,
          count);
    }
  else
    {
      printf(" !: NULL target -> no action\n");
      d_0 = 0;
    }
  return (d_0);
}
static void sample_iface0_store(void * iface,  unsigned long param2)
{

  DECLARE_SELF(INTERFACE_ENTRY(iface, MYITFTYPE, self), MYITFTYPE);
  printf(" C: called TcpStub 0x%p method %s\n", self, "store");
  if (self->client != NULL)
    {
      tcp_stub_itf_sample_iface.store(self->client, param2);
    }
  else
    {
      printf(" !: NULL target -> no action\n");
      0;
    }

}
static void sample_iface0_init(struct sample_iface * itf_)
{
  itf_->max = sample_iface0_max;
  itf_->repeat = sample_iface0_repeat;
  itf_->store = sample_iface0_store;
};
static unsigned int elem_cnt = 0;
connector_element_t * new_element_A00000002_TcpStub(void )
{
  MYITFTYPE * new_elem = (MYITFTYPE * ) malloc(sizeof (MYITFTYPE));
  if (new_elem == NULL)
    return (NULL);
  sample_iface0_init(&new_elem->self);
  new_elem->myint = 100 + (++elem_cnt);
  new_elem->client = NULL;
  new_elem->element_info.type = ELEMENT_TYPE_LOCAL_SERVER |
      ELEMENT_TYPE_REMOTE_CLIENT;
  connector_element_init(&new_elem->element_info, lookupPort,
      setTarget);
  return (&new_elem->element_info);
}
void delete_element_A00000002_TcpStub(connector_element_t * element)
{
  if (element != NULL)
    {
      DECLARE_SELF(INTERFACE_ENTRY(element, MYITFTYPE, element_info)
          , MYITFTYPE);
      tcp_client_destroy(self->client);
      free(self);
    }
}
#undef MYITFTYPE
```

A.1.4 TCP middleware skeleton

Listing A.4: Generated skeleton code for the TCP middleware

```
/* A00000006/tcp_skeleton_sample_iface.c file */
#include <tcp_comm.h>

#include "A00000003/sample_iface.h"

static void tcp_skeleton_callback_sample_iface(void * param,
    tcp_message_t * message,  uint32_t len,  int fd)
{
  int err = 0;
  int16_t func_type = tcp_decode_int16(message, &err);
  struct sample_iface * itf = (struct sample_iface * ) param;
  if (err != 0)
    return;
  switch (func_type)
    {
      case
        1:
        {
          int16_t param_1 = tcp_decode_int16(message, &err);
          int32_t param_2 = tcp_decode_int32(message, &err);
          int64_t param_3 = tcp_decode_int64(message, &err);
          if (err == 0)
            {
              long res = itf->max(itf, param_1, param_2, param_3);
              tcp_message_t * msg_ret = tcp_message_create(8);
              tcp_message_init(msg_ret);
              tcp_encode_int64(res, msg_ret, AllocateExact);
              err = tcp_message_send(msg_ret, fd);
              tcp_message_destroy(msg_ret);
              if (err != 0)
                printf("TCP failed to send back the return value for
                    function %s\n", "max");
            }
          else
            {
              printf("Params for function %s received incorrectly -
                 %d!\n", "max", err);
              tcp_message_discard(message);
            }
          break;
        }
      case
        2:
        {
          char * param_1 = tcp_decode_string(message, &err);
          uint32_t param_2 = tcp_decode_uint32(message, &err);
          if (err == 0)
            {
              char * res = itf->repeat(itf, param_1, param_2);
              tcp_message_t * msg_ret = tcp_message_create(8);
              tcp_message_init(msg_ret);
              tcp_encode_string(res, msg_ret, AllocateExact);
              err = tcp_message_send(msg_ret, fd);
              tcp_message_destroy(msg_ret);
```

```
            if (err != 0)
              printf("TCP failed to send back the return value for
                  function %s\n", "repeat");
          }
        else
          {
            printf("Params for function %s received incorrectly -
               %d!\n", "repeat", err);
            tcp_message_discard(message);
          }
        break;
      }
    case
      3:
      {
        uint64_t param_1 = tcp_decode_uint64(message, &err);
        if (err == 0)
          {
            itf->store(itf, param_1);
          }
        else
          {
            printf("Params for function %s received incorrectly -
               %d!\n", "store", err);
            tcp_message_discard(message);
          }
        break;
      }
  }
}
```

A.1.5 TCP element skeleton

The skeleton code is similar to the stub code so it is not shown here.

A.2 Generated code for logged server unit

A.2.1 Composite server unit element

Listing A.5: Generated code of logged server unit composite element

```
/* A00000004/LoggedServerUnit.c file */
#include <string.h>
#include <stdio.h>
#include <connector_common.h>

#include "A00000004/LoggedServerUnit.h"
#include "A00000005/LoggerConsole.h"
#include "A00000006/TcpSkeleton.h"

static void * compoundLookupPort(void * element,  const char *
   portName);
static int compoundSetTarget(void * element,  const char * portName,
     void * target);
connector_element_t * new_element_A00000004_LoggedServerUnit(void )
```

```
{
  compound_element_t * self = new_compound_element(2);
  connector_element_init(&self->element, compoundLookupPort,
     compoundSetTarget);
  self->subElements[0] = new_element_A00000005_LoggerConsole();
  self->subElements[1] = new_element_A00000006_TcpSkeleton();
  self->subElements[1]->setTarget(self->subElements[1], "call", self
     ->subElements[0]->lookupPort(self->subElements[0], "in"));
  return (&self->element);
}
void delete_element_A00000004_LoggedServerUnit(connector_element_t *
    element)
{
  if (element != NULL)
    {
      DECLARE_SELF(INTERFACE_ENTRY(element, compound_element_t,
          element), compound_element_t);
      delete_element_A00000005_LoggerConsole(self->subElements[0]);
      delete_element_A00000006_TcpSkeleton(self->subElements[1]);
      delete_compound_element(self);
    }
}
int get_element_A00000004_LoggedServerUnit_info(char * buf,  size_t
   pos,  size_t size,  size_t indent)
{
  const char * info = "Implementation: c_logged_server_unit\n";
  size_t mylen = strlen(info) + indent;
  size_t myindent;
  if (mylen >= (size - pos))
    return (-1);
  for (myindent = 0; myindent < indent; ++myindent)
    {
      buf[pos + myindent] = ' ';
    }
  strncpy(buf + pos + indent, info, size - pos - myindent);
  pos += mylen;
  ++indent;
  info = "Sub-element: logger_c_console_log \n";
  mylen = strlen(info) + indent;
  if (mylen >= (size - pos))
    return (-1);
  for (myindent = 0; myindent < indent; ++myindent)
    {
      buf[pos + myindent] = ' ';
    }
  strncpy(buf + pos + indent, info, size - pos - myindent);
  pos += mylen;
  info = "Sub-element: skeleton_c_tcp_skeleton \n";
  mylen = strlen(info) + indent;
  if (mylen >= (size - pos))
    return (-1);
  for (myindent = 0; myindent < indent; ++myindent)
    {
      buf[pos + myindent] = ' ';
    }
  strncpy(buf + pos + indent, info, size - pos - myindent);
  pos += mylen;
  buf[pos] = 0;
```

```
  return (pos);
}
static void * compoundLookupPort(void * element,  const char *
    portName)
{
  DECLARE_SELF(INTERFACE_ENTRY(element, compound_element_t, element)
      , compound_element_t);
  if (strcmp("line", portName) == 0)
    {
      return (self->subElements[1]->lookupPort(self->subElements[1],
          "line"));
    }
  else
    return (0);
  return (0);
}
static int compoundSetTarget(void * element,  const char * portName,
     void * target)
{
  DECLARE_SELF(INTERFACE_ENTRY(element, compound_element_t, element)
      , compound_element_t);
  if (strcmp("call", portName) == 0)
    {
      return (self->subElements[0]->setTarget(self->subElements[0],
         "out", target));
    }
  else
    if (strcmp("line", portName) == 0)
      {
        return (self->subElements[1]->setTarget(self->subElements
           [1], "line", target));
      }
    else
      return (-1);
  return (-1);
}
```

A.2.2 Logger element generated code

Listing A.6: Generated code of logged server unit composite element

```
/* A00000005/LoggerConsole.c file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <connector_common.h>

#include "A00000005/LoggerConsole.h"
#include "A00000003/sample_iface.h"

#define MYITFTYPE local_stub_itf_sample_iface

typedef struct
        {
          connector_element_t element_info;
          struct sample_iface * target;
```

```
          struct sample_iface self;
        } MYITFTYPE;
static void sample_iface0_init(struct sample_iface * itf_);;
static void * lookupPort(void * element,  const char * portName)
{
  DECLARE_SELF(INTERFACE_ENTRY(element, MYITFTYPE, element_info),
     MYITFTYPE);
  if (strcmp("in", portName) == 0)
    return (&self->self);
  else
    return (NULL);
}
static int setTarget(void * element,  const char * portName,  void *
    target)
{
  DECLARE_SELF(INTERFACE_ENTRY(element, MYITFTYPE, element_info),
     MYITFTYPE);
  if (strcmp("out", portName) == 0)
    {
      self->target = (struct sample_iface * ) target;
      return (0);
    }
  else
    return (-1);
}
static long sample_iface0_max(void * iface,  short x,  int y,  long
   z)
{
  long g_0;
  DECLARE_SELF(INTERFACE_ENTRY(iface, MYITFTYPE, self), MYITFTYPE);
  printf("LOGGER: Called method max with params \n ( ");
  if (self->target != NULL)
    {
      printf("iface = %p ", iface);
      printf("x = %d ", x);
      printf("y = %d ", y);
      printf("z = %ld ", z);
      printf(")\n");
      g_0 = self->target->max(self->target, x, y, z);

      printf("LOGGER: res = %ld\n", g_0);
    }
  else
    {
      printf(" !: NULL target -> no action\n");
      g_0 = 0;

    }
  return (g_0);
}
static char * sample_iface0_repeat(void * iface,  char * input,
   unsigned int count)
{
  char * i_0;
  DECLARE_SELF(INTERFACE_ENTRY(iface, MYITFTYPE, self), MYITFTYPE);
  printf("LOGGER: Called method repeat with params \n ( ");
  if (self->target != NULL)
    {
```

```
      printf("iface = %p ", iface);
      printf("input = '%s' ", input);
      printf("count = %u ", count);
      printf(")\n");
      i_0 = self->target->repeat(self->target, input, count);

      printf("LOGGER: res = '%s'\n", i_0);
    }
  else
    {
      printf(" !: NULL target -> no action\n");
      i_0 = 0;

    }
  return (i_0);
}
static void sample_iface0_store(void * iface,  unsigned long param2)
{

  DECLARE_SELF(INTERFACE_ENTRY(iface, MYITFTYPE, self), MYITFTYPE);
  printf("LOGGER: Called method store with params \n ( ");
  if (self->target != NULL)
    {
      printf("iface = %p ", iface);
      printf("param2 = %lu ", param2);
      printf(")\n");
      self->target->store(self->target, param2);

      printf("LOGGER: no return value\n", 0);
    }
  else
    {
      printf(" !: NULL target -> no action\n");
      0;

    }

}
static void sample_iface0_init(struct sample_iface * itf_)
{
  itf_->max = sample_iface0_max;
  itf_->repeat = sample_iface0_repeat;
  itf_->store = sample_iface0_store;
};
connector_element_t * new_element_A00000005_LoggerConsole(void )
{
  MYITFTYPE * new_elem = (MYITFTYPE * ) malloc(sizeof (MYITFTYPE));
  if (new_elem == NULL)
    return (NULL);
  sample_iface0_init(&new_elem->self);
  new_elem->element_info.type = ELEMENT_TYPE_LOCAL_SERVER |
     ELEMENT_TYPE_REMOTE_CLIENT;
  connector_element_init(&new_elem->element_info, lookupPort,
     setTarget);
  return (&new_elem->element_info);
}
void delete_element_A00000005_LoggerConsole(connector_element_t *
   element)
```

```
{
  if (element != NULL)
    {
      DECLARE_SELF(INTERFACE_ENTRY(element, MYITFTYPE, element_info)
          , MYITFTYPE);
      free(self);
    }
}
#undef MYITFTYPE
```

Appendix B

Contents of the attached CD

This thesis is accompanied by a CD-ROM which contains the implementation of the connector generator discussed in this thesis and other relevant files. The CD-ROM has the following structure:

/prerequisities/
: Required software packages for successful compilation of the connector generator.

/implementation.zip
: Zip file with the implementation, this was created to preserve deep directory hierarchy and file permissions. The zip file has the following structure:

  /build/
  : Part of the SOFA 2 infrastructure necessary for the compilation of the connector generator.

  /src/case_study/
  : Source code of the case study application described in Section 9.1.

  /src/congen/
  : Source code of the SOFA 2 connector generator.

  /src/congen/tests/
  : Several testing high-level connector specification files and a build script for generating the connectors from these specifications.

/README
: Brief description of the contents of the CD-ROM and the steps necessary for the compilation of the connector generator from its sources.

/thesis.pdf
: Electronic version of this thesis.