Issued: March 1, 2006

The Triple-M software infrastructure for YAPI and TTL source code analysis and transformations

D. Alders, O. Popp

Public Domain

Public Domain

Authors' address data: D. Alders ; d.alders@users.sourceforge.net
O. Popp ; ondrejpopp@users.sourceforge.net

:

Title: The Triple-M software infrastructure for YAPI and TTL source code analysis and transformations

Author(s): D. Alders, O. Popp

Part of project:

Customer:

Keywords: Compiler Technology; Multiprocessor mapping; System Level Design Technology; TTL; YAPI

Abstract: A high level view on the software infrastructure developed and used within the Triple-M project is presented to the reader. He or she is guided through the various tools of which only the most important implementation aspects are explained. These comprise tools with which the infrastructure is build and maintained - *C3PO* and ANTLR -, as well as the Triple-M specific tool called *SCATE* (Source Code Analysis and Transformation Environment) and the back-end tool *MPC*. At various places pointers are presented for further information.

Conclusions: After reading, the reader should be able to understand the overall picture of the software infrastructure. The manner in which the software is maintained and tested is explained as well. Design decisions are documented, and transformations are explained.

Also some code examples are presented on how to steer source code transformations using the *SCATE* tool. The examples are included here for illustration. This document is not meant to be a user manual.

Public Domain

Contents

1 Introduction **1**

1.1 Objective 1

1.2 Informal problem statement 1

1.3 YAPI 2

1.3.1 VYA 3

1.4 TTL 3

1.5 Solution approach 3

1.6 Intended audience 5

1.7 Outline of document 5

2 Overview **6**

2.1 The Triple-M tools 6

2.2 The C3PO meta level 7

2.3 SCATE components 8

3 Software synthesis with C3PO **10**

3.1 How to build a solid infrastructure 10

3.2 C3PO components 11

3.3 Datatype synthesis 12

3.4 C++ code generation 13

3.4.1 The Generate script 14

3.4.2 Base classes 15

3.4.3 Customizable classes 16

3.4.4 Robustness 17

3.4.5 The DepthFirstVisitor class 18

4 Adding functionality on top of the infrastructure **19**

4.1 Sequential container access 19

4.2 Generic AST transformations 20

4.2.1 Visitor patterns 20

4.2.2 The ParseFromString functions 22

4.3 Maintenance 23

4.3.1 CVS based patching 23

4.3.2 The mkHeader script 23

5 The C++ parser frontend 24

5.1 History 24
5.2 The symbol table 25
5.3 Using a conventional C / C++ preprocessor 26
5.4 Multi file compilation 27
5.5 Provisions for single file compilation target 28
5.6 C++ grammar extensions 29
5.7 The PreProcessingVisitor 29

6 The transform language 31

6.1 Transformation 31
6.2 Mapping specification 32
6.3 Addressing of units and instances 33
6.3.1 Structural transformation statements 35
6.3.2 Processor mapping 36
6.3.3 Transport 36
6.3.4 Net mapping 38
6.3.5 Variable mapping 39
6.3.6 Task mapping 40
6.4 API definitions 41
6.4.1 Language 42
6.4.2 Identification 42
6.4.3 Process and Network 42
6.4.4 Include and library option arguments 43
6.4.5 Header files 44
6.4.6 The main and type member function signatures 46
6.4.7 MPC 46
6.4.8 Channels 46
6.5 Build configuration 52

7 YAPI and TTL recognition 54

7.1 The YAPI representation 54

7.2 The YAPI visitor . . . 54
7.3 API validation . . . 55
7.4 Dynamic channel size derivation . . . 55
7.5 Options . . . 55

8 Transformation **56**
8.1 Visit schemes . . . 56
8.2 Structural transformations . . . 56
8.3 Functional transformations . . . 57
8.3.1 Preprocessing . . . 57
8.3.2 Making indirect access explicit . . . 62
8.3.3 Synchronization . . . 65
8.3.4 Postprocessing . . . 69
8.3.5 C++ to C conversion . . . 69

9 Code generation **73**
9.1 MPC input . . . 73
9.1.1 API specific steering . . . 73
9.2 The make system . . . 74
9.2.1 setup . . . 74
9.2.2 Makefile-Scate . . . 74
9.2.3 Make.SrcVars . . . 75
9.2.4 Make.MpcSetup . . . 75
9.2.5 Makefile-Mpc . . . 75
9.3 The mapping output files . . . 75
9.4 Recapitulation . . . 76
9.5 Graphical views . . . 76
9.5.1 SequoiaView . . . 76
9.5.2 Network view . . . 78
9.5.3 Visualisation of the file and component structure . . . 81

10 Testing **85**
10.1 Local Tests . . . 85
10.2 Regression Testing . . . 86

10.3 The cvsDiff filter script 87

11 Conclusions and future work **88**

11.1 Conclusions 88

11.2 Future work 88

11.2.1 Generic C++ / C transformation infrastructure 88

11.2.2 Legacy code transformation 88

11.2.3 GUI 89

11.2.4 API conversion 89

A C3PO generated API **90**

A.1 Member functions for the whole class 90

A.2 Member functions for scalar members 93

A.3 Member functions for collection members 94

A.4 The Enumeration datatypes 99

A.5 Member functions for enumerated types 105

A.6 Member functions for the validity layer 106

A.7 ANTLR parse function descriptions 107

A.8 The parseinfo datatype 108

A.9 The root parse functions 108

B The YAPI / TTL representation **109**

B.1 Structure 109

B.2 Functionality 109

C Command line options **111**

C.1 MPC related options 111

C.2 YAPI / TTL related options 111

C.3 options that control the transformation infrastructure 112

C.4 Makefile related options 112

D Install Directives **113**

D.1 Ports 113

D.1.1 Member declarations 113

D.1.2 portmaps 113

D.1.3 port interfaces . . . 114
D.1.4 parameters . . . 114
D.1.5 Template parameters . . . 114
D.2 The this variable . . . 115
D.3 Code hooks . . . 115
D.3.1 Extra member initializers . . . 115
D.3.2 Extra Headers . . . 115
D.3.3 The include items file . . . 115
D.3.4 The source file . . . 116
D.3.5 Extra constructor statements file . . . 116
D.3.6 The pre class file . . . 116
D.3.7 Extra member function declaration and definition files . . . 116
D.3.8 Extra member variables files . . . 117
D.3.9 The post class file . . . 117
D.4 Main and Type member function declarations . . . 117
D.5 Special inheritance . . . 118
D.6 MPC Driver . . . 118
D.7 Includes and Libraries . . . 118
D.8 Processor mapping . . . 118

E The make system 120
E.1 The Makefile-ScateSetup file . . . 120
E.1.1 Mandatory customization . . . 120
E.1.2 Optional customization . . . 122
E.1.3 Regression test Customization . . . 124
E.1.4 The Non configurable part . . . 125
E.2 The Make.SrcVars file . . . 127
E.3 The Makefile-Test file . . . 128

F Example of a YAPI to TTL transformation 130
F.1 TTLC++ . . . 131
F.2 TTLC . . . 132
F.3 Makefile-ScateSetup configuration . . . 132
F.4 Recapitulation . . . 133

G Symbol table initialization **134**

G.1 symtab file creation 134

References **135**

Distribution

1 Introduction

1.1 Objective

The objective of the Multiprocessor Mapping Methods (Triple-M) project is to provide a system-level design and programming environment for embedded multiprocessors. We consider the mapping of C++ based specifications of embedded systems functions onto heterogeneous multiprocessors containing hardware and software components. In practice this relates to a number of source code transformations on process networks.

Transformations should be specified by a system designer. The enforcement of these design decisions in the source code is automated by the Triple-M tools. The source code after transformation can - with or without additional manual modifications - be used as input for existing hardware and / or software compiler(s).

The objective of this document is to describe the software infrastructure developed to implement such a design and programming environment.

1.2 Informal problem statement

One common way of working - to get an implementation on a target architecture - is to gradually modify reference code of an application towards implementation. As a result there is no clear distinction between the functional part of the application, and the implementation specific details, since architecture specific decisions and functional implementation are intermixed. Another complicating factor is that reference code does not take the parallel processing capabilities of the target architecture into account. In general, reference code is highly sequential, and the amount of parallelism i.e. expressed not necessarily fits onto the architecture. This means that in order to produce an efficient implementation the reference code needs to be changed in such a way that the parallel processing capabilities of the architecture can be exploited. In summary; new parallelism has to be introduced, and the already present parallelism adjusted, all this with preservation of the original functionality of the application. Creation of new fitting parallelism is further complicated by the fact that the system designer in general does not have detailed knowledge about the application at hand.

Our approach is to use process networks as re-usable and architecture independent functional specifications, where the process networks reveal the task level parallelism of the application. Such a specification should ideally be written by an application designer. It is his / her task to capture the function without having an architecture in mind. These specifications should then be collected and maintained in one central database accessible to the Philips community at large.

The system designer then takes such a specification as a starting point. He or she will then have to make design decisions in order to obtain an efficient architecture specific implementation. Many of these design decisions are labor intensive to implement and therefore error prone. Consequently it is desirable to have a programming environment

where these decisions can be specified. The resulting transformations at the source code level should then be applied to the system in an automated fashion. This has several advantages one of which is that the design decisions become documented. Unfortunately, not all transformation and design decisions can be executed in one go therefore it should be possible to iterate over them. It should also be possible to deal with additional manual modifications. In order to guide this process close cooperation with version management tooling is a prerequisite.

1.3 YAPI

The purpose of YAPI [1, 2] (Y-chart API) is to provide a means for constructing high-level, functional executable models (i.e. functional specifications) of signal processing applications in such a way that these applications

- are reusable across projects,
- come with a run-time environment for efficient execution on a workstation or PC,
- provide input for platform dependent mapping and performance analysis.

YAPI is a C++ library with a set of rules for the modeling of signal processing applications as a process network, consisting of processes that are interconnected by directed first-in-first-out channels called FIFOs. Communication is based on the Kahn Process Network model [3] with theoretically unbounded FIFOs. Processes have a private data space that is not accessible to other processes. They communicate with their environment through input and output ports which are the formal arguments of processes. FIFOs connect output ports to input ports. They are the actual arguments of processes. Each port is connected to precisely one FIFO. Processes can *read* from their input ports and *write* to their output ports. Ports and FIFOs are typed, meaning that they can handle data of one type only. The structure of process networks, i.e., the binding of FIFOs to ports, is known at compile time and does not change at run time.

Functional specifications made with YAPI should comply to the set of rules expressed in [1, 2], that can be summarized as follows:

1. communication should be made explicit,
2. no global variables should be used,
3. minimal latency:
   - *write* as soon as data becomes available,
   - *read* as late as possible when the data is really needed.
4. a small token size facilitates reuse.

1.3.1 VYA

Besides the rules presented in the previous section coding rules for domains of applications should be defined to improve reuse. An example can be found in the VYA (video YAPI) coding rules which support the construction of reusable modules in the domain of video processing [2].

1.4 TTL

The purpose of TTL [4] (Task Transaction Level) is to provide an abstract interface that supports the integration of hardware / software tasks on top of platform infrastructures. Tasks are modules that perform specific functions, and are the logical entities for reuse within and across organizations. The goal of a standard TTL interface is that it:

- facilitates development of tasks by offering an abstract, easy-to-use, interface to high level platform services,
- improves re-usability and integration of tasks,
- facilitates evolution of platform infrastructures by hiding platform infrastructure implementation aspects from the tasks,
- enables the development of standard design technology for the integration of tasks,
- allow cheap and efficient implementations of platform infrastructures,
- support a broad range of platform infrastructures and applications.

As stated in section 1.3, YAPI is meant as a means for constructing high-level functional executable models without any reference to a target architecture. TTL on the other hand is meant for the integration of hardware / software tasks on top of platform infrastructures. A natural way of working is to start with a functional specification in YAPI, and to use TTL for refinement towards an efficient implementation.

1.5 Solution approach

A high level view of the Triple-M tools is shown in Figure 1. There are basically two sets of input files. One set describes the function to be implemented which is captured in a process network structure. The other set contains the design decisions made by the system architect to efficiently implement the function. The *SCATE* tool [5], which is short for Source Code Analysis and Transformation Environment, applies design decisions on the process network and produces a set of output files. These files serve as input for either additional manual modifications, another iteration cycle using the Triple-M tools, or existing hardware and / or software compilers.

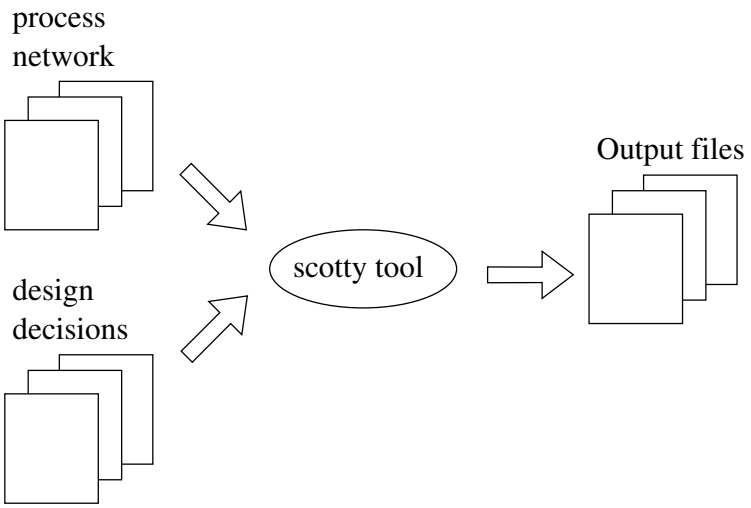

Figure 1: User view on the *SCATE* tool.

The *SCATE* tool is able to extract the process networks from C++ source code - written in YAPI or TTL type syntax -. Allowing the system architect to specify transformations on process networks. As mentioned these transformations are steered by the system architect, i.e. the tool itself cannot invent transformations. The output files are automatically restructured to fit one consistent software style. By this we mean for example that each process and each network has its own directory with separate header and source files. Furthermore, an Imakefile[6] is generated in each subdirectory such that the output can be readily compiled.

Figure 2 shows that iterations are supported at different levels of abstraction: (1) one can transform processes at the YAPI level, (2) one can transform tasks at the TTL level, and of course (3) at some point in time one has to transform YAPI processes into TTL tasks.

As shown it is not possible to go back from TTL tasks to YAPI processes[1]. Furthermore, at the time of writing some support for network structure transformations to TTL C source code is incorporated. However, for those systems it is not possible to perform type (2) iterations, since this would require the extraction of the process network from C source code.

[1] Strictly speaking this is not true, since the tools allow one to convert TTL to YAPI - when communication is based on (single cast) interface type CB only -. However, this is strongly discouraged, since a TTL model most likely contains design decisions making it unfit as a pure functional model.

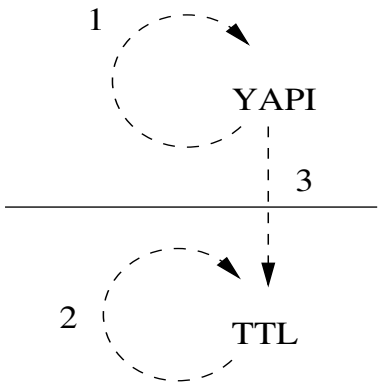

Figure 2: The *SCATE* tool supports iterations at multiple levels of abstraction as indicated by the circular dashed lines: (1) at the YAPI level, (2) at the TTL level, besides (3) it is also possible to transform YAPI into TTL.

1.6 Intended audience

This document is most relevant for *SCATE* tool developers, since the focus is on the software infrastructure. Its current possibilities and limitations are described.

This document is also of interest for anyone who wants to gain (detailed) insight in the current set of transformations. Moreover, it could also be of interest for application designers, and system designers. Potentially, the techniques described in this document will allow them to make automated (system level) design choices. Even more important it may inspire them to come up with additional transformations worth automating.

Users of the *SCATE* tool, i.e. application designers and system designers can restrict themselves to chapters 2, 6– 10.

1.7 Outline of document

In chapter 2 an overview is presented of the *SCATE* tool. The software syntheses tool called Compiler-Compiler-Compiler-Popp-Ondrej (*C3PO*) [7] is introduced in chapter 3. This tool has been used to generate the infrastructure of the *SCATE* tool. In chapter 4 it is explained in detail how to build functionality on top of such a *C3PO* generated infrastructure. The C++ front end created is discussed in chapter 5. A transformation language - to instruct the *SCATE* tool to perform a certain transformation - is discussed in chapter 6. The recognition of YAPI and TTL process networks from source code is discussed in chapter 7. Transformations are discussed in chapter 8. In chapter 9 various forms of code generation are discussed. Regression testing is described in chapter 10. Finally, we end with conclusions and future work in chapter 11.

2 Overview

2.1 The Triple-M tools

In this chapter we show multiple views on the *Triple-M* tools. One high level component view from a tool designer perspective is shown in Figure 3. As can be seen there are three components at stake. Together they form what we call the *Triple-M* tools. First there is the tool called *SCATE* [5] which does the parsing of input files, applies transformations on the internal representation, and finally writes out a set of intermediate files containing possibly transformed source code. Next, there is the tool called *MPC* [8, 9] which takes the output of *SCATE* as input[2], and produces the output requested by the user. The third component (Software automation tools) - not visible to the user - generates the software infrastructure for *SCATE* and *MPC*. On top of this infrastructure additional functionality can be built.

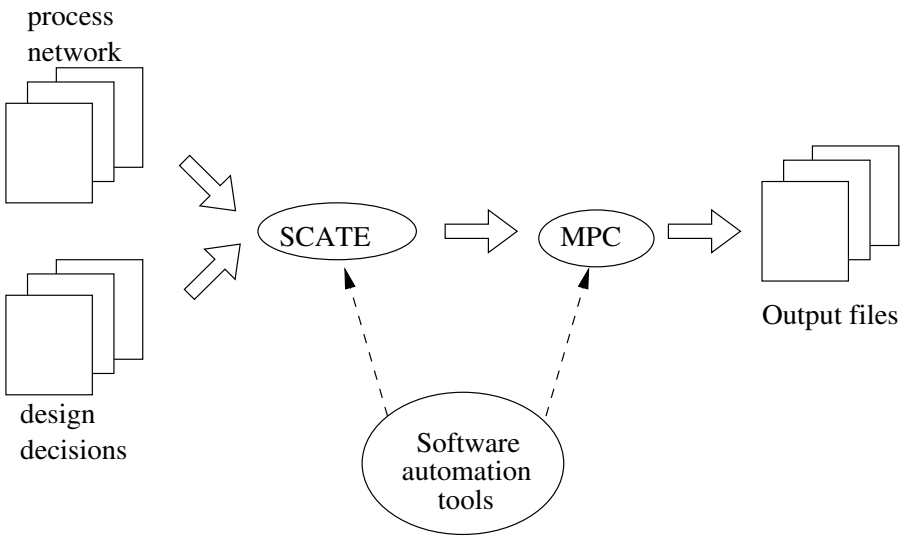

Figure 3: High level component view from a tool designer perspective.

Another view on the *Triple-M* tools is shown in Figure 4. This view concentrates on the flow of data through the tools from an user perspective.

All C++ source code is first run through the gcc preprocessor. This has consequences for comments and macros in the source code. The former is simply removed[3], while the latter is expanded. The output of this preprocessing step is a set of '*.cpp' files. These preprocessed files combined with the user defined input files are fed into the *SCATE* tool.

The *SCATE* tool internally separates the functional code, from the network description[4]

[2]Can only be done when *MPC* is used in back-end mode. This is an extension made after the publication of Technical Note [8, 9]

[3]In future versions we would like to prevent this from happening.

[4]Description of the structural interconnect of the application, such as ports, channels, process and net-

itself. This leads to two sets of output files as shown. *MPC* combines these two sets and produces output files where the semantics and the network structure are combined again. The extraction of the network structure allows to easily switch between different output formats. In practice this is done by selecting a different driver, e.g. *MPC* is equipped with a C++ (YAPI / TTL), C (TTL), and a VHDL network driver.

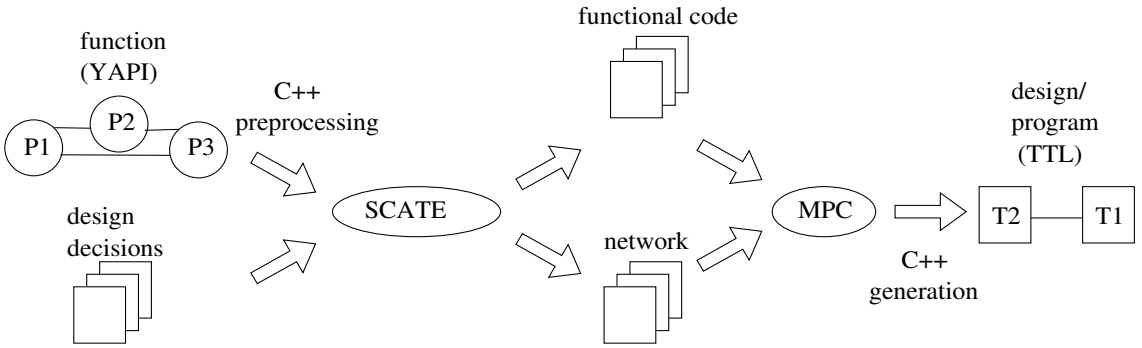

Figure 4: High level view on the flow of data through the *Triple-M* tools.

2.2 The C3PO meta level

The next view presented, shown in Figure 5, is a detailed component view from a *SCATE* tool designer perspective. Here it is shown that the software automation tools consists of two tools called *C3PO* [7][5] and *ANTLR* [10], respectively.

> ANother Tool for Language Recognition (*ANTLR*), (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, or C++ actions. *ANTLR* provides excellent support for tree construction, tree walking, and translation. There are currently way over 10,000 *ANTLR* source downloads a month.

C3PO is a tool that is able to generate (an) input file(s) (*.g) for *ANTLR*, based on its own input grammar file(s) (*.G), as is shown in Figure 5. The advantage is that *C3PO* can do this for a number of other compiler compilers as well. Moreover, it is capable of generating datatypes. These datatypes are initialized by the *ANTLR* generated parser. Besides also datatypes that are not initialized by a parser can be generated. This is symbolized by the dashed dotted arrow, for example from *C3PO* to *MPC*. Access functions for the members of the datatype are generated enabling an easy integration of the software structure.

work entities.

[5]This is a proprietary tool developed by one of the authors, Ondrej Popp.

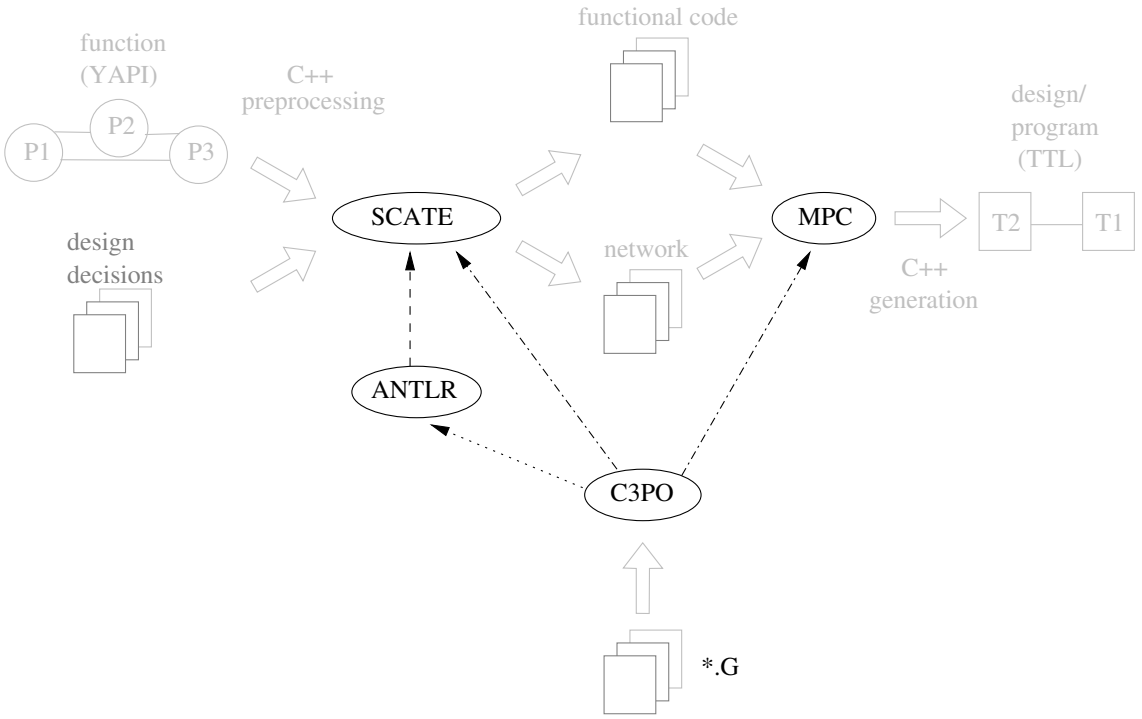

Figure 5: Component view from a *SCATE* tool designer perspective. The arrows represent datatype generation (dashed dotted line), the generation of input files (*.g) for ANTLR (dotted line), and parser generation by ANTLR (dashed).

In chapter 3 the secrets of software synthesis with *C3PO* are explained. With this approach, a large part of the software infrastructure is generated for the Triple-M tools. On top of which additional functionality is added, see chapter 4. This is done by extending the automatically generated classes with custom member functions. Maintenance issues like how to deal with generated source code and manual code within the same source file are explained as well.

2.3 SCATE components

The *SCATE* tool can (internally) be further subdivided into several components as shown in Figure 6. The arrows between the components indicate the flow of information through the tools.

The set of input files is parsed to build an abstract syntax tree (AST). This internal representation can be traversed - while enriching and / or transforming the AST - using visitors. After this step the AST can be printed in output files which can on its turn be used as input for other tools as shown.

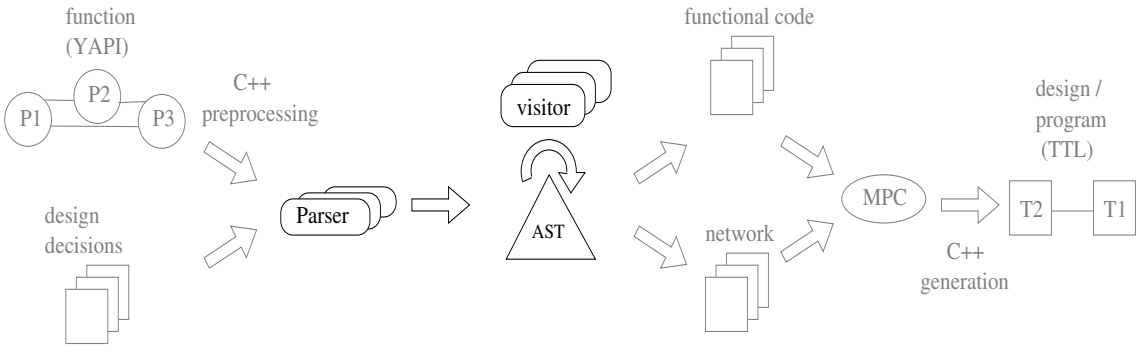

Figure 6: Component view from a detailed user perspective.

The C++ files are first run through a preprocessor thereby expanding macros and removing comments from the source code. The C++ parser frontend accepts the preprocessed files as input. This frontend will be discussed in chapter 5. It uses a fine grained C / C++ grammar that allows one to do a very detailed analysis of the source code. An historic overview of this grammar is presented as well. All application files are compiled in one go which can only be done when a particular strategy is used. The advantages of this approach and the algorithm used in this strategy are also explained.

The language with which transformations can be specified will be discussed in chapter 6. This language allows the user to configure the API of the YAPI / TTL process network. By this one directly influences the recognition of the YAPI/TTL API. This built in flexibility hopefully means that late changes in the TTL specification can be dealt with. Besides configuration the user can also express design decisions enforced on the process network. It allows the user e.g. to flatten selected parts of the hierarchy, to remove fork processes, to change interface types etc.

Chapter 7 describes the implementation of the YAPI and TTL API recognition. This allows one to iterate over both kinds of network processes and to convert YAPI into TTL.

The implementation of several transformations is described in chapter 8. First, the concept of an engine is described which enables (iterative) alternating structural transformations and functional transformations whereby information about the transformation is communicated. Hereafter various structural transformations and functional transformations are described.

In chapter 9 the code generation back-end of the *SCATE* tool is described. For this *MPC* [8, 9] is used in back-end mode to glue the process network and the semantics part together again. This decoupling has the advantage that the output format is not fixed but can be varied by selecting a different driver. The process network structure can for example be written in various output formats like VHDL and C / C++. VHDL generation is out of scope of this project however. For this project, a heterogeneous driver is being used that generates either YAPI (C++), TTL (C++), or TLL (C) code.

3 Software synthesis with C3PO

This Chapter gives an overview of automatic software generation with *C3PO*. With *C3PO*, large parts of the *SCATE* and MPC tool are automatically generated. This is what we call the *Meta level*. A meta tool (*C3PO*), is used to generate other tools such as *SCATE* and MPC. However, it is not possible to generate these tools completely. We call that part that can automatically be generated the *infrastructure*. On top of this infrastructure hand written code needs to be added to complete the tool. In this chapter, we will deal with the automatic generation of the infrastructure. Chapter 4 shows how hand written code can be added on top of this infrastructure.

3.1 How to build a solid infrastructure

A good technique to build a solid infrastructure is to bridge the difference in which humans and computers process information. While humans think in *symbols* that come natural to the human mind, the computer uses *numbers*. This means that by nature there is a gap between the representation entities the human mind uses, and the ones a computer program uses.

While a number is a natural representation entity for a computer program, numbers are really a small subset of our world, therefore they hardly ever exist on their own. This is so because extra information is needed to specify the *meaning* of a number. For example, the number 65 on a number plate of a car means something different than the same number on a house in a street. This shows that numbers in our world are mostly *embedded*, and this embedding provides additional information about their meaning.

Bad programs are programs that do not take this embedding into account. Such programs consist mainly of a large number of unstructured variables. The meaning of these variables is not explicit and needs to be 'remembered'. While this style may be acceptable for tiny to small programs, for larger programs this leads to unreadable, unmaintainable or even disastrous code. To avoid this, *datatypes* are introduced to provide the meaningful embedding for variables. With this technique, a *one to one mapping* is realized between symbols and datatypes.

A *structure* is built to reflect the *consists of* and the *is a kind of* relations between symbols. The former relation is modeled by *composition*, while the latter is modeled by means of *inheritance*. Next, *standard functionality* is created to perform common manipulations on the datatypes. For example, object creation, member access and dump functions. This functionality is called the *API* or the *application interface*. The sum of all the datatypes together with their standard functionality, is called the *infrastructure* of the program. Such an infrastructure is easy to read, easy to manage, and extendable even with high complexity because it comes natural to the human mind.

A computer program is thus built by designing and implementing a suitable infrastructure, and by *layering* the actual functionality of the program on top of it. While this brings great advantages, there is also a problem: to represent a large number of symbols we also need a large number of datatypes, which is a lot of work. A quick and dirty way to minimize

the implementation work, is to use the same datatype for different symbols. This breaks the one to one mapping and is confusing.

A much better solution is to automate the implementation of a software infrastructure which eliminates effectively the above problem. *C3PO*, which stands for *Compiler Compiler Compiler Popp Ondrej* provides such a solution. With *C3PO*, any infrastructure can be built fast and conveniently from a set of attributed grammar rules in EBNF (Extended Backus-Naur Form) notation.

3.2 C3PO components

C3PO consists of the following components,

datatype synthesis engine This engine analyzes the *information content* of a grammar rule, and derives a suitable representation for a datatype. The datatype will have the same name as the grammar rule, and its members will reflect the right hand side of the rule. For each grammar rule, a corresponding datatype is derived. The details are described in section 3.3.

Language drivers These drivers create source code from the datatype representation in the following languages,

- C
- C++
- Java

The source code includes both the definition of the datatype and its API. In section 3.4 the details of C++ code generation are explained. Details on generation of other languages is out of scope of this report.

Compiler compiler drivers These drivers generate input for a compiler compiler. Table 1 lists the various compiler compiler types with their corresponding output languages.

Compiler Compiler	Language	Additional info
lex/yacc	C	http://dinosaur.compilertools.net
flex/bison	C	http://www.gnu.org/software/bison/ http://www.gnu.org/software/flex/
antlr	C++	http://www.antlr.org , can also generate java but this is currently not supported by C3PO
javacc	java	http://javacc.dev.java.net/

Table 1: Overview of C3PO supported compiler compilers

The drivers pass down the grammar rules in a suitable format to the compiler compiler, together with action statements that create and initialize a datatype instance for each grammar rule. The compiler compiler creates then a parser from this. Such a parser is able to parse an input that corresponds to the given grammar, and stores the encountered information in the datatype instances.

Make system and directory structuring driver This driver creates a directory structure for the synthesized datatypes, their API, and the parsers. It generates also an Imake [6] based make system to compile everything into an executable.

3.3 Datatype synthesis

Datatype synthesis is performed from a set of *rules*. A rule consists of a left hand side, a colon (:), a right hand side, and ends with a semi colon (;). The left hand side specifies the name of the rule which will also be the name of the datatype. The right hand side *defines* the rule by means of *elements*. An element specifies a *member variable* of the datatype. The element notation consists of a left hand side, an arrow ($->$), and a right hand side. The left hand side refers to a rule, or datatype and specifies the *type* of the member variable. The right hand side specifies its *name*.

A rule can further be a *parser rule* or a *synthesis rule*. A parser rule is a rule that will also be part of the compiler compiler input. A synthesis rule will only be used for datatype synthesis. This distinction is expressed by embedding the rule within the $<>$ characters. This distinction is also available at the element level. Elements that will not be part of the compiler compiler input must have their left hand side embedded within parentheses (). This will thus result in an extra member variable that will not be initialized by the resulting parser.

Parser rules also contain additional elements that are not associated to any datatype. For example, syntactic separators like a comma, or embedding constructs like a set of parentheses, or a token. A token is generally used to define a *keyword*. Here, a regular expression pattern is associated with a token name and optionally with a built in type.

The following is an example in which three datatypes are synthesized. The *line*, *point* and *rect* datatype. The *line* datatype consist of three member variables *begin*, *end* and *middle* of type *point*. The *begin* and *end* variables are initialized by the parser, while the *middle* member variable is not. The *point* datatype contains the member variables *x* and *y* of type *int*. This type is derived form the INT token with the regular expression '[0-9]+'. Further, the parser rule *point* embeds the INT tokens within parentheses and separates them by a comma. So, for example the construct *(3,4)* initializes a *point* datatype and the construct *(3,4) (5,6)* initializes a *line* datatype. The *rect* datatype contains the members *upperLeft* and *lowerRight* of type *int*. This datatype is not initialized by the parser.

```
line  : point->begin
        point->end
        (point)->middle // extra member variable not
                        // initialized by the parser
      ;

point : '(' INT->x ',' INT->y ')'
      ;

// synthesis rule
<rect : point->upperLeft
        point->lowerRight
>;

int INT : '[0-9]+'; // token rule
```

3.4 C++ code generation

This section gives an overview of the C++ code that is generated by C3PO from the datatype representation as introduced in the previous section. C++ code is generated by the C++ language driver as already introduced in section 3.2 The whole infrastructure is generated from a set of grammar files, found at the root of the directory tree. A grammar file has the *.G* suffix. There is one main grammar file, and several sub grammar files. The code generated from the main grammar file resides at the root of the directory tree of the software infrastructure, in predefined subdirectories. The names of these predefined directories are fixed within *C3PO*. The code from a sub grammar file is placed in a separate subdirectory. The name of this subdirectory is not fixed and is specified within the sub grammar file itself. This is done by embedding all the rules of the sub grammar within a *name* { *(rule)+* } construct. The *name* specifies the name of the subdirectory. For example, the construct *YapiRep* { *(rule)+* } will create a *YapiRep* subdirectory, and puts the C++ code generated from the corresponding rules inside the predefined directories, within the *YapiRep* subdirectory. The subdirectory construct can be nested, so the entire set of rules can be divided over an unlimited set of subdirectories.

In order to clarify the above we present an example script below. The script introduces the toplevel rules *r1*, *..*, *rn*, and the sub-grammars *SG1*, *SG2* and *SG3* with corresponding rules. Sub-grammar *SG2* is a sibling of *SG1*, and *SG3* is nested within *SG2*. For clarity reasons the right hand side of the rules is omitted. Figure 7 shows the generated directory structure. Each directory contains the *base* and *src* subdirectories where the code for respectively the synthesized and customizable classes is stored. The *parser* subdirectory contains the code for the *ANTLR* based parser. This directory is only present if the corresponding (sub) grammar contains parser rules. In addition, the top directory contains also the *util* subdirectory. This directory is also automatically generated by *C3PO* and contains generic code for command line parsing, file handling, and text manipulation.

```
r1 : . . . ;
. . .
rn : . . . ;
SG1
{
   sg1_r1 : . . .;
   . . .
   sg1_rn : . . .;
}

SG2
{
   sg2_r1 : . . .;
   . . .
   sg2_rn : . . .;

   SG3
   {
      sg3_r1 : . . .;
      . . .
      sg3_rn : . . .;
   }
}
```

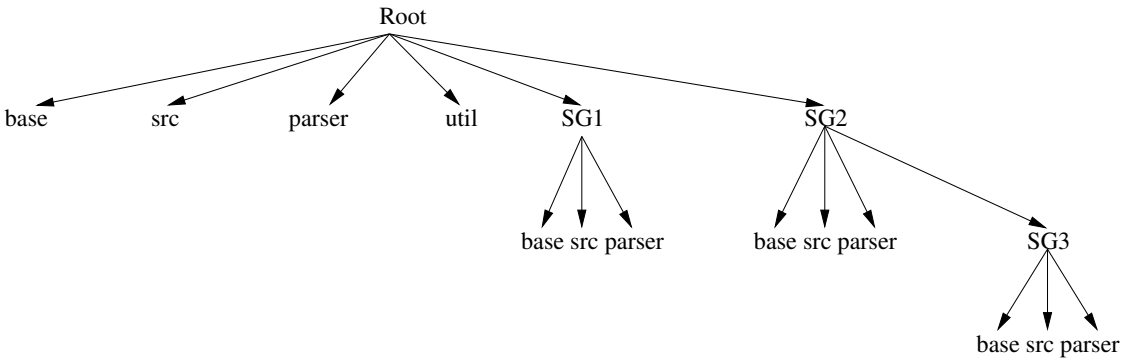

Figure 7: The generated directory structure.

3.4.1 The Generate script

Code generation is further simplified by means of the *Generate* script. This script runs *C3PO* with all the appropriate options, and creates the entire infrastructure. Thus the

whole infrastructure is maintained / extended by editing the various grammar files, and running the *Generate* script.

3.4.2 Base classes

Generated code that cannot be customized by the user is synthesized in the form of base classes and placed inside the predefined *base* subdirectory. For each rule *r*, a *rBase* class is generated. Each class consists of a set of member variables and member functions. A member function applies either to the whole class, or to a specific member of the class. Member functions that apply to the whole class are summarized in appendix A.1.

Scalar members Scalar members are synthesized from the

```
T->n
```

construct. The type of the member is specified by *T*, its name by *n*. Member functions for these members are summarized in appendix A.2.

Collection members Collection members are synthesized from the

```
(T->n)+
```

or

```
(T->n)*
```

constructs. The member types are mapped onto the C++ standard template library (STL) container classes, reference [11] page 117–173. The names of the members will be specified by *n*. A widely used name is *array*. Although all STL container types are supported, the following two are widely used.

- Sequential Container Members. These are mapped on the STL

  ```
  vector<T>
  ```

  container.

- Hashed Associative Container Members. The synthesis construct for these members is more elaborate in order to supply additional parameters for the STL container type.

  ```
  (T->n / hmap[(Tkey) : (compareObjName) ])+
  ```

  Tkey is the type of the key of the hash map, and *compareObjName* the name of the user specified function object that is to be used for comparisons.

For example,

```
(T->n / hmap[(char *) : (strCompare) ])+
```

will synthesize a

```
hash_map<const char *, T, hash<const char*>, strCompare>
```

container. To satisfy the STL key datatype requirements, each *Tkey* type that is not a read-only type will be converted to a *const* type.

Appendix A.3 summarizes the member functions for the collection members.

Enumerated types Enumerated types, like the C *enum* type represent a discrete set of values. *C3PO* generates enum types from *enum rules*. An enum rule is a rule prefixed with the *enum* keyword and with only scalar elements at the right hand side. The scalar elements are interpreted here as one of the possible values for the enum type. These elements are also the *terminals* or *tokens* of the grammar. For example, the rule:

```
enum color : RED | GREEN | BLUE;
```

will synthesize the *color* enumerated type that can take the values *RED*, *GREEN* and *BLUE*. The C++ code generation will embed a C style enum type within a C++ class. The advantage of this approach over the plain C style enum type is that different enum types can have the same values, which is impossible in C. For example, the:

```
enum trafficLight : GREEN | ORANGE | RED;
```

type can coexist with the *color* type. Appendix A.5 summarizes the member functions for the enumerated types.

3.4.3 Customizable classes

Customizable classes are intended to hold custom code. These classes are subclasses of the base classes and are placed within the *src* subdirectory. For each rule *r*, a class with the same name is generated which is a direct sub class of the *rBase* type. The files in the *src* directory are overwrite protected. After their generation, the user can then add hand written code to these files in form of new member functions. To make these new functions visible in the datatype hierarchy, each base class is generated in such a way that the types of its members are the subclass types.

3.4.4 Robustness

Most crashes in programs are caused by invalid pointers. However, most invalid pointers are just *NULL* pointers. Usually this is not really an erroneous situation, because this means that a certain member of a datatype is not valid. Usually, this situation is represented by a NULL pointer. So, the cause for most crashes can be found with member access code, and the fact that the programmer has forgotten to add a proper validity check. This situation can be greatly improved by adding proper validity checks to the member access API automatically. *C3PO* generates these validity checks within the relevant member functions of the base classes, for example within the *getMemberName()* functions. When these member functions are consistently used within the user defined functionality, this results in very robust code that (almost) never crashes. This gives a large speedup during development.

The *C3PO* generated validity layer is built around *Null Objects*. A Null object is a static instance of a class *T* that is to be used to represent non validity. In this way, no NULL pointers will be present in the code anymore to represent non validity. Appendix A.6 summarizes the member functions of the validity layer.

Input generation for ANTLR Input generation for *ANTLR* [10], is based on the generated API of the base classes. For each rule *r*, a *parseR* description is generated that is to return an initialized instance of the corresponding *r* datatype. Appendix A.7 gives an example of such a description. *ANTLR* converts this description into the actual *parseR* member function of the generated parser class [10].

Parse images With a large number of rules and their accompanying datatypes, it is not always easy to know in which instance of which datatype a certain piece of the input has been stored. For this reason every synthesized datatype contains also the *parseInfo* member of type *parseinfo*. This member is initialized by the *initParseInfo* and *finishParseInfo* member functions of the parser class. As can be seen in the listing of appendix A.7, the *initParseInfo* function is called just before a rule is parsed, and the *finishParseInfo* is called right after the rule has been parsed.

The *parseinfo* datatype is shown in appendix A.8. The information in this datatype can be used in error reports. The exact piece of input that corresponds with its associated rule, is called the *parse image* and is stored in the *parseString* member of the *parseinfo* datatype. This can span multiple files. When the *dump* member function of the top level datatype is called, the printed parse images clearly shows how the input has been decomposed. This is very helpful for debugging purposes and during the design of Visitor patterns.We elaborate further on visitor patterns in section 4.2.1.

Root parse functions Because *ANTLR* generates a recursive descent parser, parsing can begin anywhere. Normally, *C3PO* derives automatically the top level datatype of a grammar that will be the root of the parsing hierarchy. When another - or more than

one - rule is desired as the root, this can be specified by the *-root* command line option. Appendix A.9 shows the synthesized root parse functions.

3.4.5 The DepthFirstVisitor class

The *C3PO* generated *DepthFirstVisitor* class is the superclass for all visitor design patterns. See references [12] page 331–344, [13] page 944–952 and section 4.2.1 of this document. The *DepthFirstVisitor* class contains *visit* functions for each generated datatype, and each datatype contains the corresponding *accept* function.

4 Adding functionality on top of the infrastructure

Functionality can be added by extending the classes - generated by *C3PO* - in the *src* directories with custom member functions. This chapter discusses a number of software engineering techniques used with the *SCATE* infrastructure.

4.1 Sequential container access

A very frequently needed functionality is to access all elements in a container in sequential order. For sequential container access the loop variable can be implicit. This is supported by the *getElements()* function. (appendix A.3). The function returns a java style *Enumeration* object (appendix A.4) that supports sequencing through the container. Sequencing in the forward direction, so from the beginning to the end, is performed by means of the *hasNext()* and *getNext()* member functions of the Enumeration type. For each STL container category, there is a matching Enumeration type. The type of this object is predefined in the synthesized datatype for each of its collection members. The type name is the name of the collection member postfixed with *E*. The following shows a typical example for a datatype *T*, its collection member *array* with element type *Tel **.

```
void T::seqAccess()
{
   arrayE e = getElements();
   while(e.hasNext())
   {
       Tel *el = e.getNext();
       // do something with el
   }
}
```

The *getNext()* function returns the next element from the container, and advances the hidden loop variable. So, when the loop body references an element more than once, the element must be stored in a local variable. Alternatively, the *getCurrent()* function can be used to access the element previously returned by *getNext()*. The *getCurrent()* function does not increment the loop variable.

For associative containers that have the STL *pair* element type, the functions *getNextValue()*, and *getNextKey()* must be used instead of the *getNext()* function to access the element and its key respectively. Analogously, the *getValue()* and *getKey()* functions must be used instead of the *getCurrent()* function.

Although not that frequently used, the Enumeration also supports sequencing in the backward direction. This is performed by means of the *setEnd()*, *hasPrev()*, and *getPrev()* member functions. However, these functions are only available if the STL container supports this. The following is a typical example,

```
void T::seqAccessBackward()
{
   arrayE e = getElements();
   e.setEnd();
   while(e.hasPrev())
   {
       Tel *el = e.getPrev();
       // do something with el
   }
}
```

Analogously, the Enumeration type for an associative container provides also the *getPrevValue()* and *getPrevKey()* member functions.

4.2 Generic AST transformations

4.2.1 Visitor patterns

A general method for converting AST's is the use of the Visitor design pattern as Described in references [12] page 331–344, and [13] page 944–952. A visit pattern is a recipe how the AST nodes should be traversed to collect the relevant data to initialize the output AST. This information can be derived from the input AST.

DepthFirstVisitor The *C3PO* generated *DepthFirstVisitor* class contains *visit* functions for each generated datatype, and each datatype contains the corresponding *accept* function. Two variants for returning the output AST are supported. The output AST can either be specified as a member of the Visitor subclass, in which case the standard accept function can be used, or, the output AST can be a member of the *VisitorAttributes* object in which case the extended accept function must be used that passes down the *VisitorAttributes object.*

```
void accept(Visitor* const v)
void accept(Visitor* const v, VisitorAttributes* const Attr)
```

The actual initialization of the output AST is performed by the visit functions of the visitor subclass. This class can also be generated automatically, after which the various visit functions need to be implemented by hand. In practice, it has proven to be more convenient to use the variant with the standard *accept* function, so the one that has the output AST as a member of the visitor subclass. The following is an example of such a sub

class. Here, the *yapiVisitor* class is synthesized as a direct subclass of the *DepthFirstVisitor*. The *yapiRep* datatype, assumed to be synthesized elsewhere, represents the output AST. Eventually more members can be added as well. The *[new]* directive means that the member will be initialized by means of an appropriate constructor instead of the null object.

```
<yapiVisitor extends DepthFirstVisitor :
 (yapiRep)->rep = [new] // the output AST
 // ... more members can be added here
>;
```

PathVisitor The visit functions of the visitor sub class have access to one unique datatype *T* by means of its input parameter. This parameter will either be a certain *context*, a part of this context, or an unrelated part for that context. Visiting a certain context is necessary for the creation of the corresponding AST node of the output AST. Visiting the context related part is necessary for the initialization of the AST node members. Unfortunately, sometimes it is difficult to have access to both the context *and* the context parts in one visit function. When the context is visited, it may be difficult to access its parts especially when these lay much deeper in the AST tree. On the other hand, if we visit the deeper located parts, it may be difficult to get the right context for them. This problem is greatly simplified with the introduction of the *PathVisitor*. The *PathVisitor* is a special visitor intended to visit only the deeper located parts. For each datatype that is to be visited by this visitor, there is a corresponding collection member. Around this collection member, the following *get* function is implemented,

```
template<class Tc>
Tp * getTp(Tc *context, const unsigned int position = 0)
```

The template type *Tc*, is the context where the get function should start from. It is implemented as a template parameter to make sure that visiting can start at any node of the input AST. The get function returns the *Tp* object from the depth of context *context*. Usually, this will only be one unique instance of the Tp type. In this case the Tp instance lies on a single unique path from the context Tc, hence the name *PathVisitor*. In more complex situations, more instances of the Tp datatype are found. The right instance must now be selected by means of the *position* parameter. While this is also uniquely defined, this case should be avoided as much as possible because it poses a maintenance problem. When the input grammar is changed and this change reflects the position of that particular Tp instance in the AST tree, the position parameter needs to be adjusted as well.

The *PathVisitor* itself can further be a member of the other Visitor subclass. This can also be performed by *C3PO*. For example, by extending the original *yapiVisitor* description as follows.

```
<yapiVisitor extends DepthFirstVisitor :
 (yapiRep)->rep = [new] // the output AST
```

```
 (PathVisitor)->pathVisit = [new]
 // ... more members can be added
>;
```

The AST output node creation works as follows:
First, a certain context is visited by a *yapiVisitor::visit* function. Then, from that visit function, the *PathVisitor* is started that will *dig up* the context related parts after which these parts can be used to initialize the AST node. The following is a typical example of this,

```
yapiVisitor::visit(Tc *context)
{
  SomeNode *n = new SomeNode();

  Tp1 *part1 = getPathVisit()->getTp1(context);
  Tp2 *part2 = getPathVisit()->getTp2(context);

  n->setPart1(part1);
  n->setPart2(part2);

  getRep()->setSomeNode(n);
}
```

Design of visit patterns To recapitulate, a visit pattern is a recipe how the AST nodes should be traversed to collect the relevant data to initialize the output AST. This information can be derived from the input AST. Parse images, introduced in section 3.4.4, are very helpful for tracking down this kind of information. This information is then converted to a set of visit functions. For each type of AST node that has been identified, a corresponding *visit* function of the visitor subclass is created. The call to the top level visit function initializes the output AST, which can subsequently be returned.

4.2.2 The ParseFromString functions

The *parseFromString* function, appendix A.9, provides another way to convert the AST. Here, the input AST is decomposed into a string, and the *parseFromString* function creates the output AST. However, when this output AST is not the top level rule of the grammar, it may be necessary to append proper termination tokens to the end of the string to satisfy *ANTLR*'s lookahead sets.

4.3 Maintenance

4.3.1 CVS based patching

Because the customizable classes are initially created by *C3PO* and then manually extended, there is a maintenance problem when a modification of the *C3PO* input reflects a change in the generated part of the customizable class. Ideally, only the synthesized part of that class should be regenerated while the hand written part is preserved. Because both parts reside in the same file, the file cannot just be regenerated by *C3PO*. Updating the generated part by hand introduces a maintenance problem and is not desirable either. This problem has been solved by a cvs based patching method. This works as follows: besides the initial generation of the customizable file which is never overwritten, *C3PO* also generates a hidden file with the same name in the *.cvsPathDir* directory inside the *src* directory. This file is not overwrite protected, and will thus be regenerated every time *C3PO* is run. Assuming further, that the file is cvs managed and kept up to date, the difference between the new local state of the file, and the previous state in the cvs repository reflects exactly the effect of the modified *C3PO* input. This information is further used to construct a patch file, and this patch file is then applied to the overwrite protected file. In this way the generated part of the file is kept up to date, while the manual additions are preserved. Finally, the new local version of the hidden file is checked in to reflect the new state of the file. This whole process is simplified by two shell scripts. The *cvsPatch* script creates and applies all the patches in all src directories, and the *cvsCiPatch* script checks in all the hidden files.

4.3.2 The mkHeader script

Another maintenance problem is the problem of keeping the member function signature of the function interface consistent with the signature of its implementation. The C++ design philosophy advocates a clean separation of the interface of a member function from its implementation and advises to first design the interface in the class header file, and then its implementation in the source file. Unfortunately, the C++ syntax does not really reflect this approach in a clean way, because the function signature has to be repeated for both the interface and the implementation. In practice, because most of the design time is spent on the implementation of a function, rather than on its interface, any modification of the member function signature in the source file must also be repeated in the header file. A frequently made mistake is to forget to update the header file, which only wastes design time. This problem is solved by the *mkHeader* script. This script regenerates the header file from the source file, and keps thus the function signatures in sync. However, because the C++ access specifiers *public, protected* and *private* are illegal for the function implementation, this information cannot be regenerated form the source file. For this reason, the pseudo keywords *pubfun, protfun* and *prvfun* have been introduced that are supposed to be placed in front of the function signature in the source file. In addition, the pseudo keywords *stafun* and *virtfun* are also available for *static* and *virtual* functions respectively. To prevent C++ parse errors, the pseudo keywords are #defined to nothing in the *std_macro.h* header file.

5 The C++ parser frontend

5.1 History

Building a full fledged C++ parser is a real challenge. The need for such a parser arose quite a few years back during the CPA project. In this project, C functional descriptions of video algorithms had to be converted manually - due to lack of a behavioral synthesizer - into VHDL descriptions of functionally equivalent video processors. This was extremely hard to do, therefore the desire arose to try to do this at least semi automatically. Such a tool was supposed to do source code analysis, visualization, conversion to a Signal Flow Graph (SFG), transformations on the SFG, and VHDL code generation. A C parser was needed as a frontend for this tool, but to be prepared for the future, it was decided to go for C++. Unfortunately, the problem of converting C to VHDL was not recognized sufficiently so there were no possibilities at that time to pursue this activity at work. Instead of abandoning the whole idea, one of the authors (Ondrej Popp) decided to pursue this idea in his free time at home. Out of this activity *C3PO* was born with the intent to generate the infrastructure for a conversion tool.

g++ The first approach was to try to re-use the open source g++ compiler. But the g++ implementation at the time[6] did not contain a recognizable grammar, because the whole parser was completely custom made. This made it very hard to derive a suitable grammar input for *C3PO*, therefore this idea was abandoned.

The Roskind C++ grammar The second approach was to use the Roskind C++ grammar [14]. This grammar was adapted into *C3PO* input from which a lex/yacc based C++ parser implemented in C was synthesized. Around this parser, a first implementation of a source code transformation infrastructure arose. Unfortunately, because of the limited lookahead and back tracking capabilities of yacc, this implementation could not be made conflict free. This meant that many C++ constructs could not be parsed sufficiently. Only a bit more than the C subset could really be parsed.

Java / JavaCC To cope with the above problem, another implementation was tried based on the open source java based Javacc compiler compiler. This compiler compiler comes with a C++ grammar, which makes it possible to create a C++ parser implemented in java. So, java/javaCC support was built into *C3PO* and a java based C++ code analysis infrastructure was synthesized. Because of the superior lookahead and back tracking capabilities of this parser, this implementation was free of conflicts. By that time, the *C3PO* java synthesis technology, and this infrastructure was mature enough to be useful, and interest came from Paul van Loon to use this infrastructure for his work in the Cosy project. During this project, the *C3PO* generated visitor patterns have been added which was Paul's idea. He also extended the C++ grammar with missing constructs, and sug-

[6] 1996, version 2.7.x

gested to add the parse images to simplify the design of visitor patterns. The authors wish to express their gratitude for Paul's contributions to the *C3PO* technology.

C++ / ANTLR In the Triple-M project, this technology was developed further. The starting point was Paul's C++ grammar from the Cosy project. Since then the java implementation has been replaced by a C++ / *ANTLR* implementation. The reason for doing this was that no graphical output was anticipated in this project, so the necessity of using java was not that evident, and moving to C++ would eliminate the need for a java runtime. Besides, it gave the opportunity to further develop the C++ code generation and *ANTLR* support in *C3PO*. The most important new features are the generation of parse images, Null objects, and support for sub parser multiplexing built around *ANTLR*'s token stream multiplexing functionality. The latter will be discussed further in section 5.3. The version of *ANTLR* that is used with *C3PO* is version 2.7.1. This version also needs a patch to enable new features built into *ANTLR*. These consist mainly of compatibility fixes with *C3PO* and gcc 3.x, enhanced debugging / trace capabilities, and the already mentioned support for sub parser multiplexing. The patch is located in the *ANTLR / patches* subdirectory of the *SCATE* distribution tree.

5.2 The symbol table

The C++ grammar makes a distinction between a name and a user defined type name. To make this work, previously defined type names must be collected. Distinction can then be made based on this collection. Whether a name is a user defined type, depends further on its *context* that consists of any combination of (nested) scopes, namespaces, and inheritance. This information is tracked by the symbol table. The symbol table implementation is an enhancement and a port to C++ of the original java based symbol table functionality that came with the original javaCC C++ grammar. It resides in the *SymbolTable* subdirectory. The top level class of the symbol table is the *SymtabManager* class. At the C++ grammar level, the above name / type name distinction is further implemented by means of semantic lookaheads [10] based on the *SymtabManager* functionality.

Distinction between a name and a user defined type name is necessary for source code analysis. The advantage of having this already resolved in the C++ grammar, is that the synthesized parser will do this sorting automatically, so no extra pass on the AST is needed. Another advantage is that parsing of C++ code is easier with this distinction in place.

The disadvantage of this approach is that the source code has to be parsed in its entirety, to pick up all the user defined type definitions, while this is not strictly needed when only the outermost application layer of the source code is to be analyzed and / or transformed. This results in quite a big overhead on the AST. The solution to this problem is to initialize the symbol table in an alternative manner. One way to do this is to create a compact file with appropriate user defined types by hand and to parse this file before and together with the application layer. This method is not very attractive because of the manual intervention that is required. Another way is to save and restore the symbol table from a file. Both

methods are supported by this front end and they can also be combined. After a set of files have been parsed, the symbol table is saved in a proprietary format. In the next run, the symbol table is initialized directly from this file after which only the top level application layer needs to be parsed. To automate the selection of files, so which files need to be parsed and which files can be skipped, the symbol table also contains information from which files it has been previously built. After reloading of the symbol table, the source file is parsed, and include files are skipped based on this information, see also appendix G.

5.3 Using a conventional C / C++ preprocessor

To deal with compile directives, a decision had to be made whether to use a conventional C / C++ preprocessor and to parse the preprocessed files, or to extend the C++ grammar with compile directives and re-implement the C / C++ preprocessing functionality. Both methods have advantages and disadvantages. The advantage of using a conventional C / C++ preprocessor, is that no re-implementation of C / C++ preprocessing functionality is required which saves design effort. The disadvantage is that the preprocessed code is less clean because of all the macro expansions, there may be compiler / preprocessor specific hooks, the include file hierarchy has to be reconstructed from the line directives, an include file cannot easily be skipped because it is expanded into the preprocessed file, and the comments are gone. Re-implementing the C preprocessing functionality just reverses the advantages / disadvantages.

In the end, we decided to go for the conventional C / C++ preprocessor because re-implementing the C preprocessing functionality seriously complicated things. Besides the implementation effort, there is the additional problem that user defined types may also be expressed as macros. This means that macro expansion needs to be implemented in the parser in order to have a consistent symbol table lookup. This was found to be a serious complication, at least in scope of this project. And, when really required, this challenge can always be taken up in the future. However, having this choice in place, we need to deal with the disadvantages as well. Firstly to simplify things, we decided to standardize on the g++ preprocessor. This breaks portability to other compilers, however this is not really an issue within the scope of this project.

g++ preprocessor specific hooks This problem could easily be fixed by extending the C++ grammar. With the *C3PO* synthesis capabilities in place, this took almost no effort. And when needed, these hooks can just as easily be removed again.

Reconstruction of the include file hierarchy The include file hierarchy was reconstructed by means of the available line directives in the preprocessed source file. Reconstruction was easy to do by extending the grammar with the *SourceUnit* and *Module* constructs. *SourceUnit* represents the top level source file, and a *Module* the included header file. The beginning and the end of a module is expressed by means of the *START-MODULE* and *ENDMODULE* tokens that are based on the line directives. Having all this in place, reconstruction is performed automatically by the synthesized parser.

Skipping of include files Skipping of include files is not that easy, because they have been expanded in the preprocessed file. However, since the include hierarchy can be reconstructed, skipping can be seen as a seek forward in the file just beyond the end of the expanded file. But, because of the include hierarchy, parsing is needed here as well to properly identify the end of the expanded include file.

Sub parser multiplexing The above parsing problem has been solved by introducing new sub parser multiplexing functionality in *C3PO / ANTLR*. With this functionality it is possible to synthesize a sub parser from an entirely different grammar description. A hook is then added to the main parser to delegate to the sub parser at a certain point in the input stream, and another hook in the sub parser reverts back to the main parser at a later point. To make this work, both parsers have to read from the same input stream. The main problem with this approach is to precisely synchronize both parsers on this stream. Both parsers must start / re-start at exactly the right point. A further complication is that an *ANTLR* generated parser works with three buffered streams that delegate to each other. The *InputBuffer* that reads directly from the standard *istream* object, the *CharBuffer* that wraps around the InputBuffer, and the *TokenBuffer* that reads from a *TokenStream* object and holds the tokens for the parsers. The TokenBuffer is filled by the lookahead calls of the synthesized parser. During a parser switch, all these buffers need to be flushed, repositioned and refilled exactly right and in a transparent way to both parsers. The mentioned hooks implement this functionality based on the *parseinfo* object (appendix A.8), and on the for this purpose extended *ANTLR* C++ runtime library. This library can be found in the *c3po/cxx/src/antlr* subdirectory.

Having all this in place provides a technology that allows for switching to a completely different *interpretation mode* at a certain point in the file, with the possibility to revert back to the original interpretation mode at a later point. This method is very generic, and can also be used to skip files. Here, the main parser has a very fine grained interpretation mode, while the sub parser has a very coarse grained interpretation mode. The sub parser only parses the line directives and builds a representation of the include hierarchy. Everything else is skipped. In this way, a "skipped" file is represented in the AST by this coarse grained representation of the include hierarchy. This is not exactly the same as skipping a file, but it comes close as memory consumption is concerned. The sub parser resides in the *ModuleHierarchy* sub directory.

5.4 Multi file compilation

The application layer, that needs to be analyzed and transformed, consists of many source and header files. To save dynamic memory, it would be advantageous to process each source file at a time just like a conventional C++ compiler does. We call this the single file compilation mode. Unfortunately, the single file compilation mode creates a serious problem for source code transformations. Firstly, such a transformation can easily span multiple files and so the transformation has to be handled in multiple runs of the tool. This means that information needs to be communicated between the runs. Secondly, the

transformation of one file can depend on the information in another while both files are not available at the same time. Thirdly, one transformation may trigger another which only adds to this problem. All of this seriously complicates things and so the single compilation mode has been abandoned and replaced by the multi file compilation mode. In this mode, all the files are read in at once and so these problems disappear. One AST is built for the whole application layer such that everything is available at the same time. As stated above this approach uses more dynamic memory w.r.t single file compilation. In the next section strategies are introduced to reduce this drawback.

5.5 Provisions for single file compilation target

Because the transformed sources are supposed to be compiled by a conventional compiler, provisions have to be made that the created sources from the multi file AST will still compile in the single file compilation domain. This provision has been filed under the patent ID PHNL030382. The remainder of this section is a short summary of this invention.

When a header file from the application layer is included by more than one source file, multiple representations of this header file will be present in the AST. Each instance originating from a different preprocessed file[7]. These multiple representations needlessly waste dynamic memory and create a maintenance problem. So a mechanism has been put in place to eliminate the duplicate representations. One solution could be to skip all but the first inclusion of such a header file, and to reuse its representation at all the other places of the AST.

This is however not that trivial as it seems, because these representations may not be completely equal. This is caused by the fact that the preprocessed result of the original header file may vary due to a different set of compiler directives. The effects of the compiler directives are diverse, a practical abstraction into the following two cases is sufficient for our purposes,

Local effects In this case, only small parts of the header file representation are not equal. These effects are too diverse and it is difficult to perform a detailed analysis. Instead, a simple heuristic is applied that assumes that these effects will be found in representations of header files that are #included by the same *system file*, and, that this is steered from the *parent system files*. The compile directives that cause these effects are thus supposed to be embedded in the system file include hierarchy, and not above it. This means that it is still safe to reuse a toplevel system file representation. So, a system file that is included directly from the source file, or from a header file of the application layer. Of course this approach may fail when a source file, or any of its application headers steer the local effects in the system file, but this case is assumed to be rare.

Conditional inclusions In this case, a header file is either completely included or skipped by the preprocessor. It is skipped if that header file has already been encountered in

[7] For each source file there is one corresponding preprocessed file.

the same preprocessor run. This means that representations of the same header file may miss references to other header files. A representation that does not miss any references is the one that should be reused. We call such a representation *clean* and the others *unclean*. The trick is now to find and distinguish a clean representation from the unclean ones. Also, to simplify the skip and reuse process, it would be very convenient if an arrangement could be found such that a clean representation of any header file will always come first before any of its unclean representations.

Load order calculation It turns out that such an arrangement can indeed be found. This is achieved by using a specific order in which all the preprocessed files are read. It is also possible to determine this order automatically. This is based on the identified *software component clusters* and their hierarchy. A software component is a software entity to be analyzed or transformed. Such a component can for example be a user defined datatype. A software component cluster associates a software component with the source and header file(s) that describe it. Furthermore, a source file is uniquely associated with a preprocessed file. The required load order is now the order in which the preprocessed files are ordered in a bottom up manner based on the associated software component hierarchy. Header files that cannot be associated with a source file through a software component can simply be *wrapped* in a separate source file. This is a file that only simply #includes the header file.

5.6 C++ grammar extensions

Four special rules have been added to the C++ grammar. The first three, the *SyncStatement*, *SyncStatementType*, and the *ScopeStatement* are used to annotate the source code with *synchronization information*, see section 8.3.3. With a Graphical User Interface (GUI) in place this would not be necessary, and so later when a GUI has been added these rules could be removed again. Because of the *C3PO* synthesis technology extensions - like these - to an existing grammar can be done or undone very easily.

The fourth rule is the *EpicsStorageSpecifier*. This rule has been added to support the Epics specific C extensions. Supporting compiler specific extensions should be kept to a minimum to avoid unnecessary complexity of the C++ front end. The lack of a standard in this area makes the situation only worse. As these DSP compilers emerge, it would be good to discuss or have a standard to conform to as their extensions are concerned. Within the scope of this project however, the extension is limited to only this one rule.

5.7 The PreProcessingVisitor

The preprocessing visitor is the last step in the parser front end. In this step a general transformation on the C++ code is performed to simplify subsequent analysis and transformations of the source code. The transformation performed here is very basic. Firstly, the case statements are processed to associate them more directly with the case label.

Secondly, the expanded assert macro is transformed back to its unexpanded state. More transformations of this kind can be added if so desired.

6 The transform language

In this chapter a language is introduced that allows the user to specify a large variety of transformations. They range from preprocessing transformations, functional transformations, up to and including structural transformations. Furthermore, another language is introduced in which the process network API definition can be configured. This configuration language handles libraries, header files, the syntax of data and synchronization calls, and many other aspects.

6.1 Transformation

The idea of the TRANSFORMATION specification is to be able to group together a set of predefined transformations in reusable library components. System designers can freely specify their own group(s) of desired transformations from a set of possible transformations, see Table 2. Hopefully the number of these specifications and their names will become stable after a while such that they become well defined library blocks with names that express their intentional use.

OriginalStatementType:	TransformStatementType:
IF	REWRITE
SWITCH	REWRITE
OPERATOR	REWRITE
DECLARATION	REWRITE
DECLARATION	SKIP
FOR	REWRITE
WHILE	REWRITE
DO	REWRITE
=.	LOAD
.=	STORE

Table 2: All these individual transformations can be selected and combined - per scope see section 8.3.1 - in a TRANSFORMATION section.

Figure 8 shows an example TRANSFORMATION specification. The inner section specifies the desired transformations grouped together by the identifier *ProducerRewrite*. There may be more than one group of transformations in a TRANSFORMATION specification each with its own unique identifier. In this example it is, respectively, specified that *read* and *write* calls should be skipped, meaning removed from the source code[8]. *For* loops should be rewritten (see section 8.3.1) and left hand side occurrences of a variable that is being mapped should be transformed into store operations (see section 8.3.2). Note, the order of the transformations in this list is irrelevant.

[8]These references to the function names *read* and *write* are the only hard coded dependencies on YAPI or TTL in the *SCATE* tool.

```
TRANSFORMATION
{
  ProducerRewrite
  {
     READ -> SKIP
     WRITE -> SKIP
     FOR -> REWRITE
     .= -> STORE
  }
}
```

Figure 8: Example of a TRANSFORMATION section.

As stated a set of transformations as defined by *ProducerRewrite* in Figure 8 can be used when variables within a certain program scope are mapped. One simply refers to the desired library component as will be explained in detail in section 8.3.2.

6.2 Mapping specification

A MAPPING specification binds transformations to YAPI / TTL design entities. We call such a design entity simply a *unit*. A unit is either a *functional* unit, like a YAPI process or a TTL task, or it is a *structural* unit, like a YAPI process network. Therefore, transformations are also categorized as either functional or structural. Besides the transformation itself, one must also address a unit or an instance of a unit the transformation should be applied to.

Figure 9 shows a generic template for the mapping specification language. Roughly, the specification consists of a set of MAPPING sections. These sections can also be nested. Nesting represents the hierarchy of the system or a part of a system. The outer section specifies always a unit, while the inner sections specify unit instances located within the parent unit. Transformations are further specified by means of specific constructs (see sections 6.3.1– 6.3.6) within the mapping sections.

```
MAPPING <unit specification>
{
  <Transformation construct>
  <Transformation construct>
  . . .
  MAPPING <instance specification>
  {
    <Transformation construct>
    <Transformation construct>
  }
  . . .
}

MAPPING <other unit specification>
{
  . . .
}
```

Figure 9: Mapping template.

6.3 Addressing of units and instances

The following addressing schemes are available.

Generic The generic addressing scheme is specified by a $\star$. Such a section matches to either all units, or to all instances in the parent unit.

Named The named addressing scheme consists of a set of comma separated name specifications. A name specification consists of a unit name optionally followed by a colon and a comma separated set of instance names. An outermost mapping section matches to all units whose names match to any of its name specifications. For a nested mapping section, the section maps to *to all instances* of the named units within the parent unit. The optional instance names can further be used to specify only some instances of that unit.

These schemes can freely be nested and combined, see figure 10.

```
MAPPING *
{
  T  // Transform T applies to each unit
}

MAPPING u1, u2, u3
{
  T // applies to units u1, u2 and u3
}

MAPPING *
{
   T1 // applies to each unit
   MAPPING *
   {
     T2  // applies to each instance within each unit
   }
}

MAPPING u1
{
   MAPPING u3, u4
   {
     T1 // applies to each instance of either u3 or u4
        // within the parent unit u1
   }

   MAPPING u3 : i1, i2,
           u4,
           u5 : i7
   {
     T2 // applies to the following instances
        // within the parent unit u1 :
        // instances i1 and i2 of unit u3 and
        // all instances of unit u4 and
        // instance i7 of unit u5
   }
}
```

Figure 10: Example mapping sections.

6.3.1 Structural transformation statements

The following structural transformation statements are available,

SPLIT $name_1, \ldots, name_n$

SPLIT *name*

SPLIT

The split statement applies to either a unit or to an instance of a unit. For a unit, the statement makes copies of the source unit and assigns them the new names. This means that new unit types are created. To split a unit, at least one name must be specified.

When the split statement is applied to an instance of a unit, a copy is made of that unit, and the instance becomes an instance of the new unit. For the name of the new unit, the first name of the split statement is taken. But the name is optional here: when no name is specified a suitable name is generated.

split management The split statement is important when transformations are performed on a subset of instances of a unit, but not on all instances of that unit. In this case, new units have to be created for the modified instances otherwise the transformation would apply to all instances. At this moment, the user has to add proper split statements by hand, in the future an automatic split management mechanism may be provided.

FLATTEN The flatten statement removes the hierarchy of the unit it is applied to. After this operation, the unit will only contain instances of functional units. So units that do not contain any other units.

MULTICAST This statement is intended for removal of the YAPI FORK processes. It is applied on an instance of a fork process. The ports that are driven by the fork process are reconnected to the input channel that drives the input of the fork process. Then the dangling fork output channels are removed together with the fork process instance. In this way multi cast is introduced for the fork input channel.

REMOVE This statement removes either an instance of a unit or the unit itself. When it is applied on an instance, the instance is disconnected and subsequently removed. When it is applied on an unit, all instances of this unit are removed after which the unit itself is removed. Removal of a unit means that its definition is removed form the structural representation together with all cross references to it.

REMOVE NOINST The NOINST variant applies only to a unit. Here, the unit is only removed if the unit has not been instantiated. It is intended as a clean up statement after instance removal. If all instances of a unit have been removed this statement removes the unit. Otherwise it does nothing.

REMOVE TEMPLATE The TEMPLATE variant also only applies to a unit. It reduces the template parameters of the unit. These template parameters are supposed only to parameterize the transport datatypes of the unit ports and channels. Instances of the unit become instances of a splitted unit that has the template parameters removed and replaced by the actual template parameter values of the instance. When the original unit has no more instances it is removed as well.

6.3.2 Processor mapping

The processor mapping binds a unit to a certain processor type. This means that the functional code of that unit should run on that processor. Optionally a processor instance can be specified. The instance is identified by either a name or a number. The processor type is further bound to a build configuration section (see section 6.5).

```
PROCESSOR Arm
PROCESSOR Mips : 3
PROCESSOR Mips : Tuner
```

6.3.3 Transport

The intention of a TRANSPORT section is to define a synchronized container for tokens. These tokens are (usually) transported from one process to another process. In practice such a container is defined by selecting a YAPI FIFO or a TTL Channel. Synchronization is defined by a combination of the interfaces provided by the Channel, and an accompanying set of invariants [9].

Within each MAPPING section one may specify at most one TRANSPORT section. This section contains the definition of one or more *TransportInstance*s. A *TransportInstance* specifies an identifier, optionally the size of the transport entity, the kind of channel(s) that is / are used, the transport type, and in more complex cases also the synchronization of the channel.

Figure 11 shows an example TRANSPORT section. The inner section defines a *TransportInstance* with identifier 'Array' of size 1000. There may be more than one of these *TransportInstance* sections, but each of them should be identified by a unique name. The channel used to implement this *TransportInstance* is a *TTL::Channel* of transport type *int*. The exact syntax and semantics of this channel should have been defined within a API definition with the api identifier TTL, see section 6.4. The scope operator '::' is necessary in those cases where the channel name is not uniquely defined, e.g. when more than one API definition is provided to the *SCATE* tool using the same name for a channel. By default the channel is assumed to be a combined blocking channel.

[9] Invariants are required for all interface types with one exception, the interface type CB.

It is allowed to use more than one channel within one *TransportInstance* when the following restriction is obeyed. The functions provided by the interfaces of the channel should at most contain one definition for each combination of a direction and the data transfer, i.e. one definition for IN:DATA and OUT:DATA. The same holds for each combination between a direction and synchronization.

```
TRANSPORT
{
  Array[1000]
  {
    TTL::Channel<int>
  }
}
```

Figure 11: Example of a *TransportInstance* section.

In practice a channel may provide more than one interface. For instance a *TTL::Channel* offers more than one interface for establishing separate synchronization and data transfer. In such a case it is obligatory to specify which interface(s) is meant. This can be done with the keyword *USING*, see Figure 12. So apparently in this example the synchronization needs to be changed. How this new channel needs to by synchronized can be specified as well using invariants, see section 8.3.3 for a detailed explanation.

```
TRANSPORT
{
  Array[1000]
  {
    TTL::Channel<int> USING RbIn, RbOut
    {
      "1000" * #Consumer::C <= "1" *#Producer::P
      "1" * #Producer::P <= "1000" * #Consumer::C + 1000
    }
  }
}
```

Figure 12: Example of a *TransportInstance* section that makes use of the keyword USING. How this channel needs to be synchronized is expressed using invariants.

Some channels return a token reference. In order to make a binding between a variable and the token reference one should define a *VariableBinding*. Such a variable should be specified followed by the symbol $\Longleftrightarrow$, and the scope it connects to with a label, see Figure 13[10]. This kind of mapping is still not fully supported in *SCATE*, and requires

[10]This can only be used for interface types that make use of a return token like in TTL; DBI, DNI, DBI

additional work to be done.

```
TRANSPORT
{
  Array[1000]
  {
    TTL::Channel<int> USING DbiIn, DbiOut
    {
      "1000" * #Consumer::C <= "1" *#Producer::P
      "1" * #Producer::P <= "1000" * #Consumer::C + 1000
    }
    var1 <=> Consumer::C
    var2 <=> Producer::P
  }
}
```

Figure 13: Example a *TransportInstance* section that makes use of variable binding.

6.3.4 Net mapping

Figure 14 shows an example NET mapping section. This section expresses that an existing channel instance called 'fifo' is mapped onto a *TransportInstance* with identifier 'Array'. In due time it is the intention to allow mapping onto free style TRANSPORT sections as described in the previous section. However, at the time of writing NET mapping is more restricted, i.e. one can map exactly one channel at the time that offers a combined synchronization interface onto another channel that offers the same interface[11]. This allows the user to map one YAPI channel onto another YAPI channel, and it also allows one to map one YAPI / TTL channel onto another TTL channel that offers a combined blocking channel. Note, in all cases the size of the channel may be defined in this mapping step as well. Furthermore, there may be more than one of these channel mappings within a NET mapping section.

and DNO.

[11] In the example shown here the construct USING is explicitly used. This is not needed at the time of writing, due to the restricted implementation of NET mapping. This may change over time.

```
TRANSPORT
{
   Array[1000]
   {
     TTL::Channel<int> USING CbIn, CbOut
   }
}
NET
{
  fifo -> Array
}
```

Figure 14: Example NET section.

6.3.5 Variable mapping

Some transformations are restricted to work on only a selected region of the source code. Such a region can be manually defined by means of annotations of the source code as shown in Figure 15. Therefore the keyword SCOPE plus identifier has been introduced followed by a body within curly braces. This allows the user to simply refer to this SCOPE identifier *X* from within a mapping file. Unfortunately, this extension to the C++ language prevents one to compile the (*SCATE* input) code without additional effort. Hopefully, this is not a problem, since these SCOPE regions are automatically removed from the source code by *SCATE*, i.e. after transformation SCOPE regions never occur within the source code.

```
void Producer::main()
{
    const int n = 1000;
    SCOPE(X)
    {
        int R[n];
        for(int i=0;i<n;i++)
        {
            R[i]=i;
        }
        write(out, R, n);
    }
}
```

Figure 15: Example of annotated source code with a SCOPE statement.

A SCOPE identifier is used when variables are mapped. The idea of a VARIABLE mapping section is to specify which local variables should be mapped onto a channel - i.e. outside the process or task -. This means that all accesses to the variable at hand within the process or task should be made explicit. This requires a set of transformations within the source code that are specified in a TRANSFORMATION specification, see section 6.1.

One VARIABLE mapping section may be specified within a MAPPING section. An example VARIABLE mapping section is shown in Figure 16. Like in the source code example one specifies a SCOPE with an identifier - this time without braces -. This uniquely defines the region within the source code where transformations should occur, since the embracing MAPPING section defines the unit[12]. Within a SCOPE section one may specify one or more source code variables (separated by commas) that should be mapped onto a channel outside the process or task. In this example, source code variable *R* within SCOPE *X* is mapped onto *TransportInstance* 'Array' which is defined in a MAPPING section one level up ('../') in the mapping specification. The transformations that are applied within this region - some of which act on variable *R* only - is defined by the TRANSFORMATION section *ProducerRewrite*, see section 6.1. Note, variable mapping is concerned with data transfer and **not** with synchronization. Thus the fact that variable *R* is mapped onto *TransportInstance* 'Array' does not mean that 'Array' is automatically synchronized. As will be explained in section 6.3.6 this requires additional mapping information.

In this specific example it is specified that left hand side occurrences of variable *R* should be transformed into explicit store operations. Assuming 'Array' is defined in Figure 12 the definition of interface RbOut should then provide a function definition for OUT:DATA, see section 6.4.8. The name and syntax of this function is then used for the store operations in the source code.

```
VARIABLE
{
  SCOPE X
  {
    R -> ../Array TRANSFORMATION ProducerRewrite
  }
}
```

Figure 16: Example of a VARIABLE mapping section.

6.3.6 Task mapping

Figure 17 shows an example TASK mapping section. This specifies that all instances of tasks Producer and Consumer are synchronized according to *TransportInstance* 'Array'. There may be more than one of these task mapping lines.

[12]There is an issue when the MAPPING section only defines an instance, in this case one should first split the instance before doing this kind of transformations, see section 6.3.1

So a VARIABLE mapping section is concerned with data transfer, and a TASK mapping section is concerned with synchronization. This separation is made to allow, respectively, for unsynchronized pieces of memory, and semaphore channels without explicit data transfer.

```
TASK
{
  Producer, Consumer -> Array
}
```

Figure 17: Example TASK section.

6.4 API definitions

The API definition descriptions allow for specification of those semantics of the YAPI/TTL like APIs that are required in order to properly recognize, transform, and validate the APIs. The API definition provides also customization of the *SCATE* tool towards a new or modified API. Without this customization, the *SCATE* tool would have to be modified in many places.

Roughly, an api definition description consists of the API keyword, followed by an api identifier and a description body within curly braces. The api identifier is not bound to any special meaning within the tool, the relevant api semantics are fully defined by the description body. The following is an example of the top level view of the api descriptions,

```
API YAPI
{
. . .
}

API TTL
{
. . .
}
```

The description body consists of general constructs that capture the relevant API semantics. These constructs are highly tuned towards YAPI like apis. This means that only those apis that have a high syntactical similarity with the YAPI api can be captured. More concrete, sufficient expression power is available to capture the YAPI(C++), TTL(C++), and TTLC(C) apis. When more constructs are needed, they can easily be added through *C3PO*. The following subsections describe the current constructs.

6.4.1 Language

The language construct describes the implementation language of the API library. At the time of writing, this can either be C++ or C. This information is used to automatically determine whether a C++ to C conversion is required (see section 8.3.5). The following is an example of the syntax used.

```
LANGUAGE = CXX  // C++ language
LANGUAGE = C    // C   language
```

6.4.2 Identification

The ident(ification) construct specifies the name of the YAPI like Id datatype,

```
IDENT = Id
```

6.4.3 Process and Network

The process and network sections specify the names of the YAPI like process and network superclass datatypes, the in, out, and parameter datatypes for the port member declarations within the YAPI like class definition, and the *MPC* generated subdirectory for all the processes and networks respectively.

```
// YAPI
PROCESS Process
{
   PORTMBR IN    = InPort
   PORTMBR OUT   = OutPort
   PORTMBR PARAM = Param
   DIR           = Process
}
NETWORK ProcessNetwork
{
   PORTMBR IN    = InPort
   PORTMBR OUT   = OutPort
   PORTMBR PARAM = Param
   DIR           = Network
}

//TTL
PROCESS Process
{
   PORTMBR IN    = Port
   PORTMBR OUT   = Port
   PORTMBR PARAM = Param
   DIR           = Process
}
NETWORK Network
{
   PORTMBR IN    = NetworkPort
   PORTMBR OUT   = NetworkPort
   PORTMBR PARAM = Param
   DIR           = Network
}
```

6.4.4 Include and library option arguments

These directives specify the api related include (-I) and library option arguments. Please note that these are only the option *arguments* so, for example, the -I prefix should not be included here.

```
//YAPI
INCLUDE = "$(YAPI_INC)"
LIBRARY = "$(YAPI_LIB)/libyapi_g++_3_2.a"


//TTL
INCLUDE = "$(TTL_INC)"
LIBRARY = "$(TTL_LIB)/libttl.a"
```

6.4.5 Header files

This section specifies all the header files that come with the api. This information is used to distinguish between the extra header files of the application layer and the api header files.

```
//YAPI
HEADER = "io.h",
         "id.h",
         "fifo.h",
         "port.h",
         "process.h",
         "network.h",
         "select.h",
         "rte.h",
         "workload.h",
         "yapi.h",

         "scateChess.h"

//TTL
HEADER = "process.h",
         "network.h",

         "cb_in_port.h",
         "cb_out_port.h",

         "rb_in_port.h",
         "rb_out_port.h",

         "rn_in_port.h",
         "rn_out_port.h",

         "dbi_in_port.h",
         "dbi_out_port.h",

         "dbo_in_port.h",
         "dbo_out_port.h",

         "dno_in_port.h",
         "dno_out_port.h",

         "dni_in_port.h",
         "dni_out_port.h",

         "scateChess.h"
```

6.4.6 The main and type member function signatures

These directives specify the main and type member function signatures. For the type member function, only the name of the function is relevant. For the main member function, there is also a variant with a function parameter which controls the function signature relevant aspects of the C++ to C conversion. In the following example only the simple variant of the main member function is shown. The others are described in section 8.3.5.

```
MAINMBRFUNC = main
TYPEMBRFUNC = type
```

6.4.7 MPC

The *MPC* section specifies the api related *MPC* driver name, the corresponding driver option, and the default channel. The *MPC* driver is YAPI / TTL heterogeneous. The option specifies the default behavior which can be overridden on a per unit basis by the driver name.

```
// YAPI
MPC
{
   OPTION      = "-yapi"
   DRIVER      = YAPI
   DFLTCHANNEL = Fifo
}

//TTL
MPC
{
   OPTION      = "-ttl"
   DRIVER      = TTL
   DFLTCHANNEL = Channel
}
```

6.4.8 Channels

A channel provides a number of interfaces. A task may connect to such an interface via a port. Through this mechanism a task can make use of the functions provided by an interface.

Figure 18 shows the set of interface types provided by TTL at the time of writing. Their full names are listed in table 3. For each of these types a channel with interfaces and functions is defined with its specific semantics. In the coming sections the specification of these interface-types will be treated in some detail.

Before getting started we would like to note that it is the intention of the *SCATE* to exclusively provide support for transformations of interface types towards the right hand side of Figure 18. Meaning that for example interface type I can be transformed into type II, but not the other way around. This has to do with the fact that the interface types go down in abstraction when going from the left hand side to the right hand side of the figure. Therefore more and more detail has to be provided, i.e. design decisions have been made that cannot be undone.

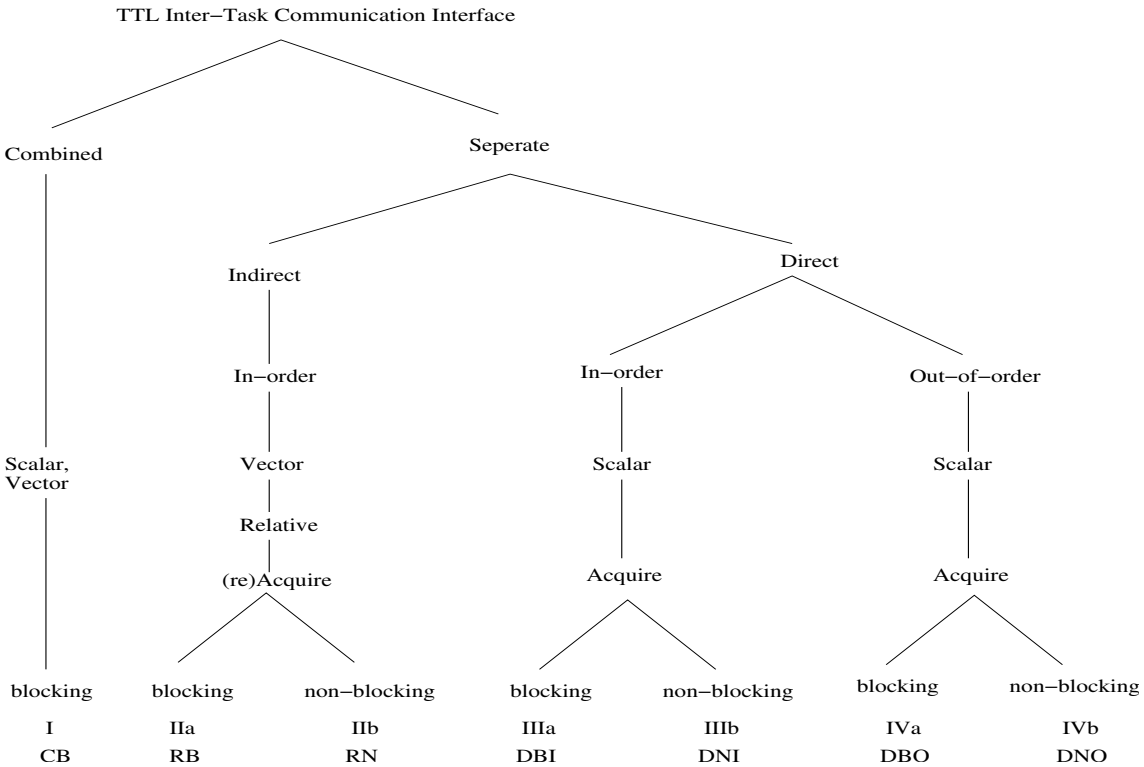

Figure 18: TTL Interface types.

The header of a CHANNEL section starts with an optional TEMPLATE keyword and ends with an identifier, see Figure 19. The keyword TEMPLATE indicates that the channel uses a templated transport datatype. So far this information has not been used within *SCATE*.

Acronym	Full Name
CB	Combined Blocking
RB	Relative Blocking
RN	Relative Non-Blocking
DBI	Direct Blocking In-order
DNI	Direct Non-Blocking In-order
DBO	Direct Blocking Out-of-order
DNO	Direct Non-Blocking Out-of-order

Table 3: TTL Interface Types.

```
TEMPLATE CHANNEL Fifo
{
...
}
```

Figure 19: *ChannelDefinition* section using the TEMPLATE keyword.

Header files The header file(s) needed to provide the implementation of a channel can be specified within the *ChannelDefinition* section using the HEADER keyword followed by the names of the header files, see Figure 20. This(These) header file(s) is(are) then included in the generated header file that describes the network structure.

```
HEADER = "fifo.h",
         "port.h"
```

Figure 20: Specification of header files that specify the channel implementation.

Parameters The meaning of the channel constructor parameters can be configured. This is done within a *ChannelParametersDefinition*, see Figure 21. Such a section starts with the keyword PARAMETERS thereafter within curly brackets one can configure that the minimum size, maximum size or simply the size of the channel is used in the channel constructor parameter list. Moreover one then also needs to configure the position within the channel constructor parameter list starting by zero.

```
PARAMETERS
{
  MIN  = 1  // second constructor parameter is minSize
  MAX  = 2  // third  constructor parameter is maxSize
  SIZE = 3  // fourth constructor parameter is size
}
```

Figure 21: Specification of channel constructor parameters.

Interfaces Table 4 and 5 show the accompanying interfaces of all interface types provided by TTL.

Interfaces can be specified within a *ChannelDefinition* section, see Figure 19. An interface section specifies the name of the interface and the number of these interfaces provided, see Figure 22. This example shows the specification of one of the interfaces of the TTL interface type CB. Finite numbers are symbolized by a number between square brackets. Infinite can be specified by not specifying the number. Currently this information is parsed, however, within the *SCATE* tool this information is not yet used. Eventually one could implement checks whether the use of a certain channel is according to these numbers specified. Besides the number of interfaces provided one can also indicate whether the interface supports multicast, by specifying the keyword MULTICAST. This information is used when source code for synchronization calls are generated on the basis of an invariant specification, see section 8.3.3.

Just like at *ChannelDefinition* level one can specify the header file(s) associated with the interface.

Interface Type	CB	RB	RN
Input interface	CbIn<T>	RbIn<T>	RnIn<T>
#Input interfaces	1	1	1
Output interface	CbOut<T>	RbOut<T>	RnOut<T>
#Output interfaces	1	1	1

Table 4: Preliminary set of channels and their interfaces provided by TTL.

```
INTERFACE CbOut[1]
{
   MULTICAST
}
```

Figure 22: Specification of an interface.

Interface Type	DBI	DNI	DBO	DNO
Input interface	DbiIn<T>	DniIn<T>	DboIn<T>	DnoIn<T>
#Input interfaces	1	1	1	1
Output interface	DbiOut<T>	DniOut<T>	DboOut<T>	DnoOut<T>
#Output interfaces	1	1	1	1

Table 5: continuation of Table 4.

Functions In Table 6-12 all available functions for the various interfaces of TTL are listed. This set is likely to change over time, which is fine since the user can easily adapt the specification files.

Interface	CbIn	CbOut
function	write(...)	read(...)

Table 6: Functions available per interface.

Interface	RbIn	RbOut
function	reAcquireData(...)	reAcquireRoom(...)
	load(...)	store(...)
	releaseRoom(...)	releaseData(...)

Table 7: continuation of Table 6.

In Figure 23 an example of an interface function specification is shown. The interface at hand provides a read function with two arguments in the parameter list, a port and a data value. The example chosen is very simple, more complex functions can use explicit call by reference type of constructions '&' or provide return values[13]. Between the (inner) curly brackets it is specified that the function provides data transfer and synchronization from a consumer perspective. The direction must be specified for both the data transfer and the synchronization provided. A consumer(producer) type of operation in this sense is characterized by IN(OUT). Furthermore, a distinction is made between the starting point of a section for which synchronization needs to be provided and the enclosing point. The former is specified by the keyword ENTRY while the latter is specified by the keyword LEAVE.

[13]For more complex examples we refer to the API definition of TTL as provided by the *SCATE* tool.

Interface	RnIn	RnOut
function	tryReAcquireData(...)	tryReAcquireRoom(...)
	load(...)	store(...)
	releaseRoom(...)	releaseData(...)

Table 8: continuation of Table 7.

Interface	DbiIn	DbiOut
function	acquireData(...)	acquireRoom(...)
	releaseRoom(...)	releaseData(...)

Table 9: continuation of Table 8.

```
INTERFACE CbIn[1]
{
   FUNCTION read(PORT, VALUE)
   {
      IN:ENTRY
      IN:DATA
      IN:LEAVE
   }
}
```

Figure 23: Specification of an interface that provides a read function, i.e. combined synchronization and data transfer from a producer point of view.

Interface	DniIn	DniOut
function	tryAcquireData(...)	tryAcquireRoom(...)
	releaseRoom(...)	releaseData(...)

Table 10: continuation of Table 9.

Interface	DboIn	DboOut
function	acquireData(. . .) releaseRoom(. . .)	acquireRoom(. . .) releaseData(. . .)

Table 11: continuation of Table 10.

Interface	DnoIn	DnoOut
function	tryAcquireData(. . .) releaseRoom(. . .)	tryAcquireRoom(. . .) releaseData(. . .)

Table 12: continuation of Table 11.

6.5 Build configuration

For each processor type, the build configuration section specifies a number of processor target specific build tools and options. Each configuration option is optional and consists of a keyword, the assignment operator and a string constant that defines the option. The order of the configuration options within the build configuration is arbitrary.

```
PROCESSOR Mips
{
   BINDIR  = "/usr/local/Mips/bin" // the bin dir for the build tools

   CC            = "..." // C compiler and options
   CCOPTIONS     = "..."

   CXX           = "..." // C+= compiler and options
   CXXOPTIONS    = "..."

   LD            = "..." // Linker and options
   LDOPTIONS     = "..."

   SIMC          = "..." // Simulator compiler and options
   SIMCOPTIONS   = "..."

   SIMLD         = "..." // Simulator linker and options
   SIMLDOPTIONS  = "..."

   AR            = "..." // Archiver and options
   AROPTIONS     = "..."

   RANLIB        = "..." // The ranlib program and options
   RANLIBOPTIONS = "..."

   LIBPOSTFIX    = "..." // The library name postfix
}
```

7 YAPI and TTL recognition

Through analysis of the C++ AST, structural and functional information of a YAPI / TTL system is extracted and placed into an internal representation. We call this representation the YAPI representation. The extraction process is configurable by means of the API definition sections discussed in the previous chapter. This enables a configurable extraction of either YAPI or TTL, and of other YAPI like descriptions of the future.

7.1 The YAPI representation

The YAPI representation is described in the *YapiRep.G* grammar file. The code resides in the *YapiRep* subdirectory. The most important part of this representation is the *unit* datatype. This datatype represents either a YAPI / TTL process or network. A set of these units form the YAPI / TTL representation of a system. The unit consists further of datatypes that describe either the *structure* (App B.1), or the *functionality* (App B.2) of the unit.

7.2 The YAPI visitor

The *yapiVisitor* implements the actual recognition and extraction process. A unit is recognized from a class definition that inherits from a special superclass. This must be either a class whose name matches one of the *PROCESS* or *NETWORK* sections of the API definitions, or the name of an already recognized unit. In the first case, the API definition is also linked to the unit, which defines the unit to be implemented by that API definition. In the latter case we speak of indirect inheritance. The unit is here linked with the API definition of its superclass unit. Ports are built from the constructor parameters, and the corresponding member declarations. The relevant member initializer constructs of the constructor implementation link a parameter name with the corresponding member declaration. Transport datatypes are derived from the template argument of the constructor parameter. Channels and instances are also build from the member declarations. Channels are recognized by means of the linked API definition. Instances are recognized to be declarations whose datatype names match another recognized unit. The relevant member initializers are analyzed to build the interconnect. The other code is filtered for relevancy, and if so it is stored into one of the relevant unit members that form the functional representation of the unit.

Sourcefile association A unit is also associated with files. One for its interface and one for its definition. Normally, this will be a header and a source file, but this is not necessarily so. For this reason, the YAPI representation contains also a file representation. A unit knows about the files that define / declare it. A file representation contains links to the units.

Software engineering style adaptation The recognition process can also deal with different software engineering styles, i.e., how the units are distributed over the source / header files. Conventionally, each unit will have a different associated header and source file. But situations where one file defines / declares more than one unit or where one unit is defined / declared in more than two files are dealt with as well. This brings the advantage that, after regeneration, the system will be converted to the conventional style. With this feature, the tool can be used to convert non conventional YAPI / TTL implementations to standardized implementations automatically.

7.3 API validation

After recognition, the extracted system representation is validated against its recognized API definition. Consistency checks are performed on the structure. Everything that does not fit the API definition is reported. In this way, any inconsistencies are tracked down in an early stage. Errors that would otherwise only surface during compilation of the transformed system.

7.4 Dynamic channel size derivation

A difficult problem to deal with is how to derive the right channel dimensions for the TTL implementation. A YAPI specification can have unbounded channel sizes, while in the TTL implementation the channel dimensions have to be set in such a way that deadlock does not occur. A solution that could be useful in some cases, is to insert the actual channel dimension after a YAPI simulation. This depends on the applied input of course, but when a representative input can be found this method could be useful. For this reason, a separate parser is available, that can parse the simulation statistics as generated by a YAPI simulation. This representation is then used to set the channel dimensions on the TTL channels. The source code that implements this feature resides in the *FifoStatus* directory within the *YapiRep* directory.

In practice one simply specifies this using the EXTRA_FLAGS in the Makefile-ScateSetup:

```
EXTRA_FLAGS = -channelInstanceStatus filename
```

where *filename* is the name of the file containing the simulation statistics as generated by a YAPI simulation.

7.5 Options

See appendix C for available options that steer the YAPI / TTL recognition, analysis, and (re)generation process.

8 Transformation

In this section structural transformations and functional transformations of the YAPI representation are discussed. On one hand there is a strong separation of structural and functional transformations. On the other hand, this separation can also be a bit vague because structural and functional transformations depend on, and influence each other. For example, when a new port is added, which is structure, functionality is also needed to send or receive data from this port. The top level transformation function is the *yapiRep::transform* function. This is mainly a wrapper around a large visitor based transformation infrastructure. The transformation infrastructure is controlled by the transform language (chapter 6) and some command line options (appendix C.3). The YAPI representation contains two sets of units, the *inputUnits*, and the *outputUnits*. The *inputUnits* form the input for the transformation infrastructure, the *outputUnits* are the selected units for code generation that may also have been transformed. Prior to code generation, a validity check on the *outputUnits* is performed and any inconsistencies are reported.

8.1 Visit schemes

The top level of the transformation infrastructure consists of two visitor classes. The *MappingVisitor* for structural manipulations and the *TransformationVisitor* for the functional transformations. This distinction had to be made because both visitors have very distinct visit schemes that cannot be combined in one visitor class. The *MappingVisitor* processes each mapping section in sequence and applies it to each unit. The *TransformationVisitor* does just the opposite, it processes each unit in sequence and applies all transformations on it.

8.2 Structural transformations

Structural transformations are mostly performed on an unit or on an instance of a unit. For some of these operations corresponding transformation statements are available in the transform language. These statements have already been described in section 6.3.1. The following transformations are available,

Multicast This is the multicast operation that corresponds with the MULTICAST statement.

Creation A structural creation API is available for creation of new units, instances, ports, instaceports, channelInstances, and nets.

Connection / Disconnection A connection / disconnection API supports manipulation of the net connections.

Deletion The deletion API is the inverse of the creation API. The deletion of units and instances corresponds also to the REMOVE and REMOVE NOINST statements.

Reduction of template parameters This operation corresponds with the REMOVE TEMPLATE statement.

Splitting See the SPLIT statement.

Scope Editing A scope edit operation manipulates the C++ scope of a unit. In this way, the C++ class name of a unit is changed, together with all its scoped member function definitions.

Reparenting A reparent[14] operation moves an instance to another hierarchal level together with all of its connections. The move is either up or down the hierarchy. This is a very powerful operation because repetitive reparent operations can complete change the system structure. It can also be used for flattening.

Flattening See the FLATTEN statement.

8.3 Functional transformations

In this section we discuss various aspects of functional transformations. Preprocessing transformations (section 8.3.1) are either done to make the real desired transformation easier to implement or to make the desired transformation at all possible. Various transformations are discussed in sections 8.3.3 and 8.3.2, whereby the border line between functional transformations and structural transformations is not very strict, so far for the title of this section. Postprocessing transformations are discussed in section 8.3.4. Finally, we conclude with section 8.3.5 on the conversion of source code from C++ to C.

8.3.1 Preprocessing

There are three visitor schemes involved in the preprocessing step. All three will be briefly discussed here.

SpecificationInitVisitor This visitor establishes cross links between specification elements, see Table 13. These cross links are set to simplify the gathering of data in the implementation of algorithms. For example at the level of *VariableMapping* cross links are made to the corresponding *TransportInstance*, *Transformation*, and *MappingSpecification*.

SetUnitTransportInstanceVisitor This visitor basically does what its name suggests. It creates a link between every *TransportInstance* and the unit in which it is instantiated.

[14] Unfortunately, the reparent operation has only been planned but never implemented because of other priorities.

Datatype	link to
VariableMapping	TransportInstance
	Transformation
	MappingSpecification
TaskMapping	MappingSpecification
	TransportInstance
ChannelMapping	MappingSpecification
	TransportInstance
Channel	ChannelDefinition
MappingSpecification	MappingSpecification

Table 13: Cross links established by *SpecificationVisitor*.

TransformPreProcessingVisitor This section provides an informal description of the preprocessing transformations provided by *TransformPreProcessingVisitor*. These transformations are sometimes required when local variables are explicitly mapped onto resources outside a process (section 8.3.2). The goal is twofold:

- It makes the abstract syntax tree (AST) more homogeneous by limiting the number of syntactical constructs without changing the functionality. As a result this implicitly limits the number of transformations.
- Some transformations would simply be much more complex or even impossible when executed in one go.

All the transformations provided by *TransformPreProcessingVisitor* are informally described using an example in the following sections. By default none of these transformations are performed. Instead, each of them can be individually selected in the mapping file, see section 6.1 Table 2. One reason for selecting these kind of preprocessing transformations is to enable variable mapping, see section 6.3.5

Figure 24, 25 and 26 show transformations applied on *SelectionStatement* "if".[15]

```
if (i=j,k=l)
{
    ...;
}
```

```
i=j
if (k=l)
{
    ...;
}
```

Figure 24: An *Expression* in the guard of an *SelectionStatement* construct is broken down at *AssignmentExpression* level. The *AssignmentExpression* at the far right hand site of the original *Expression* becomes the new guard.

[15]The other *SelectionStatement* construct "switch" is transformed in a similar way as shown in Figure 25 and 24.

```
if (int i=1)
{
   ...;
}
```

```
{
  int i;
  if (i=1)
  {
     ...;
  }
}
```

Figure 25: The *Declaration* in the guard is moved outside the scope of the if statement, therefore a new scope has to be introduced to prevent interference with other variables that may have been declared in the outer scope. The guard itself is replaced by an *AssignmentExpression*.

```
if (j)
{
   ...;
}
else if (int i=j)
{
  ..;
}
```

```
if (j)
{
   ...;
}
else
{
  {
    int i;
    if (i=j)
    {
       ...;
    }
  }
}
```

Figure 26: The *Declaration* is moved outside the scope of the if statement, therefore a new scope has to be introduced. The guard itself is replaced by an *AssignmentExpression*.

Figure 27 shows an example of an operator transformation. The following operators are all expanded in a similar manner[16]: $--$, $+=$, $\%=$, $*=$, $-=$, $<<=$, $>>=$, $\&=$, $|=$, $=$.

[16]This transformation acts on Statement-Expression level, thus operators occurring in parameter lists are NOT transformed. Recursive application of transformations, e.g. first transforming a "for" loop into a "while" loop, may solve part of this limitation.

```
i++;
```

```
i;
i=i+1;
```

Figure 27: An *Operator* is fully expanded into separate *Expression*s.

Figure 28 shows the breakdown of a *SimpleDeclaration* at *InitDeclarator* level.

```
int i=j,k=l;
```

```
int i=j;
int k=l;
```

Figure 28: A *SimpleDeclaration* is broken down at *InitDeclarator* level.

Figure 29, 30 and 31 shows the transformation of the *IterationStatement* "for" into "while".

```
for(i=1;j=2,k=2;l=3)
{
    ...;
}
```

```
i = 1;
j = 2;
while (k = 2)
{
   ...;
   l = 3;
   j = 2;
}
```

Figure 29: The first Expression of the parameter list is executed only once at the start of the loop. The second Expression of the parameter list is executed, but only the right hand site is used in the guard. The left hand site is repeated at the end of the loop body. The second Expression of the parameter list is executed after each iteration of the loop.

```
for(int i=1;j=2,k=2;l=3)
{
    ...;
}
```

```
{
   int i = 1;
   j = 2;
   while (k = 2)
   {
      ...;
      l = 3;
      j = 2;
   }
}
```

Figure 30: The *Declaration* is moved out of the parameter list. Therefore a new scope is introduced.

```
for(;;)
{
    ...;
}
```

```
while(true)
{
    ...;
}
```

Figure 31: No parameters in the for loop is translated into forever true.

Figure 32 and 33 show transformations applied on a “while” loop.

```
while (i=j,k=l)
{
    ...;
}
```

```
i = j;
while (k = l)
{
   ...;
   i = j;
}
```

Figure 32: The left hand site of the *Expression* is moved outside of the loop, and is repeated at the end of the loop body. Only the right hand site of the *Expression* is used in the guard.

```
while (int j=k)
{
  ..;
}
```

```
{
  int j;
  while(j=k)
  {
    ..;
  }
}
```

Figure 33: The *Declaration* is moved outside the guard which is replaced by an *AssignmentExpression*. Therefore a new scope is introduced.

Figure 34 shows a transformation applied on a “do” loop.

```
do
{
    ...;
}  while (i=j,k=l);
```

```
do
{
   ...;
   i = j;
}while (k = l);
```

Figure 34: The first *AssignmentExpression* is pushed out of the while parameter list.

8.3.2 Making indirect access explicit

ExpressionTransformation transforms a variable that occurs in an *Expression* (or *AssignmentExpression*) into a (hopefully) functionally equivalent *StatementList*, containing *load* and *store* operations on a channel, and a transformed *Expression* (or *AssignmentExpression*). No guarentees are provided, since the functional equivalence depends on many details like the semantics of the channel, the details of the TRANSFORMATION specification (section 6.1), etcetera. For the moment the strategy is to provide a flexible and generic framework for variable rewriting with the hope that functional equivalency can be obtained.

Variables The transformations provided by *ExpressionTransformation* can be used at will by the system designer. In practice this is done by mapping a local program variable (indicated by *R* in all examples shown here) onto a channel that provides separate functions for IN:DATA and OUT:DATA. The kind of channel used will influence the final syntax of the transformed variables in the program text as will be described in section 6.4.8.

In general one can distinguish between two cases, either a variable occurs as a r-value or as a l-value. Both cases are handled differently, see Table 14 and 15. These tables

demonstrate for all possible input *AssignmentExpression*'s the resulting *StatementList* and transformed *AssignmentExpression*. In some cases a fresh variable is introduced into the program.[17] The $type$ of a variable need not be specified but is automatically extracted from the declaration of the mapped variable.

Input AssignmentExpression used as r-value	Return value	
	StatementList	AssignmentExpression
i	-	i
R	T LOCAL_R_0; load(port,LOCAL_R_0);	LOCAL_R_0;
R[i]	T LOCAL_R_0; load(port,LOCAL_R_0,f(i));	LOCAL_R_0;

Table 14: Return values of variables - when used as a r-value. Variable R represents a variable which is mapped onto a channel, and T is used to indicate its type. Variable i represents a variable which is not mapped. Variables starting with $LOCAL$ should be a fresh variable in the current scope. Address calculation is not yet supported, instead a function call $f(i)$ is generated which requires manual adjustment.

Input AssignmentExpression used as l-value	Return value	
	StatementList	AssignmentExpression
i=j	-	i=j
R=j	store(port,j);	R
R[i]=j	store(port,i,j);	R[i]

Table 15: Return values of variables - when used as a l-value. Variable R represents a variable which is mapped onto a channel. Variables i and j represent variables which are not mapped. The return variables R and $R[i]$ are again processed within *Expression-Transformation* as described by Table 14

Algorithm In the previous section the transformation of fairly simple *AssignmentExpression*s was described. In general more complex *Expression*s should also be handled correctly. The following abstract algorithm is used in combination with the Tables 14 and 15 to enable this:

1 Breakdown at *Expression*[18] level from right to left (depth first).

2 Breakdown at *AssignmentExpression* see footnote 18 level from right to left (depth first).

[17] Before introducing a new variable into the program text a check is performed whether it is a fresh variable in the current scope. If not, a new variable name is generated recursively.

[18] According to the grammar rules used by *SCOTTY*.

3 Process all r-values using Table 14 and do in place transformations.
POST-CONDITION: all variables at right hand side are transformed.

4 Combine the resulting r-value with the (depth-first) l-value.

5 Process the l-value, of the newly composed *AssignmentExpression*, using Table 15.
POST-CONDITION: all variables at left hand side are transformed.

6 Continue with 2 until finished.

7 Continue with 1 until finished.

8 DONE.

The only thing missing in this algorithm is that an *AssignmentExpression* within square brackets ("[]") is recursively processed using the same algorithm as soon as it is encountered.

In order to clarify the algorithm in more detail let us consider an example. Assume as input for *ExpressionTransformation* the following *AssignmentExpression*:

$$R = R + i; \tag{1}$$

Furthermore, we assume a TRANSFORMATION specification where load and store operations are introduced in the current scope, see section 6.1. Now according to the algorithm:

1 Input is broken down at *Expression* level. Since there is only one *Expression* nothing is done.

2 Input is broken down at *AssignmentExpression* level, see Figure 35.

3 The r-value is transformed using Table 14. This is shown as the stage I of Table 16.

4 The return *AssignmentExpression* of the previous step in the algorithm is combined, from right to left, into a new input *AssignmentExpression* used as input in stage II.

5 In the stage II the l-value is transformed using Table 15.

6 Continue with step 2 of the algorithm.

2 The *AssignmentExpression* cannot be broken down in smaller peaces.

3 The return value of the previous step can no longer be combined with anything else so this is further processed as an r-values as shown in stage III.

Resulting into:

```
T Local_R
load(INport,Local_R)
store(OUTport,Local_R+i)
T Local_R_0
load(INport,Local_R_0)
Local_R_0
```

Additional transformations on top of this result in a slightly different outcome w.r.t. $Local_R + i$. In practice the result of this *Expression* is put into a local variable, which on its turn is used as a parameter for the store operation.

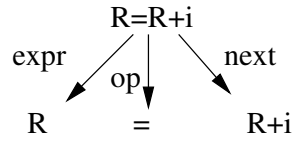

Figure 35: Breakdown of an *AssignmentExpression*.

	AssignmentExpression	Return value	
		StatementList	AssignmentExpression
I	$R + i$	$TLocal_R$ $load(INport, Local_R)$	$Local_R + i$
II	$R = Local_R + i$	$store(OUTport, Local_R + i)$	R
III	R	$TLocal_R_0$ $load(INport, Local_R_0)$	$Local_R_0$

Table 16: Example showing the stages which are used to transform *Expression* $R = R+i$.

8.3.3 Synchronization

In this section we use a technique to map process network communication onto multi-processor architectures with shared memory. This technique is called Action Synchronization.

Invariants Table 17, 18 and 19[19] show the relationship between program hooks, invariants on one hand, and synchronization calls on the other hand. The designer must manually add program hooks, in the form of ENTRY(A) and LEAVE(A) statements, into the program text. These statements define the points where synchronization calls may be introduced. The name of a label - A in this example - can be chosen freely. The label A

[19]Note, this interface type is currently NOT provided by TTL. The example illustrates the fact that semaphores can be handled with the method used.

as defined within ENTRY(A) and LEAVE(A) statements in combination with invariant(s) referring to such a label determines what kind, and how many, synchronization call(s) is / are inserted into the program text.

program hook	invariant pattern	N	RB
ENTRY(A)	$S::A + \ldots \leq \ldots + N$	> 0	acquireRoom(...)
LEAVE(A)	$\ldots \leq S::A + \ldots$	0	releaseData(...)
ENTRY(A)	$S::A + \ldots \leq \ldots$	0	acquireData(...)
LEAVE(A)	$\ldots \leq S::A + \ldots + N$	> 0	releaseRoom(...)

Table 17: Relationship between program hooks, invariant patterns, and the resulting synchronization constructs in the program text for the TTL C++ interface type RB.

program hook	invariant pattern	N	RN
ENTRY(A)	$S::A + \ldots \leq \ldots + N$	> 0	if (!acquireRoom(...)) return
LEAVE(A)	$\ldots \leq S::A + \ldots$	0	releaseData(...)
ENTRY(A)	$S::A + \ldots \leq \ldots$	0	if (!acquireData(...)) return
LEAVE(A)	$\ldots \leq S::A + \ldots + N$	> 0	releaseRoom(...)

Table 18: Relationship between program hooks, invariant patterns, and the resulting synchronization constructs in the program text for the TTL C++ interface type RN.

Table 20 demonstrates various patterns, that are often observed in invariants. As can be seen there may be more than one reference to the same label. These may either occur within one invariant and / or be distributed over multiple invariants. Depending on the pattern this leads to the introduction of one or more synchronization call(s). With the exception of those interface types that do not define the use of ENTRY and LEAVE. This could for example be the case for an interface type that offers no synchronization functions but only data transfer functions.

One interesting pattern to take a closer look at is called $multicast$. This consists of multiple occurrences ($n > 1$), for a single channel instance, of the pattern (with $i = 1 \ldots n$):

$$\begin{aligned} \sum_k S_i::A^k &\leq \sum_j P::B^j \\ \wedge \sum_j P::B^j &\leq \sum_k S_i::A^k + N \end{aligned} \tag{2}$$

where labels A^k are used in consumer i, and labels B^j occur in the producer. Superscript is used here to indicate multiple - possibly different - labels.

program hook	invariant pattern	N	Semaphore
ENTRY(A)	$S::A + \ldots \leq \ldots + N$	> 0	P(...)
LEAVE(A)	$\ldots \leq S::A + \ldots$	0	V(...)
ENTRY(A)	$S::A + \ldots \leq \ldots$	0	P(...)
LEAVE(A)	$\ldots \leq S::A + \ldots + N$	> 0	V(...)

Table 19: Relationship between program hooks, invariant patterns, and the resulting synchronization constructs in the program text for an interface type that provides semaphores.

Assuming one producer (P) and two consumers (S_1 and S_2, both using the same label), we have ($n = 2$, $j = 1$ and $k = 1$):

$$\begin{aligned} S_1::A \leq P::B \wedge P::B \leq S_1::A + N \\ \wedge S_2::A \leq P::B \wedge P::B \leq S_2::A + N \end{aligned} \tag{3}$$

Such a *multicast* pattern introduces only one - for every side w.r.t. $\leq$ - synchronization call for every reference to label B within the producer program text. In all other cases there is a one to one relation between a label and a synchronization call within the program text.

use of a label in invariant(s)	invariant pattern
once	$S::A + \ldots \leq \ldots$
	$\ldots \leq S::A + \ldots$
multiple times	$S::A + S::A + \ldots \leq \ldots$
	$\ldots \leq S::A + S::A + \ldots$
	$S::A + \ldots \leq S::A + \ldots$
multicast	$\sum_k S_i::A^k \leq \sum_j P::B^j$ $\wedge \sum_j P::B^j \leq \sum_k S_i::A^k + N$

Table 20: Various patterns in invariants. $S::$ is the scope operator and, A and B are labels.

Address generation There are various ways in which address calculation is supported by TTL. For example in order means that the acquired tokens are known by the channel, and out of order means that the acquired tokens are known by the task. Currently, none of these address calculations are supported by the *SCATE* tooling. Instead an address is always put to '0', so manual adjustments of offsets is required.

Visitors involved The implementation of structural transformations can become quite complex, therefore multiple visitors are involved in an attempt to keep things simple. The

key idea is that all these visitors communicate with each other via one common datatype that is handed over. Each visitor enriches this datatype with specific information such that in the end all necessary information is gathered. After which the desired structural transformation can be applied using the common datatype.

The common datatype is based on three other datatypes shown in Table 21 to accommodate the necessary information. Note that information about the *unit* where *ports* are instantiated, and where *channel*s are instantiated is separated. This is necessary because both are instantiated at different levels in the hierarchy.

datatype	The information it contains
ChannelUnitStatementSet	”Set of ChannelUnitStatement”
ChannelUnitStatement	”Unit with a set of ChannelStatement”
ChannelStatement	”channelInstance with a set of PortsUnitStatement”
PortsUnitStatement	”Unit with its portset.”

Table 21: Data structure that contains the information build by several visitors.

The *StructTransTaskMapVisitor* derives structural transformations based on *TaskMapping*. The source code of the *unit* for which *TaskMapping* is specified is visited. *SyncStatement*'s together with invariants determine how many and which *port*'s should be created in this *unit*. The *unit* where a *channelInstance* needs to be added is obtained via the *transportInstance*.

The *StructTransVarMapVisitor* derives structural transformations based on data transfer transformations. These transformations are introduced as a result of *VariableMapping*. Via the *TransformationStatement* it is decided if *load* or *store* mapping is allowed. If so, a *channelInstance* is created - in case this was not already done by the *StructTransTaskMapVisitor* - in the *unit* of the *TransportInstance*. *Port*'s are created in the *unit* visited.

The *StructuralInstanceMappingVisitor* collects the instance path's of the channels and ports which should be introduced based on the analysis as described in the previous sections.

Finally all this information collected - in the datatype *ChannelUnitStatement* - is used to implement the structural transformation. This is done via a static function of *unit*.

Source code As mentioned above the source code is transformed based on ENTRY and LEAVE points and the corresponding invariants. Within *TransformationVisitor* a transform function on a *SyncStatement* is implemented. A *SyncStatement* is an extension of the C++ language grammar that specifies ENTRY and LEAVE as constructs.

Within this transform function a call is made to *getSyncStatList()* which runs over multiple datatypes. This function returns a *StatementList* that is inserted at the place where previously a *SyncStatement* occurred within the program text.

8.3.4 Postprocessing

Unused ports and channels in the design are not removed by default. In order to remove them one should explicitly specify the *-removeUnusedPorts* and / or the *-removeUnusedChannel* flag in the Makefile-ScateSetup file using the EXTRA_FLAGS:

```
EXTRA_FLAGS = -removeUnusedPorts -removeUnusedChannels
```

Initially the *RemoveUnusedPortsVisitor* starts by making a list containing all the ports of a unit. Then this visitor runs over all the source code of the unit and checks if a port is referenced at the Statement-Expression level. If so the port is removed from the list. The postcondition of this algorithm is that one is left with a list of ports that is never referenced in the source code. These ports are then removed from the unit with an accompanying warning at the command line. Note, this algorithm runs over all units.

At the end of the algorithm the function *removeUnusedNetworkPorts* of *unit* is called. This removes ports at a network level that is connected to a *net* that is not driven and has no drivers.

Similarly, the *removeUnusedChannels* flag triggers a function called *removeUnusedChannels* specified in *unit*. This function checks if a *net* connected to a *channelInstance* has connected *instance*s. If this is not the case the channel is removed.

8.3.5 C++ to C conversion

The C++ to C conversion is implemented in the *Cxx2CVisitor*. It is not our intention to implement a fully functional C++ to C conversion for every construct in the language. Instead, only the functionality that is really needed in the YAPI / TTL to TTLC translation context has been implemented. To make this transformation also useful outside the YAPI / TTL(C) context, the transformation has been kept generic, and independent of the YAPI / TTL(C) semantics. Even so, the implementation of this transformation proved to be quite expensive and elaborate. Roughly, we support the conversion to C of the YAPI C++ infrastructure, while it is assumed that the functional code of the processes is more or less plain C. The following code manipulations are performed,

Conversion of a class definition to a struct definition The generation of the struct definition is actually performed by the *MPC* TTLC driver. To make this work, member declarations of the original class have to be converted first. This is handled by one or more of the following items.

Introduction and propagation of the 'this' pointer Each reference to a member variable *m*, has been replaced by a

```
thisVar->m
```

construct. An appropriate 'this' variable is introduced and inserted into the parameter list of the member functions that have such member variable references in their

function body. The parameter declaration is also propagated up the function call hierarchy. The generic approach is to make the 'this' variable to be a pointer to the corresponding struct definition.

An exception to this forms the 'this' parameter of the main member function. The TTLC standard requires this parameter to be of the *void** datatype. This breaks the generic approach, but to accommodate for this nevertheless, the parameter of the main member function can be configured by means of the MAINMBRFUNC directive of the transform API (see section 6.4.6). This is the API we are transforming to, so generally TTLC. Besides the original function name, an optional parameter specification has been added to this directive. In the body of the main function, and in all the sub-functions the standard this pointer variable is used. Only the parameter of the main function is allowed to be something else. The following options are available,

default behavior

```
MAINMBRFUNC = main
```

Adds the standard this pointer parameter to the main function. The following is an example,

```
void
MP3_Decode_Frm_main(TTLCMP3_Decode_Frm MP3_Decode_FrmThisVar)
```

no parameter

```
MAINMBRFUNC = main()
MAINMBRFUNC = main(void)
```

Does not add a this parameter to the main function, but instantiates an external declaration to the this pointer variable in the body of the main function. This variable still needs to be defined somewhere else, this is not handled by this transformation and is thus left to the application programmer.

example,

```
void MP3_Decode_Frm_main(void)
{
   extern TTLCMP3_Decode_Frm MP3_Decode_FrmThisVar;
   ....
}
```

Explicit parameter

```
MAINMBRFUNC = main(T t)
MAINMBRFUNC = main(T)
```

```
MAINMBRFUNC = main(void*)
```

Adds a this parameter of type *T*, with optionally the name *t*. If the parameter name is omitted, it defaults to *thisHandle*. In addition, the pointer operator * can postfix the type. Within the function body, a simple cast is performed from the specified type, to the 'standard' type. Care should be taken that the specified type can be simply casted to this type, otherwise errors will occur...

examples,

```
MAINMBRFUNC = main(T t)
```

gives,

```
void MP3_Decode_Frm_main(T t)
{
   TTLCMP3_Decode_Frm MP3_Decode_FrmThisVar =
   (TTLCMP3_Decode_Frm) t;
   ...
}
```

```
MAINMBRFUNC = main(T)
```

gives,

```
void MP3_Decode_Frm_main(T thisHandle)
{
   TTLCMP3_Decode_Frm MP3_Decode_FrmThisVar =
   (TTLCMP3_Decode_Frm) thisHandle;
   ...
}
```

```
MAINMBRFUNC = main(void*)
```

gives,

```
void MP3_Decode_Frm_main(void* thisHandle)
{
   TTLCMP3_Decode_Frm MP3_Decode_FrmThisVar =
   (TTLCMP3_Decode_Frm) thisHandle;
   ...
}
```

---

member initializers These are converted to assignment statements and placed within the constructor bodies. The original member initializers are removed.

Moving declarations to the beginning of a scope In C, a declaration can not follow a statement within the same scope, while in C++ this is allowed. So, all declarations are moved to the beginning of the scope. Alternatively, a new scope may be created by means of an extra compound statement.

Reduction of scoped declarations and expressions Each

```
T::t
```

is converted into

```
T_t
```

Reduction of simple new expressions These are the *new* expressions that can directly be converted to *malloc* calls. Constructor calls with arguments are not converted. For these expressions appropriate warning messages are generated. Examples,

```
new int    is converted into malloc(sizeof(int))
new int[n] is converted into malloc(sizeof(int) * n)
new T(3)   no conversion, a warning message is generated
```

Delete expressions These are replaced by corresponding *free* calls. No checks are performed whether the corresponding constructor calls have indeed been replaced for malloc calls. This is not a problem because these will only be the constructor calls with arguments. And for these, proper warnings are already generated.

Conversion to globals Static member variables, member function declarations and member function implementations that are placed within the class definition are made global.

Simple eliminations The following constructs are simply eliminated from the code. This approach may not always produce the desired result, but this solution has been chosen anyway because a full fledged solution is too expensive.

- C++ library files, like iostream.
- The using directive.
- The reference (&) operators.
- The const function qualifiers.

9 Code generation

9.1 MPC input

A trade-off had to be made to either extend the original *MPC* YAPI driver [8, 9] for code generation, or to implement the code generation allover again within *SCATE*. The advantage of utilizing *MPC* is the re-use of an already existing elaborate functionality that would have to be re-implemented otherwise. The availability of other *MPC* drivers such as vhdl, verilog or a schematic generator is also very attractive.

MPC is in fact a structure generator. It is intended to generate a structure in numerous output formats, where 'structure' means the hierarchal interconnect of basic blocks, for example YAPI processes. Using *MPC* means thus that the YAPI driver needs to be extended in such a way that the functional code can be passed through from *SCATE*. The driver needs to be extended to generate the TTLC++ and TTLC formats as well.

In the end it was decided to go for the re-use of the already existing functionality in *MPC*; i.e. to use *MPC* as a back-end tool for *SCATE*. To preserve the user level design entry functionality of *MPC*, a new option *-backend* had to be introduced to make *MPC* behave as a back-end tool. The generation of *MPC* input code had to be separated into a structural part that corresponds with the *MPC* network language, and a functional part that has to be expressed in the *MPC* install language. Generation of the structural part is very straight forward. Generation of the functional part is more difficult. Here, the functional code has to be filtered and divided into many code fragments that must be hooked into the structural driver by means of many new install directives. These directives are summarized in appendix D.3. Additional provisions that had to be added to predict the default behavior of the *MPC* based code generation, and to steer this behavior through other install directives when necessary. An example of this is that because *MPC* re-generates the YAPI Process / Network constructor from the structural part, the constructor as encountered in the original source code needs to be stripped from the functional part. Only those additional parts that cannot be derived from the structural information, like additional code in the constructor body, have to be hooked in.

9.1.1 API specific steering

Having all this in place brings the advantage of heterogeneous system generation by means of API specific steering. What is meant by a heterogeneous system is that one part of the system can for example be YAPI, while another part can be TTL. At the moment this is not supported by the YAPI / TTL library because there are no trans-API channels. However, the automatic API recognition (per unit) based on the API definitions (section 6.4) makes it possible to generate each unit of a system in its own specific API. An install directive sets the API for each unit which controls the driver (API specific steering). This process could even be extended on a per channel or on a per port basis. In the per channel scenario, each channel in a process network can have its own API, and the connected ports are configured to match the channel. In the per port scenario, each

port can have its own API. Ports need to be connected by means of matching, possibly trans-API channels. The per channel and per port scenarios are not fully operational yet. The main reason for this is that we have never tested this because of the lacking support for these features from the API side. However, the tooling functionality is almost there, and as soon as the support form the API side is there, if so desired, it is expected to have the tooling support in place with minimal effort. A default output API can be set as well through the command line option. In this way only those install directives need to be set that reflect a deviant API from the default one.

9.2 The make system

The make system used is built in functionality that simplifies setting up and running *SCATE* and *MPC*.

Assuming there is a working directory called *$WORKDIR*, we recommend to put your application, i.e., all header and source files of your application without libraries that you do not want to transform, in the directory:
$WORKDIR/Generate/MyApp[20].
You can choose the name *MyApp*; probably you will choose the name of your application, e.g., *jpeg_decoding*. Once you have your files in this directory you can use *SCATE* from there.

9.2.1 setup

Executing *scate -setup* will generate the file *Makefile-ScateSetup*. This is the only file you should configure. Howto do this is described in appendix E.1.

9.2.2 Makefile-Scate

When you have configured the file *Makefile-ScateSetup* you can run *scate* from this file. To this end you execute *make -f Makefile-ScateSetup* which will generate the files *Make.SrcVars* and *Makefile-Scate*. *Makefile-Scate* is the actual makefile you will use to run *scate*. All the relevant options from the setup file have been transferred into this file. To run *scate* from this file, you execute *make -f Makefile-Scate* which will parse and transform your application and will generate input for *MPC*. You can delete the output of *scate* by executing *make -f Makefile-Scate very_clean*. There is also a less stringent option *make -f Makefile-Scate clean* that does not throw away the load order file (see section 5.5), *Make.SrcVars*, wrapper files (see section 5.5) or any of the preprocessed files.

[20]The application code needs to be fully functional.

9.2.3 Make.SrcVars

The *Make.SrcVars* file is included by the *Makefile-Scate* file and specifies the input sources that will be read by *scate*. During this run, *scate* will sort the input sources according to the load order calculation algorithm as described in section 5.5. This will also modify the *Make.SrcVars* file. See appendix E.2.

9.2.4 Make.MpcSetup

MPC also has a setup mechanism. To simplify things when *MPC* is used as a back-end tool, the setup mechanism of *MPC* is activated from within the *Makefile-Scate* file. This is achieved by placing the relevant *MPC* setup targets in the *Make.MpcSetup* file, and to include this file within the *Makefile-Scate* file. As a result of this, *MPC* will be run from the *Makefile-Scate* file to set up itself by means of the relevant *MPC* related options that have been recorded into the *Make.MpcSetup* file. This will generate all the relevant files to run *MPC*. See also Reference [8, 9], section 3.1. Among other things, the *Makefile-Mpc* file is generated, and is used to run *MPC*.

9.2.5 Makefile-Mpc

In order to assemble your transformed program into a compilable and linkable program, you can now run *MPC* by means of *make -f Makefile-Mpc*. If your transformed program is a YAPI program, then the program is generated in the directory:
$WORKDIR/Design/MyApp/Yapi

A TTL program is generated in:
$WORKDIR/Design/MyApp/TTL or *$WORKDIR/Design/MyApp/TTLC*
for C++ or C, respectively. You can compile and link your program by entering this directory and executing *../../Compile*. This compile script generates Makefiles using *imake* [6], and compiles and links your program.

The file structure for YAPI and TTL programs is as follows, networks are in the *Network* subdirectory, processes are in the *Process* subdirectory, and the top level network and main function are in the *TestBench* subdirectory. See also reference [8], section 3.5.

9.3 The mapping output files

When you run *scate*, it also produces mapping files. For each top level unit, a corresponding *.map* file is generated. This file reflects the current structural state of the top level unit, and lists its hierarchal composition in a top down manner. The format of this file is according to sections 6.2, and 6.3, with the transformation constructs omitted. First make a copy of the map file since it will be overwritten in the next run. These copied *.map* files can subsequently be edited and enriched with relevant transformation constructs, after which they can be used in subsequent *scate* runs. An example of a YAPI to TTL transformation can be found in appendix F.

9.4 Recapitulation

Getting started with a fresh application basically requires the following commands:

```
scate -setup;
make -f Makefile-ScateSetup;
make -f Makefile-Scate;
make -f Makefile-Mpc;
```

After modifying the application or other changes to any of the input files:

```
make -f Makefile-Scate clean;
make -f Makefile-ScateSetup;
make -f Makefile-Scate;
make -f Makefile-Mpc;
```

The '*make -f Makefile-Scate clean*' command is strictly speaking not necessary. However, some changes to the code may influence the install files causing side effects that were not anticipated. By removing the install files in each iteration side effects are avoided. Note, this command does not have a large penalty on the speed of the tools, in contrast to '*make -f Makefile-Scate very_clean*' that requires two runs to recover the correct load order.

9.5 Graphical views

There are various visualization views that can be generated by the *SCATE* tool. Two of them focus on the process network structure and use existing visualization tools; `dotty` [15] and `SequoiaView` [16]. Besides, two views on the file structure of the application at hand are offered.

9.5.1 SequoiaView

What is SequoiaView? At the TU/e a new method has been developed and implemented to display large hierarchical structures as squarified treemaps [16]. SequoiaView as the tool is called, uses a visualization technique named cushion treemaps to provide you with a view on your hierarchical structure in one single picture. An example of how the output looks like for a jpeg decoder is shown in Figure 36. Compare this sequoai view with the network view on the same jpeg decoder shown in Figure 37. What SequoiaView does not show are the channels and ports. In this way very complex hierarchical structures like the file structure on a harddisk can be visualized.

Additional information can be visualized by assigning a meaningful attribute like size or color. An attribute like a file size can be used to get a measure of the complexity of all the files in your process network in one view. One could also use color to visualize the modification date of the files to check whether reuse really works. One could potentially couple this viewer to your editor such that it allows one to quickly jump to the desired

process in your design. A user interface for the *SCATE* tool should allow this kind of interactions.

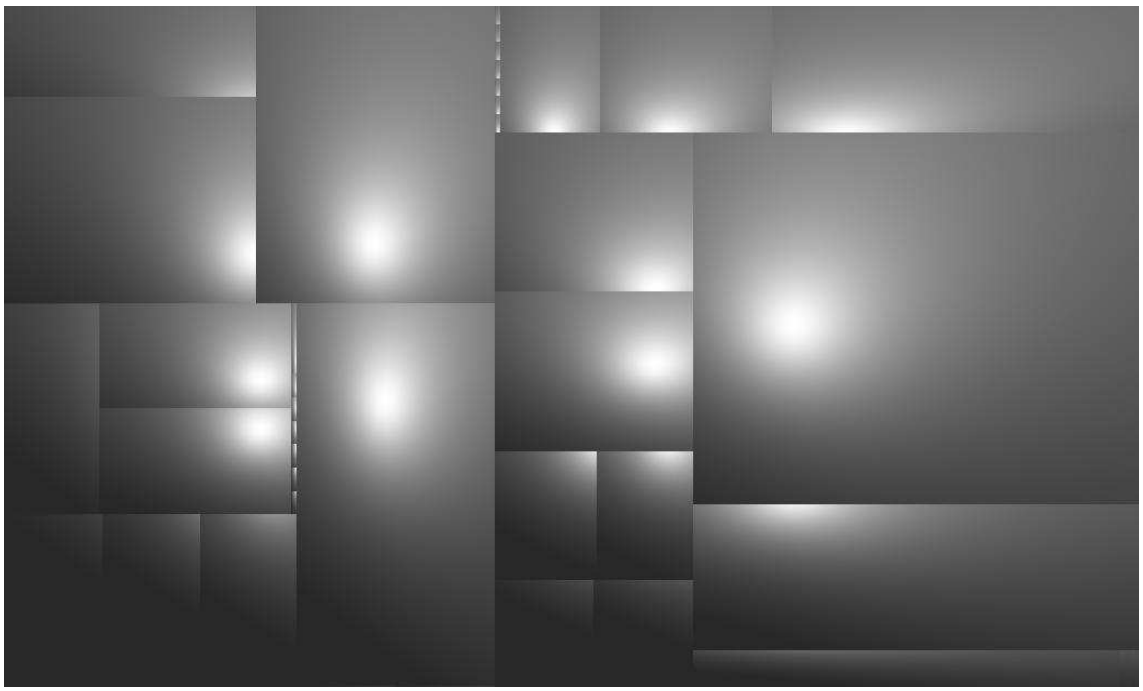

Figure 36: SequoiaView on a hierarchical structure of jpeg decoder.

Note that SequoiaView is available at the TU/e [16] for the Windows platform only. On Linux one could try to use *wine* to run SequoiaView.

How to use SequoiaView? The option $-sequoiaView$ is used to print a process network in SequoiaView format. One should use in the file Makefile-ScateSetup the EXTRA_FLAGS in the following way:

```
EXTRA_FLAGS = -sequoiaView
```

After creating the 'sequoiaview.tre' file by running the *SCATE* tool, the SequoiaView tool can be used to visualize the file.

For viewing options we refer to the SequoiaView manual pages.

Current implementation limitations In the current implementation the creation date and modification date of the file are fixed strings. Moreover, in case the 'file' at hand cannot be found a minimum file size is taken. This is because the algorithm assumes the lower case unit name plus extension ('.h' or '.cc') to be the name of the file. If this was not the case in the input files the current algorithm simply uses the default size.

9.5.2 Network view

What is dotty? When large process networks are created it is often difficult to obtain a good overview of the network. A graphical representation can help. The *Graphviz* tools [17] (`dotty` and `dot`) contain all ingredients needed to produce an graphical representation.

The dotty format is a textual format that can be used to view graphs. The program `dotty` can be used to view the graph via the X environment, or the program `dot` can be used to convert the graph into e.g. postscript format, using the following command:

```
dot -Tps dottyview.dot -o dottyview.ps
```

For output in other formats we refer to the *Graphviz* manual pages.

An example process network visualized via dotty is shown in Figure 37. Note that dotty is available for users at the Nat.Lab. through the *cadappl* tree (`cadenv graphviz`).

How to use dotty The option *-dottyView* is used to create a process network in dotty format. One can change the default options by specifying a file with the desired options. For both options one should use the EXTRA _FLAGS in the file Makefile-ScateSetup in the following way:

```
EXTRA_FLAGS = -dottyView -V /path/to/dotty.cfg
```

where /path/to/dotty.cfg specifies the location and filename that contains the desired configuration.

The output file created by running the *SCATE* tool is called *dottyview.dot*.

Dotty configuration The graph representation of the dotty format has various options to control the formatting of the graph output. The default options can be modified via a configuration file. In this text based file options are specified as `name = value` pairs. Below, the default option settings are listed.

Default Option Settings:

```
InitLevel                       = 10
TopLevel                        = 0
PageSize                        = A4p
Ratio                           = fill
Fifo                            = false
Decorate                        = false

ProcessNetworkColor             = white
ProcessColor                    = white
```

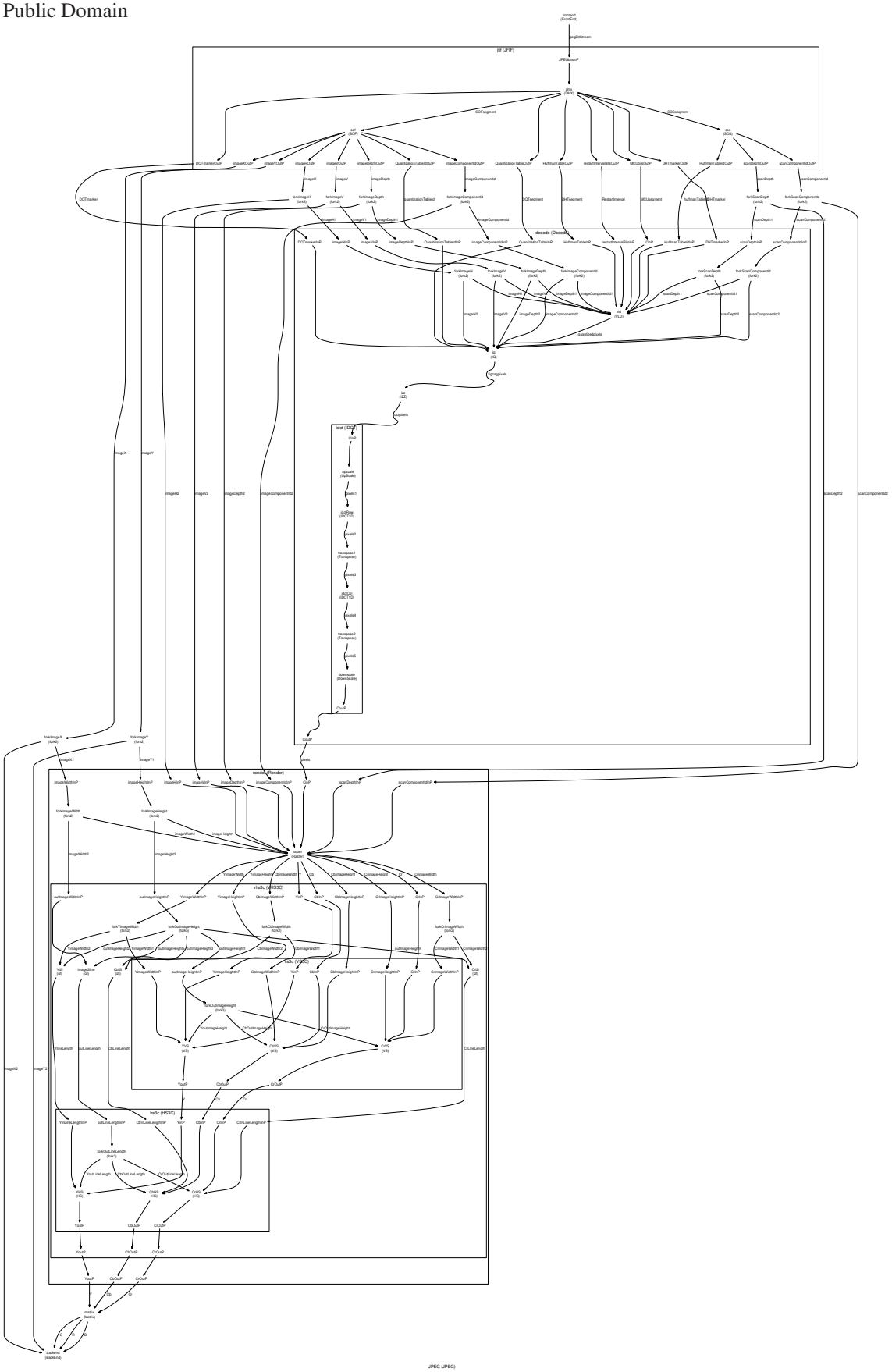

Figure 37: Process network structure of a jpeg decoder.

```
PortColor                       = white
FifoColor                       = white

ProcessNetworkClusterFontColor = black
ProcessNetworkFontColor        = black
ProcessFontColor               = black
PortFontColor                  = black
FifoFontColor                  = black

ProcessNetworkClusterFontName  = Helvetica
ProcessNetworkFontName         = Helvetica
ProcessFontName                = Helvetica
PortFontName                   = Helvetica
FifoFontName                   = Helvetica

ProcessNetworkClusterFontSize  = 20
ProcessNetworkFontSize         = 16
ProcessFontSize                = 16
PortFontSize                   = 16
FifoFontSize                   = 16
```

The options can be chosen from the following values:

InitLevel

InitLevel is specific for network view and will be discussed below.

TopLevel

Toplevel is specific for network view and will be discussed below.

PageSize: A4p, A4l, A3p, A3l, A2p, A2l, A1p, A1l, A0p or A0l.

Specifies the page size and orientation.

For example, A4p represents A4 portrait, A2l represents A2 landscape, etc.

Ratio: fill, compress, or auto.

Fill stretches the drawing, whereas *compress* compresses the drawing to fit it in the given page size.

Fifo: true, or false.

If set to true, FIFO are shown as diamond-shaped nodes, otherwise as normal edges.

Decorate: true, or false.

If set to true, a line is drawn that connects the FIFO label with the edges. These lines are only visible in the postscript output, not in X (dotty).

Color: white, lavender, gray, black, red, pink, brown, beige, orange, yellow, gold, green, cyan, blue, navy, magenta, purple, or violet.

Font: Times-Roman, Helvetica, or Courier.

Partial hierarchy viewing We will now discuss the options InitLevel and TopLevel when using the dotty configuration:

InitLevel: integer.

For the moment let's assume $TopLevel = 0$.

In this case InitLevel specifies the number of hierarchical levels that are shown. When set to 1, only the top level process network is shown, and all sub-networks are shown as simple nodes. When set to 2, the contents of the top level network and its sub-networks are shown, and all sub-networks are shown as simple nodes, etcetera.

TopLevel: integer.

The hierarchy levels shown actually depend on the difference between $InitLevel - TopLevel$, however, for the moment let's assume InitLevel = ∞.

In this case TopLevel specifies the top level of the hierarchical levels that is shown. When set to 1, the top level process network is the first level of the hierarchy, and all sub-networks are shown. When set to 2, the toplevel is the second level of the hierarchy, and all sub-networks are shown.

Let's take another more complicated example and assume we have a hierarchy which has four levels and we choose $InitLevel = 3$ and $Toplevel = 1$.

The number of levels shown now depends on $InitLevel - TopLevel = 3 - 1 = 2$. Thus what is shown is the first level (because TopLevel = 1) up to the second level (because $InitLevel - TopLevel = 3 - 1 = 2$, thus potentially up to the first two levels), which are shown as simple nodes. This interplay of TopLevel and InitLevel enables one to visualize the hierarchy from some starting point up to another level thereafter the details are hidden. Potentially more than one graph is shown since there may be more than one sub level which satisfies the requirements.

Let's take the example discussed in the previous paragraph and apply it on the jpeg decoder shown in Figure 37. There is more than one subnetwork which satisfies the requirement and the results are shown in Figure 38- 40.

9.5.3 Visualisation of the file and component structure

Two views are available for visualization of the file include hierarchy. Both views are generated into the *fileHierarchy.dot* file under control of the following command-line options,

-showFileHierarchy With this option, the entire file include hierarchy of a YAPI/TTL system is shown. The following figure illustrates this for the producer consumer example. The nodes of the graph represent the files, the edges represent the *includes* relation. The edge numbers represent the include order within the parent file. The arrow like pentagon shaped gray 'entry nodes', represent the preprocessed source

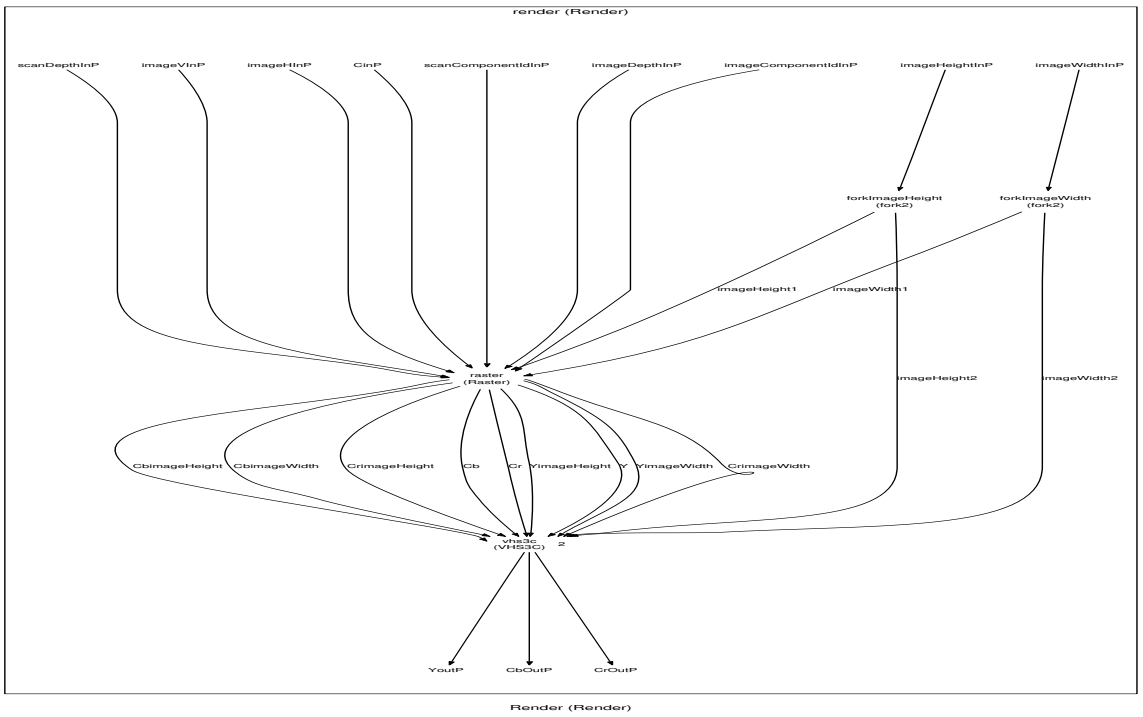

Figure 38: Subnetwork Render: dotty output of jpeg decoder network using $InitLevel = 3$ and $TopLevel = 1$, see Figure 39 and 40 as well.

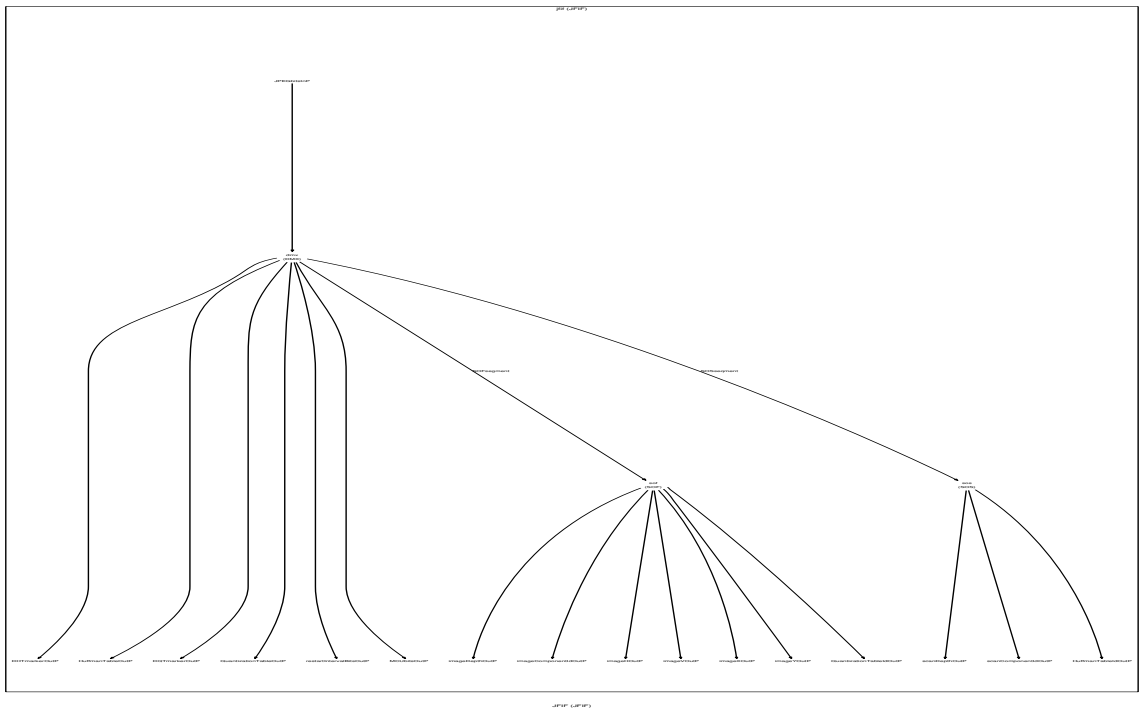

Figure 39: Subnetwork JFIF: continuation of Figure 38.

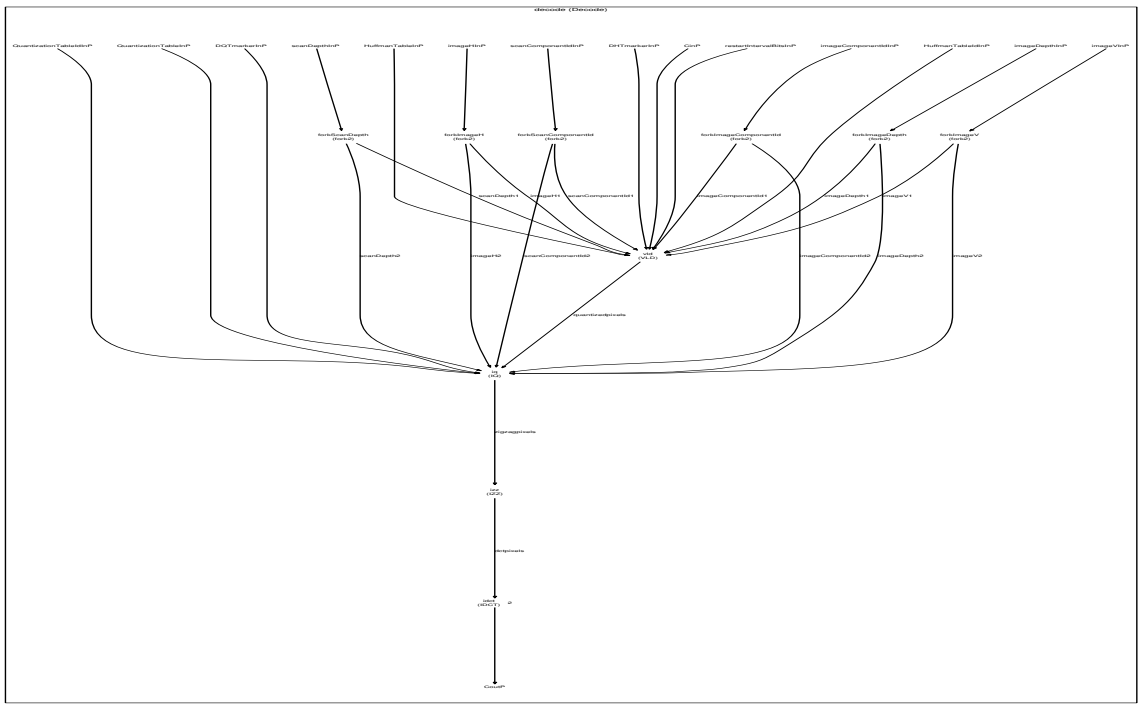

Figure 40: Subnetwork Decoder: continuation of Figure 38.

file. This file is the root of the include hierarchy. There are three of such roots, one for the producer, oner for the consumer and one for the producer-consumer network pc. This reflects the multi file compilation capability of the tool. The numbers of the root nodes represent the *load order* of the associated files. (See also 5.4, and 5.5).

-showHierarchy This option composes *software component clusters* annotations (section 5.5) to the file hierarchy. A software component is a software entity to be analyzed or transformed. A software component cluster is a sub graph that associates a software component with the source and header file(s) that describe it. Figure 42 illustrates this. The subgraphs are placed within the rectangular boxes. Besides the associated files, the subgraph contains also the software component itself. This is represented by the extra gray node. A circular shape represents a YAPI process or TTL task. The octagonal shape represents a network, a component with hierarchy. Two gray edges are added to represent the *implements* and the *declares* file association. The 'dot' endings of these edges at the border of the software component node reflect these association types. In case of the declares relation, the dot is not filled. In case of the implements relation it is. Same applies (not shown) for the software component nodes themselves. In case of a missing implementation, there will be no implements edge for this node, and the node itself is left unfilled.

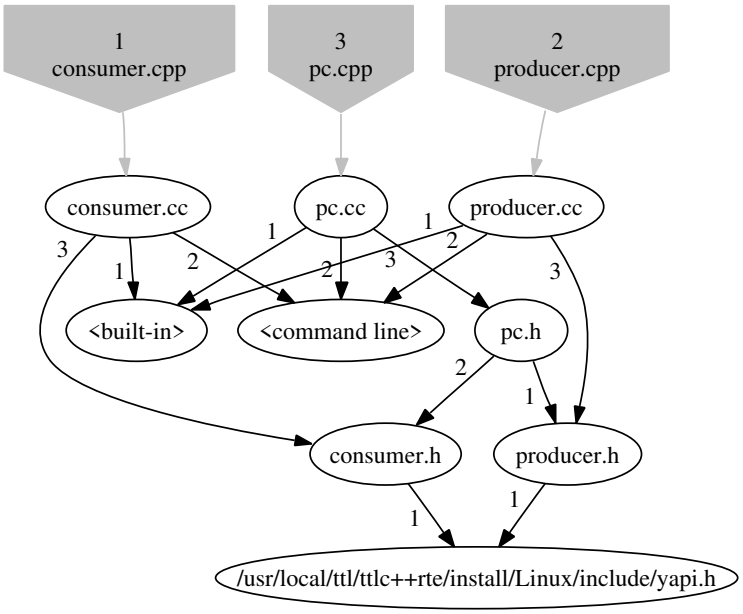

Figure 41: The file include hierarchy of the producer consumer example

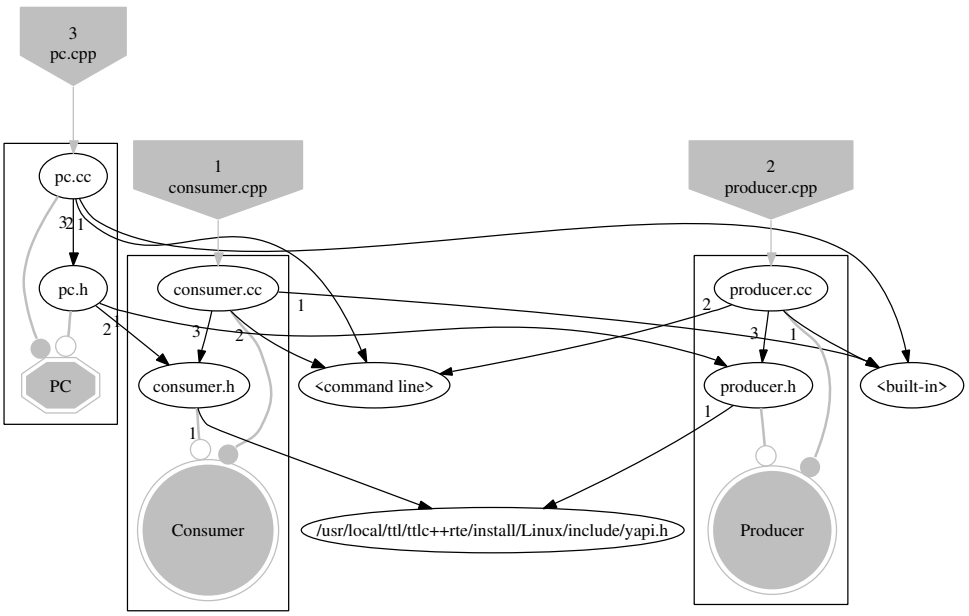

Figure 42: The file include hierarchy of the producer consumer example with software component cluster annotations

10 Testing

A distinction is made between local tests - performed by individual tool developers - and regression tests. The former is maintained on a private CVS branch of the tool developer. The latter is automated as much as possible to allow for regression testing without manual interference. In an ideal world one would run a cron job overnight, after which the output of the test would be sent by mail to all developers. Both forms of testing are maintained in a repository separate from the *SCATE* tool, since they contain applications that are considered to be IP of Philips.

Strategy All tests are maintained in CVS. This allows one to keep track of all changes - no matter how small or how large - as time progresses. This applies for all files involved; i.e. input files, intermediate files, and output files. We would like to stress the importance to keep track of all changes - at the source code level - during all stages of the development. For example, at the early stages of the development one can focus on the effects an additional transformation has - while being implemented - by focusing on the output source code. This is the case even when the generated process network cannot be readily compiled yet. Also side effects that were not anticipated beforehand immediately show up as differences between a previously stored version in CVS and the local file. In general one needs to understand all differences that may occur at source level. Once a difference is fully understood one could accept the (newly generated) files as a new reference point.

10.1 Local Tests

The following approach has been used to arrange local tests by individual tool developers:

- Keep the application code separate from the *SCATE* tool.
  - Application developers - not necessary members of the *SCATE* tool development team - maintain their applications as usual. Preferably under version management as well.
  - Each member of the *SCATE* tool development team can use a local copy of the application. This allows developers to work on a different version - preferably tagged - of the same application.
  - Applications are treated as (possibly sensitive) Philips IP. Someone who has a copy of the *SCATE* tool does not necessarily needs them.
- For the test directory we use separate branches in CVS for each of the developers.
  - It allows the developers to test different aspects of the tools.
  - The exact circumstances under which a bug surfaces can be easily recreated. This is done by tagging the application when a bug occurs, and by referring to this tag in the bug report. Other developers can start working on the problem

by simply updating to the branch and tag at hand (except of course for differences in environment variables and the like). No or at least less time is wasted in recreating the bug before one can start working on the real problem.

  - When automatic testing should be performed one could script these tests.

- From within a test directory links can be used to the application code at hand. In this way the application does not need to be part of the CVS tree of the tools.

  - One does not have to maintain a CVS tree of the application at hand.
  - The directory structure of the application does not need to be standardized. The assumption made is that the source code should not depend on the directory structure, i.e. no relative and absolute paths should be used within the application source code.
  - Small changes to the application code made by the application developer can be dealt with by a simple update command. With small we mean that the directory structure should remain the same and the filenames should not be changed as well. In some cases it cannot be avoided to copy application files to the test CVS tree. This is for example necessary when annotations of the original source code are needed to perform a transformation. A diff between the original sources and the annotated ones can easily reveal the changes made.

10.2 Regression Testing

A *Makefile-Test* can be automatically generated by using the *SCATE* flag *-mkTestMakefile* in the *Makefile-ScateSetup*, see appendix E.3. Furthermore, by providing the necessary information in the 'Test Related Part' of the *Makefile-ScateSetup* a ready to use and fully functional *Makefile-Test* can be generated, see appendix E.1.3. A standardized set of targets is provided by the *Makefile-Test*, since in the end this determines the tests that can be performed. In order to further automate regression testing a script is made available - called *scate_test* - that recursively runs over a directory structure in search of one or more *Makefile-Test* file(s). Based on the strategy described the following tests are automatically performed:

- Build native application
- Run native application
- Generate SCATE application
- Build SCATE application
- Run SCATE application
- Diff between the output file of the native application and the output file produced by running the SCATE application.

All these test produce a status like 'passed' or 'failed'. The outcome of the *scate _test* script can be visualized in the form of an html page. As stated above this file could be mailed to all developers after running a regression test.

Tips CVS cannot handle symbolic links. This can be circumvented by using the following trick. One can collect links using the unix *find* command and tar them. Such a tar file can be handled and checked in into CVS, allowing one to recreate an environment which makes use of symbolic links by simply uncompressing the tarred file. One word of advice is to not use absolute paths, instead revert to relative paths when possible. Even more useful is to agree on a set of environment variables that can be used to e.g. define the root directory of the *SCATE* tool.

10.3 The cvsDiff filter script

Running a 'cvs diff' over a test directory (potentially) produces a lot of output. Some of the output produced can, however, be safely ignored. Therefore, a script called *cvsDiff* is created that is able to filter the output of a 'cvs diff' command[21]. It removes differences due to changes in time stamps in the header of a file, moreover it also removes all differences that occur for a set of predefined files. This allows one to focus on *real* and more important changes in the source code. The script is quite flexible, and accepts normal 'cvs diff' options as well. This means that the script can also be applied on a single file or for instance on the current directory only.

[21] Note no information is lost, since the actual 'cvs diff' output can still be inspected.

11 Conclusions and future work

11.1 Conclusions

In this document we present a high level view on the Triple-M software infrastructure for YAPI and TTL source code analysis and transformations[22]. The information presented is relevant for everyone who wants to use the infrastructure, understand existing transformations, or implement transformations at the source code level. Note, this document is not intended as a user guide.

The focus of this document is on the *SCATE* tool, besides also the software automation tools - i.e. *C3PO* and *ANTLR* - and the back-end tool *MPC* are discussed in some detail. The shear size of these tools make it virtually impossible to give a very detailed description. Hopefully, the granularity of detail presented here is well chosen in order to be useful to the reader.

11.2 Future work

11.2.1 Generic C++ / C transformation infrastructure

In general, the YAPI to TTLC++ transformations can be performed quite efficiently. However, converting C++ to C proved to be quite elaborate. This is due to the fact that, while the C++ to C++ transformations are reasonably local modifications - at statement level, or addition of new statements - C++ to C conversion is much more global. Here, statements have to be moved across different files, placed into different contexts, and member function parameters need to be changed which influences all member function declarations, definitions and their calls. To improve on this the tool needs more transformation power. The following two issues should be investigated,

- Research for an efficient generic transformation infrastructure. What kind of representation is needed? How should the transformation API look like?
- Research possibilities to interface with a dedicated transformation framework such as Stratego [18].

11.2.2 Legacy code transformation

Currently not many people are using the YAPI programming style. One reason being the availability of a large non YAPI like code base. Having transformations in place that could, at least semi-automatically, convert any style of C / C++ code into a YAPI or TTL (CB) system would be extremely useful.

[22] A tool developer can find more information on implementation details in the CHANGES file that is located at the root of the *SCATE* source tree. The grammar files (*.G) - found at the same location - are also very instructive about implementation details.

11.2.3 GUI

There is currently no user interface available. In a real release this should be included. Here a list of the functionality that would be required:

- A graphical language should be in place that reflects a YAPI system and a platform architecture.
- Graphically create and modify (existing) process network models:
  - Enable plug and play of existing models at the YAPI level.
  - Make links between graphical view, source code and mapping files.
  - Switch between dotty like views and sequoia views.
- A graphical user interface should be in place that is able to visualize the C++ code, and let the user specify transformations interactively. With a point and click interface transformations should be initiated and the result visualized.
- Select multiple regions in source code on which a particular transformation needs to be specified; as a replacement for the SCOPE construct.
- Specify transformations on the process network using radio buttons and the like.
- Abstract from invariants when possible.
- Enable iterations starting from a functional level down to implementation level; i.e. manage directory structures and deal with version management.

11.2.4 API conversion

Currently support is incorporated to convert a process network from one API into another API. Examples of this are conversions from YAPI to TTL, YAPI to TTLC, and TTL to TTLC. Similarly one can think of a (semi) automatic conversion to suitable input code for behavioral synthesis. A detailed case study would be needed to provide the necessary input for such a conversion.

A C3PO generated API

A.1 Member functions for the whole class

Constructors Two constructors are available, the default *rBase(void)* constructor, and a *rBase(parameter, ..)* constructor that initializes the scalar members of the class by means of its parameters.

Copy Constructors Two copy constructors are available to create a new object of a class from either a reference or a pointer,

```
rBase(const rBase& src)
rBase(rBase* const src)
```

Copy functions Two copy functions are available, a deep copy function and a copy function that does deep copying only to a certain *level* and shallow copying afterwards. If *level* is zero, then the function is a shallow copy function.

```
rBase * deepCopy(void)
rBase * copy(const int level = 0)
```

Assign functions Assign functions are just like the copy functions, with the exception that they 'copy in place'. So instead of creating a new object from the *this* object, the members of the *this* object are assigned from the *src* object.

```
void deepAssign(rBase* const src)
void assign(rBase* const src, const int level = 0)
```

Implicit cast operators Two cast operators are available to convert the object to either its this pointer or its const this pointer,

```
operator rBase * const() const
operator rBase *()
```

Compare functions Two compare functions are available, the *eqSelf* function, to compare the whole object based on the value of the *this* pointer, and the *Eq* function that does a member wise comparison.

```
bool eqSelf(rBase* const r)
bool Eq(rBase* const r)
```

dump functions The dump functions print the contents of the object, the values of its members. Two variants are available, the *dump* function, that prints to the output stream *ostream*, and the *dumpToString* function that prints to a string. The *showEmpty* argument of the dump function controls whether undefined member values should be shown.

```
dump(const string& Offset = "",
     ostream& os = cout,
     const bool showEmpty = false)

const string & dumpToString(void)
```

operator<< functions The operator stream functions are wrappers around the dump function. Two variants are available, one to stream from a pointer to the object, and another to stream the object itself.

```
friend ostream & operator<<(ostream os, rBase* const r)
friend ostream & operator<<(ostream os, const rBase r)
```

accept functions The accept functions are part of the standard visitor design pattern (as described in references [12] page 331–344, and [13] page 944–952), they call the appropriate visit methods of the Visitor class. Two variants are available, the standard accept method, and an extended one that passes down the *VisitorAttributes* object.

```
void accept(Visitor* const v)
void accept(Visitor* const v, VisitorAttributes* const Attr)
```

free functions The free functions, free up allocated storage for the members of the object. This is done by calling the destructor functions, *delete* on the members and setting the member values to invalid. Two variants are available, the *Free* function that recursively frees up members to a certain *level*, and the *deepFree* function that frees up everything.

```
void Free(const int level = 0)
void deepFree(void)
```

the destructor function The destructor function is a wrapper around the free functions. To control the freeing behavior of the destructor, and because the destructor function cannot have any parameters, a special member variable of type *destroyinfo* is available in each class. The destroyinfo can be set to either *Deep* or to a certain *level* by means of the *destroyinfo::setDeep(const bool deepMbrIn)* and *destroyinfo::setLevel(const int levelIn)* member functions. When set to deep, the destructor calls the *deepFree* function, otherwise the *Free* function is called, and the *level* parameter is passed down from the *destroyinfo* member variable.

`~rBase(void)`

A.2 Member functions for scalar members

For each scalar member of type *T* and name *n* the following functions are generated,

```
rBase * setN(const T nIn) : Set the value of the member
T getN(void)              : Returns the value of the member
bool eqN(const T nIn)}    : Compares the value of the member
```

The following functions are conditionally generated,

```
bool hasN(void)
```

this function returns true when the value of the member is valid. It is generated if the member type is a pointer or the generated member variable has a specified default value. The latter is accomplished by appending

```
'=' '[' default value ']'
```

to the grammatical element notation of the member variable. So, for example

```
 = [-1]
```

```
bool eqSelf(const T nIn)
```

This function is only generated if the member type is a pointer, and does a pointer based comparison.

```
T endOfRecursion(void)
```

This function is synthesized for *linked list* members. Which means that $T = rBase$ or a sub class of $rBase$. The function returns the last element in the linked chain.

A.3 Member functions for collection members

The following summarizes the synthesized member functions for the collection members. The name of the member function consists of a base name, and optional prefix and postfix parts, symbolically placed within the {...} braces. The prefix and / or postfix parts are automatically calculated and added when the member function name needs to be more unique. The member name *n* is assumed to be *array*. The *Tmbr* type denotes the type of the collection member such as $vector < T >$, *T* denotes the type of the container element and *Tkey* the type of the hash map key.

Creation function Creates a new rBase object with one element in the collection member.

```
// sequential container
static rBase *
{rBase}NewWithElement{InArray}(const T element)

// hashed container
static rBase *
{rBase}NewWithElement{InArray}(const T element,
                               const Tkey const key)
```

Comparison function

```
bool eqArray(rBase* const r)
```

Member access function

```
Tmbr & getArray(void)
```

Validity function returns true if the container member is valid and not empty

```
bool hasArray(void)
```

size functions

```
unsigned int size{OfArray}(void);  // current size
void setSize{OfArray}(const unsigned int size);
bool isEmpty{Array}(void)
```

add element functions

```
// sequential variant,
void addFirst{ToArray}(const T t) // add to beginning

void add{ToArray}(const T t)      // add to end
rBase & operator+=(const T t)     // operator variant

// hash map variant
void add{ToArray}(const T t, const Tkey const key)
```

append functions Appends the contents of the array member of the src object to the end of the array member of this object.

```
// both sequential and hashed variants
void append{Array}(rBase* const src)

rBase & operator+=(rBase* const src) // operator variant
```

Index functions

```
Tmbr::iterator index{Array}(const int position) // pos to iterator
int index{Array}(Tmbr::iterator it)             // iterator to pos

Tmbr::iterator begin{OfArray}(void)  // iterator at begin
Tmbr::iterator end{OfArray}(void)    // iterator at end
```

put functions These functions place an element at the specified position into the array. When the array already contains an element on that position, that element is removed from the array. For the hashed container variant, this function is called

insert for reasons of compatibility with the STL naming convention, although it really behaves as a put function.

```
// sequential variant
void put{IntoArray}(const T t,
                    Tmbr::iterator it)
void put{IntoArray}(const T t,
                    const unsigned int position)

// hashed container variant
void insert{IntoArray}(const T t,
                       const Tkey const key)
```

insert functions This function is only synthesized for the sequential variant and inserts an element at the specified position, optionally moving all other elements including the one at the specified position one position to the right.

```
// sequential container variants
void insert{IntoArray}(const T t,
                       Tmbr::iterator it)
void insert{IntoArray}(const T t,
                       const unsigned int position)
```

element access functions

```
\\ sequential variant

T firstElement{OfArray}(void)

T elementAt{Array}(Tmbr::iterator it)
T elementAt{Array}(const unsigned int position)

T lastElement{OfArray}(void)
```

```
// hashed variant

pair<const Tkey const, T> * firstElement{OfArray}(void)
T firstValue{OfArray}(void)
const Tkey const firstKey{OfArray}(void)

pair<const Tkey const, T> *
elementAt{Array}(Tmbr::iterator it)

pair<const Tkey const, T> *
elementAt{Array}(const unsigned int position)

T valueAt{Array}(Tmbr::iterator it)
T valueAt{Array}(const unsigned int position)

const Tkey const keyAt{Array}(Tmbr::iterator it)
const Tkey const keyAt{Array}(const unsigned int position)

pair<const Tkey const, T> * lastElement{OfArray}(void)
T int lastValue{OfArray}(void)
const Tkey const lastKeyOf{Array}(void)
```

Enumeration function This function returns a 'Java style' Enumeration type

```
TmbrE<Tmbr>
```

re-implemented in C++.

```
// both sequential and hashed container variants
TmbrE<Tmbr> & getElements{OfArray}(void)
```

selection functions

```
// both sequential and hashed variants
// return a new object with its array member filtered
// by a selection function
rBase * select{FromArray}(bool (*sel_f) (T),
                          bool state = true)

// hashed variants
T findElement{InArray}(const Tkey const key)
T & operator[](const Tkey const key) // operator variant
```

erase functions

```
// both sequential and hashed variants
void remove{FromArray}(Tmbr::iterator it)
void remove{FromArray}(const unsigned int position)

void clear{Array}(void) // remove all elements

// additional functions for the sequential variant
T popFirst{FromArray}(void)
T popElement{FromArray}(const unsigned int position)
T popLast{FromArray}(void)

// additional functions for the hashed variant
int remove{FromArray}(const Tkey const key) // return number of
                                            // removed elements

pair<const Tkey const, T> * popFirst{FromArray}(void)
T popFirstValue{FromArray}(void)
const Tkey const popFirstKey{FromArray}(void)

pair<const Tkey const, T> *
popElement{FromArray}(const unsigned int position)

T popValue{FromArray}(const unsigned int position)
const Tkey const popKey(const unsigned int position)

pair<const Tkey const, T> * popLast{FromArray}(void)
T popLastValue{FromArray}(void)
const Tkey const popLastKey{FromArray}(void)
```

A.4 The Enumeration datatypes

The Enumeration datatypes are templated C++ classes that mimic the behavior of the java Enumeration type. See reference [13] page 337 and section 4.1. The collection member function *getElements()* returns an instance of an appropriate Enumeration type.

```
#ifndef Enumeration_INCLUDED
#define Enumeration_INCLUDED

template<class T, class Tmbr>
class ForwardEnumeration
{
   public:
   ForwardEnumeration(Tmbr * mbrIn)
   : it(mbrIn->begin()),
     mbr(mbrIn),
     cachedBegin(mbrIn->begin()),
     cachedEnd(mbrIn->end())
   {
   }

   bool hasNext()                          {return (it != getEnd());}
   typename Tmbr::value_type getNext()     {return *(it++);}
   typename Tmbr::value_type getCurrent() {return *it;}

   void reset() {it = getBegin();}
   void sync()  {it = T::advance(getBegin(),
                                 ::distance(cachedBegin, cachedEnd));
                 cachedBegin = getBegin();
                 cachedEnd   = getEnd();
                }

   protected:
      typename Tmbr::iterator getBegin() {return mbr->begin();}
      typename Tmbr::iterator getEnd()   {return mbr->end();}
      typename Tmbr::iterator it;

   private:
      Tmbr *mbr;
      typename Tmbr::iterator cachedBegin;
      typename Tmbr::iterator cachedEnd;
};
```

```
template<class T, class Tmbr>
class BiDirEnumeration : public ForwardEnumeration<T, Tmbr>
{
   public:
   BiDirEnumeration(Tmbr * mbrIn)
   : ForwardEnumeration<T, Tmbr>(mbrIn) {}

   bool hasPrev()
   {return (BiDirEnumeration<T, Tmbr>::it !=
            BiDirEnumeration<T, Tmbr>::getBegin());}

   typename Tmbr::value_type getPrev()
   {return *(--BiDirEnumeration<T, Tmbr>::it);}

   void setEnd()
   {BiDirEnumeration<T, Tmbr>::it =
    BiDirEnumeration<T, Tmbr>::getEnd();}
};
```

```
template<class T, class Tmbr>
class MapEnumeration : public BiDirEnumeration<T, Tmbr>
{
   public:
   MapEnumeration(Tmbr * mbrIn)
   : BiDirEnumeration<T, Tmbr>(mbrIn) {}

   typename Tmbr::reference getNext()
   {return *(MapEnumeration<T, Tmbr>::it++);}

   typename Tmbr::reference getCurrent()
   {return *MapEnumeration<T, Tmbr>::it;}

   typename Tmbr::reference getPrev()
   {return *(--MapEnumeration<T, Tmbr>::it);}

   typename Tmbr::data_type getValue()
   {return T::getValue(&getCurrent());}

   typename Tmbr::key_type  getKey()
   {return T::getKey(&getCurrent());}

   typename Tmbr::data_type getNextValue()
   {return T::getValue(&getNext());}

   typename Tmbr::key_type  getNextKey()
   {return T::getKey(&getNext());}

   typename Tmbr::data_type getPrevValue()
   {return T::getValue(&getPrev());}

   typename Tmbr::key_type  getPrevKey()
   {return T::getKey(&getPrev());}
};
```

```
template<class Tmbr>
class RandomEnumeration
{
   public:
   RandomEnumeration(Tmbr * mbrIn)
   : mbr(mbrIn),
     index(0)
   {
   }

   bool hasNext() {return (index < mbr->size());}
   bool hasPrev() {return (index > 0);}
   typename Tmbr::value_type getCurrent() {return (*mbr)[index];}
   typename Tmbr::value_type getNext()    {return (*mbr)[index++];}
   typename Tmbr::value_type getPrev()    {return (*mbr)[--index];}
   void reset()  {index = 0;}
   void setEnd() {index = mbr->size();}
   void sync()   {}

   private:
      Tmbr *mbr;
      unsigned int index;
};
```

```
template<class Tmbr>
class BufferEnumeration
{
   public:
   BufferEnumeration(Tmbr * mbrIn)
   : origMbr(mbrIn), mbr(*mbrIn)
   {
   }

   virtual ~BufferEnumeration() {}

   bool hasNext()                          {return !mbr.empty();}

   virtual typename Tmbr::value_type getCurrent() = 0;

   typename Tmbr::value_type getNext()
   {
     typename Tmbr::value_type t = getCurrent();
     mbr.pop();
     return t;
   }

   void reset()  {mbr = *origMbr;}

   void sync()   {reset();}

   protected:
   Tmbr *origMbr;
   Tmbr mbr;
};
```

```
template<class Tmbr>
class StackEnumeration : public BufferEnumeration<Tmbr>
{
   public:
   StackEnumeration(Tmbr *mbrIn) : BufferEnumeration<Tmbr>(mbrIn) {}

   typename Tmbr::value_type getCurrent()
   {return StackEnumeration<Tmbr>::mbr.top();}
};
```

```
template<class Tmbr>
class PriorityQueueEnumeration : public StackEnumeration<Tmbr>
{
   public:
   PriorityQueueEnumeration(Tmbr *mbrIn)
   : StackEnumeration<Tmbr>(mbrIn) {}
};



template<class Tmbr>
class QueueEnumeration : public BufferEnumeration<Tmbr>
{
   public:
   QueueEnumeration(Tmbr *mbrIn)
   : BufferEnumeration<Tmbr>(mbrIn) {}

   typename Tmbr::value_type getCurrent()
   {return QueueEnumeration<Tmbr>::mbr.front();}
};

#endif
```

A.5 Member functions for enumerated types

The following shows the member functions for the color type

```
enum color : RED | GREEN | BLUE;
```

enum object functions These functions return a static instance of the enumerated type, set to one of its values or to *UNDEFINED*

```
static const color* RED(void)
static const color* GREEN(void)
static const color* BLUE(void)
static const color* UNDEFINED(void)
```

compare function This function compares the input enumerated value *c* with the one of *this* object, and returns true if they match.

```
bool Eq(const color* c) const
```

is functions The *is*{*RED*, *GREEN*, *BLUE*} functions return true if the *this* object has the enumerated value *RED*, *GREEN* or *BLUE* respectively. The *isValid* function returns false on *UNDEFINED* and true otherwise.

```
bool isRED(void} const
bool isGREEN(void) const
bool isBLUE(void) const
bool isValid(void) const
```

parse function The parse function converts a string to an enumerated color object. The string can either be the name of the associated token / terminal or its associated *pattern*. This pattern must be a string constant, so it can be statically analyzed. For example, when the rule

```
void RED : ``rood'';
```

is added, the token *RED* is also associated with the pattern *"rood"*. The parse function will now return the *RED* color object on either the string *"RED"* or *"rood"*. In case of no matching string, the color object *UNDEFINED* is returned.

```
static const color* parse(const string& s)
```

toString Functions Two functions are provided, the *dumpToString* function that returns the associated name of the terminal / token, and the *toString* function that returns either the associated *pattern* or the name of the terminal / token if no pattern has been defined.

```
const string& dumpToString(void) const
const string& toString(void) const
```

the dump function This is a wrapper around the *dumpToString* function.

```
void dump(const string& Offset = "",
          ostream& os = cout) const
```

operator<< functions There are wrappers around the *dump* function.

```
friend ostream & operator<<(ostream& os,
                            const colorBase* c)
friend ostream & operator<<(ostream& os,
                            const colorBase& c)
```

A.6 Member functions for the validity layer

```
bool isValid(void)        // checks validity of this object
static bool isValid(T* t) // checks validity of object t
static T* const getNull(void) // returns the null object for type T
```

A.7 ANTLR parse function descriptions

For example the rule,

```
 R : INT->first '(' (INT->array)+ ')' R->next
   ;
```

is converted to,

```
parseR[R *rIn = R::getNull()] returns [R *r]
{
   if (R::isValid(rIn))
      r = rIn;
   else
      r = new R();

   int first = r->getFirst();
   R *next = r->getNext();
   initParseInfo(r->getParseInfo());
}
 :
   (
      tok:INT {first = (int) strtol((text) tok->getText().c_str(),
                                    NULL, 0);}
      "("
      (
         options
         {
            greedy = true;
         }
         :
         tok:INT {r->addToArray((int) strtol((text)
                                             tok->getText().c_str(),
                                             NULL, 0));}
      )+
      ")"
      next = parseR[next]
   )
   {
      r->setFirst(first);
      r->setNext(next);
      finishParseInfo(r->getParseInfo());
   }
;
```

A.8 The parseinfo datatype

The parseinfo datatype has the following fields,

```
bool errorMbr; // true in case of parse error

string startParseItem; // the name of the file or of the string buffer
                       // where the input for the rule begins.
int startLine;
int startCol;
int startPos; // start position in the
              // input stream

string endParseItem; // the name of the file or of the string buffer
                     // where the input for the rule ends
int endLine;
int endCol;
int endPos; // end position in the
            // input stream

string parseString; // the exact piece of input
                    // that corresponds with
                    // the parsed rule
```

A.9 The root parse functions

For each root rule, the following root functions are synthesized as member functions of the parser class. If *rIn* is a null object, a new object will be created and returned by these functions. Otherwise, the *rIn* object is augmented and returned.

```
static R * parseRFromFiles(nameset* const filenames,
                           R* rIn);
static R * parseRFromStrings(nameset* const strings,
                             R* rIn);
static R * parseRFromString(const char* string,
                            R* rIn);
static R * parseRFromStream(istream& is,
                            R* rIn);
static R * parseRFromInputStream(R* rIn);
```

B The YAPI / TTL representation

B.1 Structure

The structure consists roughly of the following datatypes.

unit representation of a unit

unitset a set of units

port representation of a port of a unit

portset a set of ports

instance an instance of a unit

instanceset a set of instances

instacePort port of an instance, the instantiated port of the instantiated unit

instancePortset a set of instance ports

channelInstance an instance of a channel

channelInstanceSet a set of channel instances

net Representation of a connection. Global nets connect a port of the parent unit with instance ports. Local nets are associated with a channel instance and connect the channel instance to instance ports.

netlist a set of nets

B.2 Functionality

The functionality representation consists roughly of a set of C++ AST subtrees. Each subtree represents a part of the functional code associated with a unit. Each subtree is available as a member of a unit. Roughly, the following members are available,

ExternalDeclarationSet preClassItems code that precede the class definition for this unit as encountered in the header file.

MemberDeclarationSet extraPublicMemberVariables public class member variables that are not InPort, OutPort, ChannelInstance, and process (Network) instances

MemberDeclarationSet extraPublicMemberFunctionDecls extra public member function declarations inside the class

MemberDeclarationSet extraPublicMemberFunctionDefs extra public member function definitions inside the class

MemberDeclarationSet extraProtectedMemberVariables extra protected member variables

MemberDeclarationSet extraProtectedMemberFunctionDecls extra protected member function declarations

MemberDeclarationSet extraProtectedMemberFunctionDefs extra protected member function definitions

MemberDeclarationSet extraPrivateMemberVariables extra private member variables

MemberDeclarationSet extraPrivateMemberFunctionDecls extra private member function declarations

MemberDeclarationSet extraPrivateMemberFunctionDefs extra private member function definitions

MemberInitializerList extraPublicMemberInits extra public member initializers

MemberInitializerList extraProtectedMemberInits extra protected member initializers

MemberInitializerList extraPrivateMemberInits extra private member initializers

FunctionDirectDeclarator mainFunc the encountered main (member) function

FunctionDirectDeclarator typeFunc the encountered type (member) function

ExternalDeclarationSet postClassItems code that follows the class definition for this unit as encountered in the header file

ExternalDeclarationSet includeItems include statements as encountered in the source file

ExternalDeclarationSet sourceItems the other source code for this unit as encountered in the source file

SimpleDeclaration thisVarDecl the 'this ptr' variable for this unit to be used for conversions to C

StatementList extraConstructorStats extra statements for the constructor body

C Command line options

C.1 MPC related options

-mpc < .. > **(1 arg exactly)** The name of the executable of the MPC tool to run, default is ”mpc”

-srcRoot < .. > **(1 arg exactly)** the MPC -srcRoot option, default is defined by MPC (”../../Design”)

-srcLib < .. > **(1 arg exactly)** the MPC -srcLib option, default is the current working directory.

-rmObsolete remove obsolete MPC subdirectories

C.2 YAPI / TTL related options

-o < .. > **(1 arg exactly)** The name of the default api to translate to as specified in the ApiDefinition section

-srcSuffix < .. > **(1 arg exactly)** The source file suffix, default is ”cc”

-cppSuffix < .. > **(1 arg exactly)** The preprocessed output file suffix, default is ”cpp”

-cppIncludes < .. > ∗ **(any number of args)** -I options, without the -I prefix for the preprocessor only. Default is ”yapi-pseudo-include”

-extraDefines < .. > ∗ **(any number of args)** extra -D options, without the -D prefix

-extraIncludes < .. > ∗ **(any number of args)** extra -I options, without the -I prefix

-extraLibraries < .. > ∗ **(any number of args)** full paths to extra libraries

-ignoreMain Whether the global (testbench) main code should be ignored

-dfltMinChannelSize < .. > **(1 arg exactly)** The default value (int) to be used for the minimum channelInstance size

-dfltChannelSize < .. > **(1 arg exactly)** The default value (int) to be used for the actual channelInstance size

-dfltMaxChannelSize < .. > **(1 arg exactly)** The default value (int) to be used for the maximum channelInstance size

-origSrcRoot < .. > **(1 arg exactly)** Original source root directory of application (input) files for scate

-origExeName < .. > **(1 arg exactly)** Original source its executable name

-origExeArgs < .. > ∗ **(any number of args)** Original source its executable arguments

-channelInstanceStatus < .. > ∗ **(any number of args)** The yapi simulation output files, to get simulated channelInstance sizes from

C.3 options that control the transformation infrastructure

-removeUnusedPorts Remove unused ports

-removeUnusedChannels Remove unused channels

-noFuncTransform Disable functional transformations

-noStructTransform Disable structural transformations

-noTransform Disable both functional and structural transformations

-broadcastTransform Enable broadcast transformation

-s < .. > ∗ **(any number of args)** The transformation specification files in the transform language

C.4 Makefile related options

-setup Create initial setup files, implies mkMakeSystem

-mkMakeSystem Create makefile related files

-mkTestMakefile Create test makefile

-mkTopLevelMakefile Create toplevel makefile

-mkImportMakefile Create import makefile

D Install Directives

These install directives are unit specific. More information can be found in reference [8], section 3.6 and further.

D.1 Ports

Port attributes are specified according to reference [8], section 3.6.3,

```
IN|OUT
TYPE=".." # transport datatype
INIT=".." # expression
".." # port name
```

The optional INIT part is a new extension. The expression is either a member initializer expression or an assignment expression related to the port. For example a constant value. The port name will be the name of the constructor parameter.

D.1.1 Member declarations

These are the port related member variables of the YAPI / TTC++ class definition. Two *MPC* databases are available, one for the member names, and one for the member types. The databases relate the constructor parameter name to the member name and type respectively. The following shows a port name *p* being related to the member variable *p_n* of type *p_t*

```
DATABASE["MEMBER_NAMES"]
{
   LOCAL ITEM["p"] = "p_n"
   ...
}

DATABASE["MEMBER_TYPES"]
{
   LOCAL ITEM["p"] = "p_t"
   ...
}
```

D.1.2 portmaps

A portmap is a table that relates a port to a certain *id*.

```
DATABASE["PORTMAP"]
{
   LOCAL ITEM["p"] = "id"
   ...
}
```

D.1.3 port interfaces

The following relates a port with a specific YAPI / TTL interface

```
DATABASE["PORTINTERFACE"]
{
   LOCAL ITEM["p"] = "CbIn"
   ...
}
```

D.1.4 parameters

Ports that are parameters are specified according to the *PARAMETERS* directive, reference [8] section 3.6.3.3, page 57. The additional *CALL_BY_VALUE* directive specifies a parameter to be a non reference, reference [8] section 4.1.1, page 94.

```
LOCAL SET["PARAMETERS"]  = "p1", "p2"
LOCAL SET["CALL_BY_VALUE"] = "p2"
```

D.1.5 Template parameters

Ports that have a template transport datatype are configured by means of the *GET_SIG_TYPE* function, according to reference [8], section 3.6.3.5.

```
CELL SINK(T)
{
   IN TYPE=T "input"
   var T = GET_SIG_TYPE("input")

   . . .
}
```

D.2 The this variable

The *this* variable as needed for C++ to C conversions is configured by the following two directives,

```
LOCAL ITEM["THIS_VAR_TYPE"] = "TTLCMP3_Decode_Frm"
LOCAL ITEM["THIS_VAR_NAME"] = "MP3_Decode_FrmThisVar"
```

D.3 Code hooks

Code hooks represent pieces of code fragments that will be inserted at the right place in the *MPC* output. The value of an install directive can either be the code fragment itself, or it can be a file that contains the code fragment.

D.3.1 Extra member initializers

The extra member initializers are related to extra member variables of a class. So anything that is not a port, a channel or an instance of a unit.

```
LOCAL SET["EXTRA_PUBLIC_MEMBER_INITS"]    = "publicVar(3)"
LOCAL SET["EXTRA_PROTECTED_MEMBER_INITS"] = "protectedVar(true)"
LOCAL SET["EXTRA_PRIVATE_MEMBER_INITS"]   = "init_done(false)"
```

D.3.2 Extra Headers

These are all headers of all interfaces of the transform API that differ from the *MPC* output API standard headers.

```
GLOBAL SET["EXTRA_HEADERS"] = "cb_in_port.h",
                              "cb_out_port.h"
```

D.3.3 The include items file

This file contains the #include statements as found in the source files associated with the relevant unit.

```
LOCAL ITEM["INCLUDE_ITEMS_FILE"] =
   "MP3_Decode_Frm.includeItems.cc.MPC"
```

D.3.4 The source file

This file contains the other functional code for the unit. The constructor definition is omitted from this file because it is regenerated by *MPC*.

```
LOCAL ITEM["SOURCE_FILE"] = "MP3_Decode_Frm.cc.MPC"
```

D.3.5 Extra constructor statements file

This file contains any extra constructor statements, as found within the constructor body.

```
LOCAL ITEM["EXTRA_CONSTRUCTOR_STATS_FILE"] =
   "MP3_Decode_Frm.extraConstructorStats.cc.MPC"
```

D.3.6 The pre class file

This file contains any code that precedes the class definition.

```
LOCAL ITEM["PRE_CLASS_FILE"] = "MP3_Decode_Frm.preClass.h.MPC"
```

D.3.7 Extra member function declaration and definition files

These files contain any extra member function declaration and definitions. These are any extra member functions that are not part of the YAPI / TTL structure.

```
LOCAL ITEM["EXTRA_PUBLIC_MEMBER_FUNCTION_DECLS_FILE"] =
   "MP3_Decode_Frm.extraPublicMemberFunctionDecls.h.MPC"

LOCAL ITEM["EXTRA_PUBLIC_MEMBER_FUNCTION_DEFS_FILE"] =
   "MP3_Decode_Frm.extraPublicMemberFunctionDefs.h.MPC"

LOCAL ITEM["EXTRA_PROTECTED_MEMBER_FUNCTION_DECLS_FILE"] =
   "MP3_Decode_Frm.extraProtectedMemberFunctionDecls.h.MPC"

LOCAL ITEM["EXTRA_PROTECTED_MEMBER_FUNCTION_DEFS_FILE"] =
   "MP3_Decode_Frm.extraProtectedMemberFunctionDefs.h.MPC"

LOCAL ITEM["EXTRA_PRIVATE_MEMBER_FUNCTION_DECLS_FILE"] =
   "MP3_Decode_Frm.extraPrivateMemberFunctionDecls.h.MPC"

LOCAL ITEM["EXTRA_PRIVATE_MEMBER_FUNCTION_DEFS_FILE"] =
   "MP3_Decode_Frm.extraPrivateMemberFunctionDefs.h.MPC"
```

D.3.8 Extra member variables files

These files contain any extra member variable declarations

```
LOCAL ITEM["EXTRA_PUBLIC_MEMBER_VARIABLES_FILE"] =
   "MP3_Decode_Frm.extraPublicMemberVariables.h.MPC"

LOCAL ITEM["EXTRA_PROTECTED_MEMBER_VARIABLES_FILE"] =
   "MP3_Decode_Frm.extraProtectedMemberVariables.h.MPC"

LOCAL ITEM["EXTRA_PRIVATE_MEMBER_VARIABLES_FILE"] =
   "MP3_Decode_Frm.extraPrivateMemberVariables.h.MPC"
```

D.3.9 The post class file

This file contains any code that follows the class definition.

```
LOCAL ITEM["POST_CLASS_FILE"] = "MP3_Decode_Frm.postClass.h.MPC"
```

D.4 Main and Type member function declarations

The *CUSTOM* directive specifies which function contains 'custom' code. With custom is meant, that the default generation of this function by *MPC* is disabled. The values

assigned to this directive are either ”main” for the main member function, and ”type” for the type member function.

In addition, the C++ to C conversion needs the actual name of the main member specified as well. This is done by means of the *MAIN_MBR_FUNCTION* directive.

```
LOCAL ITEM["MAIN_MBR_FUNCTION"] = "MP3_Decode_Frm_main"

LOCAL SET["CUSTOM"] = "type",
                      "main"
```

D.5 Special inheritance

These directives allow for specification of other super-classes for the YAPI / TTL Process and Network classes. Public, Protected, Private and multiple inheritance is supported.

```
LOCAL SET["PUBLIC"]    = "PublicBase1", "PublicBase2", . . .
LOCAL SET["PROTECTED"] = "ProtectedBase1", "ProtectedBase2", . . .
LOCAL SET["PRIVATE"]   = "PrivateBase1", "PrivateBase2", . . .
```

D.6 MPC Driver

This directive specifies the *MPC* driver for the unit. The value for this directive corresponds with the *DRIVER* specification of the transform API. See section 6.4.7.

```
LOCAL ITEM["DRIVER"] = "TTLC"
```

D.7 Includes and Libraries

These directives specify the necessary -I options and libraries for the unit.

```
GLOBAL SET["INCLUDES"] = "-I$(TTLC_INC)", . . .

GLOBAL SET["LIBRARIES"] = "$(TTLC_LIB)/libttl.a", . . .
```

D.8 Processor mapping

These directives specify a specific processor mapping (6.3.2), together with its build configuration (6.5), and the processor dependent libraries for the unit.

```
DATABASE["PROCESSOR"]
{
   DATABASE["MIPS"]
   {
      CELLNAME = "MIPS"
      INSTNAME = "3"
      GLOBAL ITEM["BINDIR"] = "$(CAKEBIN)"
      GLOBAL ITEM["CXX"] = "mips-g++"
      GLOBAL ITEM["LD"] = "mips-ld"
      GLOBAL ITEM["AR"] = "mips-ar"
      GLOBAL ITEM["AROPTIONS"] = "clq"
      GLOBAL ITEM["RANLIB"] = "mips-ranlib"
      GLOBAL ITEM["LIBPOSTFIX"] = "Mips"
      . . .

      GLOBAL SET["LIBRARIES"] = "$(YAPI_LIB)/libyapi.a",
                                "libMips.a"
   }
}
```

E The make system

E.1 The Makefile-ScateSetup file

E.1.1 Mandatory customization

The mandatory configurable part of this file looks as follows:

```
01 ################################### Mandatory Part
02
03 SCATE = scate
04
05 # The mpc executable
06 MPC = mpc
07
08 # The name of the mpc source library
09 SRCLIB = $(shell basename `pwd`)
10
11 # The include paths without the -I prefix for the preprocessor
12 #For example the paths to the api (yapi/ttl/..) include headers
13 CPP_INCLUDES = '$$(YAPIROOT)/include'
14
15 # API specification
16 # use quotes, and an extra $ to preserve environment variables
17 APISPECIFICATION = '$$(TRIPLEMROOT)/share/api/yapi.dat'
18
```

Lines 1 up to 18 constitute the mandatory part. Here you configure the *SCATE* (line 3) and *MPC* (line 6) version you want to use. The default values listed here suffice unless one wants to do something special.

The variable SRCLIB in Line 9 specifies from where you will run the tool *MPC* in a later stage. Since *SCATE* will generate input for *MPC* in the current directory, the default in Line 9 suffices in most cases.

The variable CPP_INCLUDES in Line 13 specifies the include path(s) for the preprocessor. If you transform a YAPI program to a TTL program, then only the YAPI include directory is needed in order to generate preprocessed files. The current default value in *Makefile-ScateSetup* is not correct when the ED&T release version 1.3 of YAPI is used. You should then set CPP_INCLUDES to *'$$(YAPIROOT)/include/yapi'* including the quotes. The quotes and the double '$$' ensure that the file *Makefile-Scate* that is generated to run *scate* contains *$(YAPIROOT)/include/yapi* as include path rather than its evaluated version.

The variable APISPECIFICATION in Line 17 specifies files that define the API's needed during program transformation. When transforming from YAPI to TTLC++, two API def-

initions are required YAPI (*yapi.dat*) and TTL (*ttlc++.dat*), respectively. You can specify the absolute path or the relative path. We recommend a third alternative using the variable TRIPLEMROOT, i.e.,

```
# APISPECIFICATION = '$$(TRIPLEMROOT)/share/api/yapi.dat'
APISPECIFICATION = '$$(TRIPLEMROOT)/share/api/yapi.dat'  \
                   '$$(TRIPLEMROOT)/share/api/ttlc++.dat'
```

E.1.2 Optional customization

Next the optional part of the Makefile-ScateSetup,

```
19 ##################################### Optional Part
20
21 # The parent of the mpc source library
22 # SRCROOT = ../../Design
23
24 # Mapping files
25 # MAPPINGFILES = mapping.dat transform.dat
26
27 # Extra flags
28 # EXTRA_FLAGS = -removeUnusedPorts
29
30 # The default Api to translate to,
31 # this must match the name of one of the Api definitions
32 # DEFAULT_OUTPUT_API = YAPI
33
34 # Extra -D flags, without the -D prefix
35 # EXTRA_DEFINES = VERBOSE
36
37 # Extra -I flags, without the -I prefix
38 # use quotes, and an extra $ to preserve environment variables
39 # EXTRA_INCLUDES = '$$(VYAROOT)/include'
40
41 # Full paths to extra libraries
42 # EXTRA_LIBRARIES = '$$(VYALIBDIR)/libvya.a'
43
44 # Default minimum channel size
45 # DFLT_MIN_CHANNEL_SIZE = 1
46
47 # Default actual channel size
48 # DFLT_CHANNEL_SIZE = 2048
49
50 # Default maximum channel size
51 # DFLT_MAX_CHANNEL_SIZE = 2048
52
```

The variable SRCROOT in Line 22 specifies where *MPC* will generate the transformed program. We recommend this default setting such that your transformed program will be generated in *$WORKDIR/Design/MyApp*.

The variable MAPPINGFILES in Line 25 specifies those files that you want *SCATE* to use to transform your program.

```
# Mapping files
# MAPPINGFILES = mapping.dat transform.dat
MAPPINGFILES = '$$(TRIPLEMROOT)/share/mapping.dat' \
               '$$(TRIPLEMROOT)/share/transform.dat'
```

The variable EXTRA_FLAGS in line 28 specifies extra command line options that you want for *SCATE*. For instance, you can provide *-dottyView* such that *SCATE* produces dotty output representing the transformed program.

```
# Extra flags
# EXTRA_FLAGS = -removeUnusedPorts
EXTRA_FLAGS = -dottyView
```

The variable DEFAULT_OUTPUT_API in Line 32 specifies to which API you want to transform your program. The variable can take the values YAPI, TTL, and TTLC. We recommend not to use this variable, but rather to use the explicit mapping of interfaces which is explained in this document [23].

The variable EXTRA_DEFINES in Line 35 specifies extra preprocessing directives. You should exclude the -D, e.g,

```
# Extra -D flags, without the -D prefix
# EXTRA_DEFINES = VERBOSE

EXTRA_DEFINES = VERBOSE
```

will give -DVERBOSE as extra directive.

The variables EXTRA_INCLUDES in Line 39 and EXTRA_LIBRARIES in Line 42 specify extra include and library directories that you might need to compile and link your program. These include files are excluded from program transformation.

The variables DFLT_MIN_CHANNEL_SIZE, DFLT_CHANNEL_SIZE, and DFLT_MAX_CHANNEL_SIZE in Lines 45, 48, and 51 specify the default channel sizes that you want for you transformed program. There are two other ways to define channel sizes, see section 7.4 and 6.3.4.

[23] Due to a bug at the time of writing one has to set this variable to TTL when iterating over TTL process network models

E.1.3 Regression test Customization

Regression testing assumes the existence of a compilable, and runnable test application. The root directory where this application resides can be specified using the variable ORIG_SRC_ROOT in Line 57. Currently the assumption is made that a Makefile should be present in this directory. The default target (i.e. running 'make') should compile the application without any human intervention. The name of the executable that is produced should be specified using the variable ORIG_EXE_NAME in Line 60. In some cases extra arguments are required for running, this can be specified using the variable ORIG_EXE_ARG in Line 63. The optionally produced output is sometimes collected in a file, this can be specified using the variable NAME_OUTPUT_FILE in Line 66.

This input allows one to built regression test that can be run without any human intervention, see section 10 for further information.

```
53 ##################################### Test Related Part
54
55 # Original source root directory of application files
56 # (Makefile is assumed to be present as well)
57 # ORIG_SRC_ROOT =
58
59 # Original source its executable name
60 # ORIG_EXE_NAME =
61
62 # Original source its executable arguments
63 # ORIG_EXE_ARGS =
64
65 # Name of output file
66 # NAME_OUTPUT_FILE =
```

E.1.4 The Non configurable part

```
67 ######## Nothing needs to be configured beyond this line
68
69 SCATEFLAGS = -setup
70
71 ifdef SRCROOT
72    SCATEFLAGS += -srcRoot $(SRCROOT)
73 endif
74
75 ifdef MAPPINGFILES
76    ifdef APISPECIFICATION
77       SCATEFLAGS += -s $(MAPPINGFILES) $(APISPECIFICATION)
78    else
79       SCATEFLAGS += -s $(MAPPINGFILES)
80    endif
81 else
82    ifdef APISPECIFICATION
83       SCATEFLAGS += -s $(APISPECIFICATION)
84    endif
85 endif
86
87 ifdef EXTRA_FLAGS
88    SCATEFLAGS += $(EXTRA_FLAGS)
89 endif
90
91 ifdef SRCLIB
92    SCATEFLAGS += -srcLib $(SRCLIB)
93 endif
94
95 ifdef DEFAULT_OUTPUT_API
96    SCATEFLAGS += -o $(DEFAULT_OUTPUT_API)
97 endif
98
99 ifdef EXTRA_DEFINES
100    SCATEFLAGS += -extraDefines $(EXTRA_DEFINES)
101 endif
102
103 ifdef CPP_INCLUDES
104    SCATEFLAGS += -cppIncludes $(CPP_INCLUDES)
105 endif
106
107 ifdef EXTRA_INCLUDES
108    SCATEFLAGS += -extraIncludes $(EXTRA_INCLUDES)
109 endif
110
```

```
111 ifdef EXTRA_LIBRARIES
112    SCATEFLAGS += -extraLibraries $(EXTRA_LIBRARIES)
113 endif
114
115 ifdef DFLT_MIN_CHANNEL_SIZE
116    SCATEFLAGS += -dfltMinChannelSize $(DFLT_MIN_CHANNEL_SIZE)
117 endif
118
119 ifdef DFLT_CHANNEL_SIZE
120    SCATEFLAGS += -dfltChannelSize $(DFLT_CHANNEL_SIZE)
121 endif
122
123 ifdef DFLT_MAX_CHANNEL_SIZE
124    SCATEFLAGS += -dfltMaxChannelSize $(DFLT_MAX_CHANNEL_SIZE)
125 endif
126
127 ifdef ORIG_SRC_ROOT
128    SCATEFLAGS += -origSrcRoot $(ORIG_SRC_ROOT)
129 endif
130
131 ifdef ORIG_EXE_NAME
132    SCATEFLAGS += -origExeName $(ORIG_EXE_NAME)
133 endif
134
135 ifdef ORIG_EXE_ARGS
136    SCATEFLAGS += -origExeArgs $(ORIG_EXE_ARGS)
137 endif
138
139 ifdef NAME_OUTPUT_FILE
140    SCATEFLAGS += -nameOutputFile $(NAME_OUTPUT_FILE)
141 endif
142
143 ifdef MPC
144    SCATEFLAGS += -mpc $(MPC)
145 endif
146
147 SCATEFLAGS += -f
148
149 all::
150         $(SCATE) $(SCATEFLAGS)
```

E.2 The Make.SrcVars file

The initial file looks like,

```
SRCS = $(shell ls *.cc)
CPPS = $(SRCS:%.cc=%.cpp)
```

After load order generation the file is changed into something like this,

```
LOAD_FILE = JPEG.load
CPPS = $(shell cat $(LOAD_FILE))
SRCS = $(CPPS:%.cpp=%.cc)
```

where the name of the load file depends on the application at hand.

E.3 The Makefile-Test file

The *Makefile-Test* is generated by specifying the flag *-mkTestMakefile* in the *Makefile-ScateSetup*. Without any configuration of the 'Test Related Part' (see appendix E.1.3) the generated file looks like,

```
01 # This Makefile is generated by scate
02 # ----------------------------------------------------------------
03 # This Makefile defines targets to direct building, running and
04 # cleaning the native application and the transformed application,
05 # as well as for comparing results. This file should not hold the
06 # targets or commands to actually build applications,
07 # but invoke the application's own Makefile for that end.
08 #
09 # Modify the following targets of this Makefile (when applicable):
10 # - build-orig-app:             build the original application.
11 # - build-scate-app:            build the scate application.
12 # - generate-scate-app:         generate scate application.
13 # - run-orig-app:               run the original application.
14 # - run-scate-app:              run the scate application.
15 # - run-cmp:                    run comparison on ...
16 # - clean-orig-app:             clean original application .
17 # - clean-scate-app:            clean scate application .
18 # - clean-scate-intermediates:  clean scate intermediate files.
19 # - clean:                      clean all the above
20 #
21 # Note: To tell scate_test that target run-cmp must be used,
22 # uncomment target has-run-cmp.
23 #       If run-cmp is used, make sure run-orig-app and
24 #       run-scate-app use different result files!
25 #
26
27 build-orig-app:
28  cd ; \
29  $(MAKE); \
30  cd -;
31
32 build-scate-app:
33  cd ../../Design/ttl/TTL; \
34  ../../Compile; \
35  cd -;
36
37 generate-scate-app:
38  $(MAKE) -f Makefile-Scate clean; \
39  $(MAKE) -f Makefile-ScateSetup; \
40  $(MAKE) -f Makefile-Scate MAKELEVEL=0; \
41  $(MAKE) -f Makefile-Mpc;
```

```
42
43 run-orig-app:
44  cd ; \
45  ./ ; \
46  cd -;
47
48 run-scate-app:
49  cd ../../Design/ttl/TTL/TestBench/SetupModel; \
50  ./SetupModel; \
51  cd -;
52
53 has-run-cmp:
54
55 run-cmp:
56  diff -s ../../Design/ttl/TTL/TestBench/SetupModel/ /;
57
58 clean-orig-app:
59  cd ; \
60  $(MAKE) clean; \
61  cd -;
62
63 clean-scate-app:
64  cd ../../Design/ttl/TTL; \
65  $(MAKE) clean; \
66  cd -;
67
68 clean-scate:
69  $(MAKE) -f Makefile-Scate clean;
70
71 very-clean-scate:
72  $(MAKE) -f Makefile-Scate very_clean;
73
74 clean: clean-orig-app clean-scate-app clean-scate
75
76 very-clean: clean-orig-app clean-scate-app very-clean-scate
```

This file basically defines all the necessary targets to perform a regression test. However, the *Makefile-Test* as printed here is not operational. This would have required a well configured 'Test Related Part' of the *Makefile-ScateSetup*. A fully functional regression test suite has been defined in this manner that does not require any human intervention.

Of course one can always adopt the *Makefile-Test* file [24] manually when the assumptions made do not apply to your test example. However, one should realize that when cooperation with the *scate_test* script is required the default targets should be obeyed.

[24] The *Makefile-Test* is treated as overwrite protected.

F Example of a YAPI to TTL transformation

If you want to translate your program from YAPI to TTL, then you need to consider the following things. The environment variables *YAPIROOT* and *TRIPLEMROOT* should be known in the shell you use, and point to the correct paths.

One has to realize that the input is YAPI. Therefore, the include path for the preprocessor must point to the YAPI header files. The default value is fine unless one uses ED&T version 1.3 of YAPI. In this case one needs to adjust the *Makefile-ScottySetup* file to,

```
CPP_INCLUDES = '$$(YAPIROOT)/include/yapi'
```

You need to specify the correct API description files, i.e.,:

```
APISPECIFICATION = '$$(TRIPLEMROOT)/share/api/yapi.dat' \
                   '$$(TRIPLEMROOT)/share/api/ttlc++.dat'
```

or

```
APISPECIFICATION = '$$(TRIPLEMROOT)/share/api/yapi.dat' \
                   '$$(TRIPLEMROOT)/share/api/ttlc.dat'
```

You need to explicitly map YAPI FIFOs onto TTL Channels. To this end, you can modify the toplevel *.map* file generated by *SCATE* as described below. When you run *SCATE*, it produces mapping files called **.map*. If the class that you instantiate in your main function is called *TopLevel*, then *SCATE* will generate the file *TopLevel.map*. This file describes the toplevel network. You can modify it, as illustrated in the following example.

We use the producer-consumer example as distributed in the YAPI releases. The *SCATE* generated map file *PC.map* of this example look as follows:

```
MAPPING PC
{
   TRANSPORT
   {
      mem_fifo
      {
         Channel<int>
      }
   }
   NET
   {
      fifo -> mem_fifo
   }
   MAPPING Consumer : cons
   {
   }
   MAPPING Producer : prod
   {
   }
}
```

The map file describes a default template for transformation. It introduces a new transport medium called *mem_fifo* that is of type

```
Channel<int>
```

Then it maps the existing YAPI FIFO called 'fifo' onto *mem_fifo*. The consumer and producer remain the same. Currently only interface type CB is supported in a NET mapping section.

F.1 TTLC++

For transformation to TTLC++, you need to copy the map file, for instance, to *PC2TTL.map*. Remember that *SCATE* regenerates the file *PC.map* each time you run it. So you need to make a copy to avoid losing your changes during an iteration.. Next, make the following changes to the transport section:

```
TRANSPORT
{
   mem_fifo[100]
   {
      TTL::Channel<int> USING CbOut, CbIn
   }
}
```

The value 100 between square brackets specifies the size of the channel in number of tokens. So in this case, the channel can contain 100 integers.

The prefix 'TTL::' indicates - similar to a namespace in the C++ programming language - in which API the *Channel* definition can be found (see API definition file 'ttlc++.dat'). If you only use one API definition in your "Makefile-ScateSetup" or if *Channel* is uniquely defined within the APIs used, then the prefix is not needed.

The USING clause specifies the interfaces of the channel *mem_fifo*.

F.2 TTLC

For transformation to TTLC, the *PC2TTL.map* looks as follows:

```
TRANSPORT
{
   mem_fifo[100]
   {
      TTLC::ttlChannel<int> USING CbOut, CbIn
   }
}
```

F.3 Makefile-ScateSetup configuration

Mapping files can be added to the MAPPINGFILES variable in the *Makefile-ScateSetup* file, in order for *SCATE* to pick them up i.e,

```
MAPPINGFILES = ./PC2TTL.map
```

You also need to consider the fact that the toplevel main function is not translated by *SCATE*. Edit the toplevel main function manually. To avoid that this main function is overwritten when you rerun *MPC*, you can add the additional flag *-ignoreMain* to the EXTRA_FLAGS variable, for instance,

```
EXTRA_FLAGS = -ignoreMain
```

F.4 Recapitulation

Based on the instructions above one should be able to perform the following set of transformations:

YAPI to YAPI comprises a file and directory restructuring of the initial application. Note, this file and directory restructuring holds for every API-to-API transformation performed by the Triple-M tooling.

YAPI to TTLC++ comprises the transformation to the CB interface type of TTL. Since TTL does not support infinite buffer sizes various methods are supported to fix the buffer size in the transformed application. Note, this mapping of buffer sizes can be applied in every API-API transformation.

YAPI to TTLC comprises the transformation to the CB interface type of TTLC. Since the C-language does not support the notion of a class, transformations are applied to functionally equivalent C-language constructs like structs, while member functions are transformed to global functions. A number of other transformations are supported as well; e.g. all declarations are moved to the beginning of the scope.

TTLC++ to TTLC++ can handle all the general functional transformations mentioned above. This transformation demonstrates that the input API does not necessary needs to be YAPI, but in fact it is a configurable API. The construction of the network should however be done in a YAPI like manner, otherwise the network would not be recognized.

TTLC++ to TTLC is a transformation where TTLC++ is handled as the input API, in combination with the translation of C++ to C. Both are described above, in TTL C++-to-TTL C++, and YAPI-to-TTL C, respectively.

G Symbol table initialization

With this feature, the symbol table can be initialized from a file before parsing of the C++ files. With such a preinitialized symbol table parsing goes much faster because the system files do not need to be parsed anymore. For example, when the functional run-time environment for YAPI or TTL is included, then also their implementations will be parsed. This has become an issue with gcc 3.x preprocessed files, because of the integration of the standard template library within the system files. Mostly in *iostream* and *fstream*.

To use this new feature, the option *-symtab symbol table files* is available. You can add this option to the extra flags in the *Makefile-ScateSetup* file.

```
EXTRA_FLAGS = -symtab '$$(TRIPLEMROOT)/share/symtab.yapi'
```

Upon each run, the *SCATE* tool generates a *symtab* file which can be used to pre initialize the symbol table in the next run. The symbol table keeps track of the (system) files it has been built with during the initial run, and these files are omitted from parsing in the *-symtab* mode in a similar way as the 'already included header files' (section 5.3). There are two classes of files that are detected, the system files and the input api (YAPI / TTL library header) files. The system files are matched with their full path names because these names are not unique over the different system file directories. The api files are only matched against the base name, which allows them to be skipped even in that case that the symbol table was initialized from another set of these files than the ones currently included in the *.cpp* file being analyzed. When a header file is encountered in the *.cpp* file that does not match with any of the symbol table file descriptions this file is completely parsed. To recognize the api files, the ApiDefinition sections (section 6.4) are used.

G.1 symtab file creation

Normally, *scate -cxxParseOnly main.cpp -s apideffile(s)* will create an adequate symtab file. This process is however a bit creative, because you need to pick up as many system files as possible. But, when this is successful, and when residing on the same platform, a symtab file can be reused over different applications because the included system files do not really change. Predefined symbol tables are available in the files *$(TRIPLEMROOT)/share/symtab.yapi* and *$(TRIPLEMROOT)/share/symtab.ttlc++* for YAPI and TTLC++ respectively

References

[1] E.A. de Kock, G. Essink, W.M.J. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Krijutzer, P. Lieverse, and K.A. Vissers. YAPI; Application Modeling for Signal Processing Systems. In *37th Design Automation Conference*, pages 402–405, 2000.

[2] http://sourceforge.net/projects/y-api/ .

[3] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing*, pages 471–475. North-Holland Publishing Co., 1974.

[4] Pieter van der Wolf, Erwin de Kock, Thomas Hendriksson, Wido Kruijtzer, and Gerben Essink. Design and Programming of Embedded Multiprocessors: An Interface-Centric Approach. Stockholm, Sweden, September 8–10 2004. CODES + ISSS.

[5] http://sourceforge.net/projects/scate/ .

[6] *Software Portability with imake*. Number ISBN: 1-56592-226-3. O'Reilly & Associates, Inc., 1996.

[7] http://sourceforge.net/projects/c3po/ .

[8] O Popp. YAPI network generation with the Multi Purpose Network Compiler (MPC) (might eventually be published here: http://sourceforge.net/projects/mpsc/). Technical report, July 2003.

[9] http://sourceforge.net/projects/mpsc/ .

[10] http://www.antlr.org/ .

[11] David R. Musser and Atul Saini. *STL tutorial and Reference Guide C++ programming with the Standard Template Library*. Addison Wesley, 1996.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[13] Bruce Eckel. *Thinking in JAVA*. Prentice Hall PTR, 1998.

[14] http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?Roskind+grammars .

[15] http://www.research.att.com/sw/tools/graphviz .

[16] http://www.win.tue.nl/sequoiaview/ .

[17] http://www.research.att.com/sw/tools/graphviz .

[18] http://catamaran.labs.cs.uu.nl/twiki/pt/bin/view/Stratego/WebHome .

Public Domain

Public Domain

Author(s) D. Alders, O. Popp

Title The Triple-M software infrastructure for YAPI and TTL source code analysis and transformations