Issued: March 1, 2006

Scate Manual

D. Alders

Authors' address data: D. Alders ; d.alders@sourceforge.net

:

Title: Scate Manual

Author(s): D. Alders

Part of project:

Customer:

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

Abstract: This document is a preliminary version of an user manual for the Triple-M tools. The installation and configuration of an unofficial release of the tooling is explained. After which the reader is guided through a number of transformations on process networks. All relevant source code is included in the manual. Furthermore, iterations over transformations are explained.

Conclusions: After reading the user should be able to install and configure the Scate tools. He /she should also be able to perform a number of source code transformations on process networks, and setup iterations over transformations with the help of version management. All this provides the reader hands on experience, which could lead to valuable feedback.

Contents

Distribution

1 Introduction

1.1 Objective

This document is an user manual for the tools developed within the Triple-M project. The objective of the tooling is to provide a system-level design and programming environment for embedded multiprocessors.

The reader is guided - in a step-by-step fashion - through a set of transformations on Producer-Consumer type of networks. All relevant input and output source code is included in this manual.

After reading, one should be able to conduct similar transformations as presented here on any hierarchical process network that can be described by the API description language of the tooling.

The relevance and power of the tooling described in this document should become apparent from the examples presented. One of the main objectives is to get feedback from potential users even though the tools are still in an early stage of development. Hopefully, the reader is inspired to propose new (relevant) transformations to further improve the design and programming environment.

1.2 Intended audience

This document is primarily intended for application designers and system designers of signal processing applications. The former specifies the functionality of the system, while the latter implements the functionality. Both types of jobs are supported by the design and programming environment described.

The models on which transformations can be applied are developed in the C / C++ language. Knowledge of the C language suffices to describe the functionalities of the processes. When applicable, knowledge of C++ is only required for the interfaces of the processes and the structures of the process network(s). Furthermore, we assume that the reader is familiar with Task Transaction Level modeling [1, 2, 3] in general.

The reader is strongly encouraged to go through all the source code examples presented, and to actually perform the transformations him / her self. This provides the reader with some valuable hands-on experience.

1.3 Outline of document

In chapter 2 we explain the installation and configuration process of the *SCATE* tool [4]. In chapter 3 we continue with several high level views on the tooling. An overview of the input and output files involved is presented as well. Next, Producer-Consumer examples are presented in chapter 4. All relevant source code fragments are included in the text. Finally, we end with troubleshooting in chapter 5.

2 Setup

An official release of the tooling is available at sourceforge [4]. In this chapter we describe the installation proces. Even though, the source code is also available at sourceforge [4], the installation process starting from sources is not described here.

2.1 Installation

Download a binary distribution of SCATE from sourceforge [4].

Choose a directory where you want to install the Triple-M tools, say '~/Triple-M'. Change into the directory and unpack the tarball:

```
tar fxzv scate-release-1.0.tgz
```

This will create a directory structure like:

```
~/Triple-M/release/ [version number/] bin/
                                       doc/
                                       examples/
                                       pc/
                                       scripts/
                                       share/    api/
                                                 include/
                                                 mapping/
                                                 symtab/
```

Furthermore, it is assumed that the runtime environment(s) of the process network(s) at hand are properly installed. A YAPI runtime environment can be obtained from sourceforge [2]. In this manual we also make use of another runtime environment called TTL. Unfortunately, TTL is only available to the Philips community. For the examples that require TTL alternative source code is recommended. In due time this manual may be rewritten to only include YAPI examples.

For a correct installation all steps described in section 2.2 need to be completed as well.

2.2 Configuration

The easiest and most flexible way to configure the Triple-M tooling is by defining environment variables. For some of these variables separate example scripts are provided. For example the various implementations of the functional runtime environment of YAPI can be selected by using the script:

- yapiInit

This script should be manually configured to reflect the directories used in your local environment. The environment variables are defined for the current shell. Thus these scripts should be activated like (with an optional argument):

```
. yapiInit [argument]
```

The definition of all the environment variables involved in the Triple-M tooling can best be collected in one file. Such a file can be executed during startup, e.g. via ".vueprofile" when using Linux. This would then statically define the environment variables for all shells at login. Another way of working is to collect them in a file that is executed before the tool is used, e.g. in an executable file. In this way the tooling versions with which a process network is transformed is automatically documented.

```
# exports $YAPIROOT
. yapiInit

# Directory where Triple-M release can be found
export TRIPLEMROOT=/path/to/release/ [version number/]
echo TRIPLEMROOT=$TRIPLEMROOT

# MPC version used
export MPC=$TRIPLEMROOT/bin/mpc-version-number
echo MPC=$MPC
```

As long as there is no official cadenv installation process defined for the Triple-M tools one has to manually add the bin and scripts directories of the release to the executable PATH.

```
export PATH=$PATH:$TRIPLEMROOT/release/ [version number/] bin/:
                  $TRIPLEMROOT/release/ [version number/] scripts/
```

2.3 Versions

The following versions are used and 'supported':

- TTL C++ tag v1_01.
- TTL C tag JK6.
- YAPI from sourceforge [2]
- MPC from sourceforge [5]

Assumptions made:

- Gcc version 3.x.
- Imake.

3 Overview

A high level view of the Triple-M tools is presented in Figure 1. There are basically two sets of input files. One set describes the function to be implemented that is captured in an executable process network structure. The other set contains the design decisions made by the system architect to efficiently implement the function. The tools apply 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 tools, or existing hardware and / or software compilers.

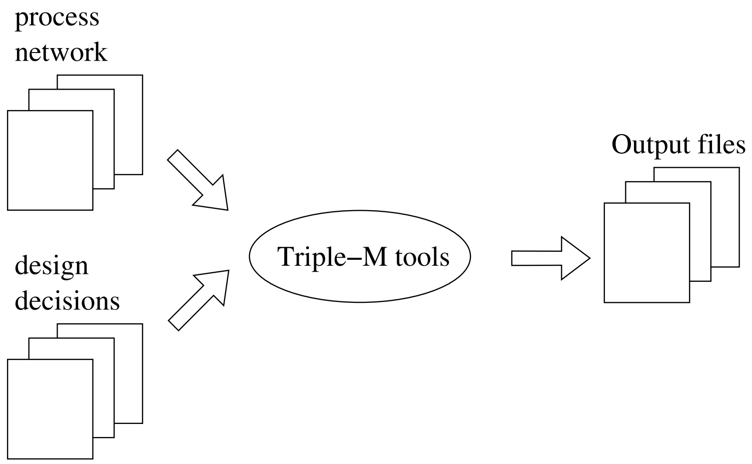

Figure 1: User view on the *Triple-M* tools.

The Triple-M tools are able to extract the process network from C++ source code - written in YAPI or TTL type syntax - , see reference [6] for more detailed information. Thereby, allowing the system architect to specify transformations on process networks, 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. This programming style is enforced even when the input files uses a different style. Furthermore, *Imake-files* [7] are generated such that the output can be readily compiled.

Another high level view on the *Triple-M* tools is shown in Figure 2. From the figure it is clear that the Triple-M tools consist in fact out of two seperate tools; *SCATE* (Source Code Analysis and Transformation Environment), and *MPC* (Multi Purpose Network Compiler), see section 3.2.

All C++ source code provided to the tools is first run through the *gcc* preprocessor. This has consequences for comments and macros in the source code. The former is simply removed [1], 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.

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

The *SCATE* tool internally separates the functional source code from the network structure. This results in two sets of intermediate output files [2] as shown. *MPC* combines these two sets and produces output files [3] where the functional source code and the network structure are combined again. The extraction of the network structure allows to 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.

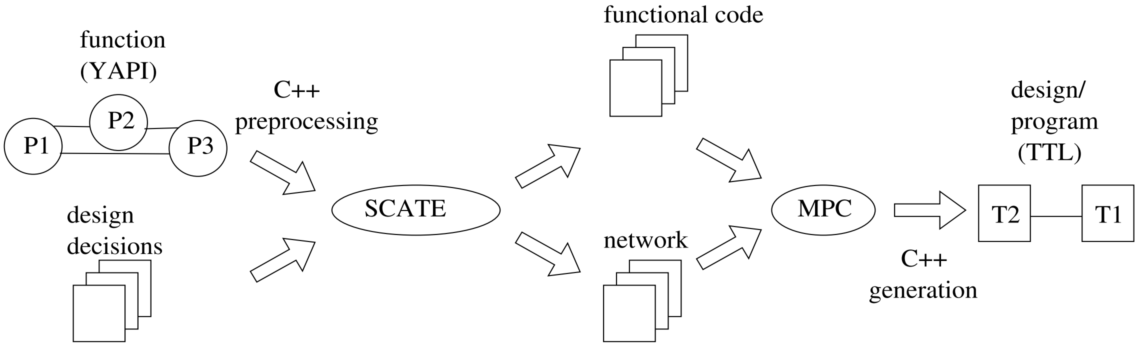

Figure 2: High level view on the flow of data through the *Triple-M* tools.

3.1 SCATE

The *SCATE* tool as described above can (internally) be further subdivided into several components as shown in figure 3. The arrows between the components indicate the flow of information through the tools.

The preprocessed 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 [8]. 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.

The tooling is able to recognize and extract the network and the member functions of YAPI and TTL. This is accomplished by providing a (programmable) description of these api's to the tooling which is generic enough to express current versions of both YAPI and TTL. In this way the tooling itself does not have to know the programmable details of those api's.

3.2 MPC

MPC is a tool for the generation of network descriptions [5]. It is multi purpose since it is intended to generate network descriptions in different languages or formats from the same input specification. Currently the following language drivers are available: C++ (YAPI / TTL), C (TTL), and VHDL.

[2] Intermediate files produced by *SCATE*: *.MPC, installfile.yapi, installfile.yapi.*, *.net.

[3] Intermediate files produced by MPC are not explicitly shown in the figure: installfile.*.yapi.

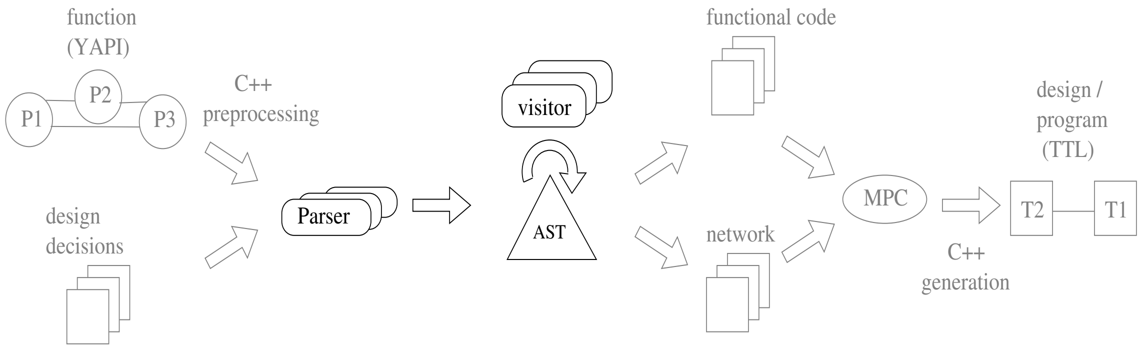

Figure 3: Component level view from a detailed user perspective.

MPC can be used as a front-end tool or - as is the case here - as a back-end tool. In the latter MPC also needs to cope with the functional part of source code. For this end several provisions were added in the form of installfiles after the publication of the MPC documentation [5]. The syntax of these additional installfiles are discussed in reference [6].

3.3 Getting started

It is assumed that an input process network description for *SCATE* has its own directory within a directory called 'Generate'.

The common way of working is to feed commands to the *SCATE* tool via a file called "Makefile-ScateSetup". An initial version of this file can be generated using the command:

```
scate -setup
```

The "Makefile-ScateSetup" file can be viewed upon as the toplevel "Makefile" in a series of related "Makefile"s, see figure 4. In each step a "Makefile-*" is generated that serves as input for the next step. At the left hand side the commands are shown, while at the right hand side the output files are shown. Makefiles between brackets are optional, those will only be generated when the appropriate flags are set in the "Makefile-ScateSetup".

At all times it is allowed to edit the toplevel "Makefile-ScateSetup", e.g. due to changing mapping requirements. By repeating the commands 2–4 shown in figure 4 all derived files will be updated. In this way additional changes will ripple through all files involved.

Let us for the moment take a closer look at the toplevel file. The mandatory part of the generated "Makefile-ScateSetup" looks like:

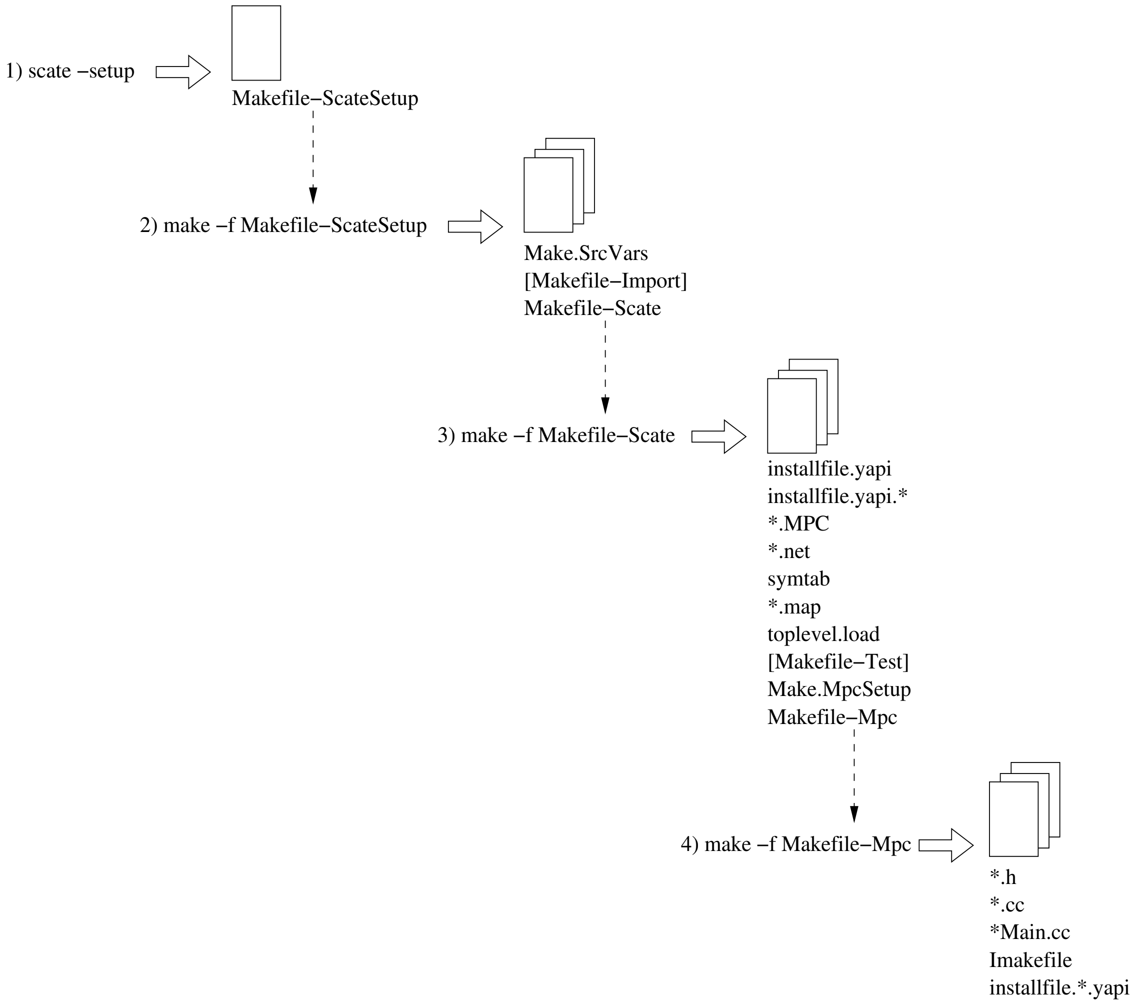

Figure 4: Makefile hierarchy.

```
##################################### Mandatory Part ###############

SCATE = scate

# The mpc executable
MPC = mpc

# The name of the mpc source library, for example the name of the yapi system being transformed
SRCLIB = $(shell basename `pwd`)

# The include paths without the -I prefix for the preprocessor only
# For example the paths to the api (yapi/ttl/..) include headers
# Use quotes, and an extra $ to preserve (environment) variables
CPP_INCLUDES = '$$(YAPIROOT)/include'

# API specification
APISPECIFICATION = '$$(TRIPLEMROOT)/share/api/yapi.dat'

```

Lines 01 up to 18 constitute the mandatory part. Here you configure the *SCATE* (line 03) and MPC (line 06) versions you want to use. The default value listed here for *SCATE* suffices unless one wants to do something special. In case MPC has been defined as an environment variable line 06 should be commented out.

The variable SRCLIB in line 09 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 09 suffices in most cases.

The variable CPP_INCLUDES in line 14 specifies the include path(s) for the preprocessor. The current default value in "Makefile-ScateSetup" is correct when you want to iterate over YAPI sources (provided the environment variable YAPIROOT has been properly defined). The quotes and the double '$$' ensure that the file "Makefile-Scate" that will be generated in the next step contains *$(YAPIROOT)/include* as include path rather than its evaluated version.

The variable APISPECIFICATION in line 17 specifies file(s) that define the API('s) needed during program transformation. When iterations over YAPI are performed, only the YAPI API definition is required (can be found in the file 'yapi.dat' provided in the (Triple-M release). There are several syntax styles possible here. One could specify the absolute path or the relative path of the file(s) involved. However, we recommend a third alternative, i.e., by defining the variable TRIPLEMROOT, which is the default.

The optional part of the generated "Makefile-ScateSetup" looks like:

```
###################################### Optional Part ###############

# The parent of the mpc source library
# SRCROOT = ../../Design

# Mapping files
# MAPPINGFILES = mapping.dat transform.dat

# Extra flags
# EXTRA_FLAGS = -removeUnusedPorts

# The default Api to translate to,
# this must match the name of one of the Api definitions in the
  spec files
# DEFAULT_OUTPUT_API = YAPI

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

# Extra -I flags, without the -I prefix
# use quotes, and an extra $ to preserve (environment) variables
# EXTRA_INCLUDES = '$$(VYAROOT)/include' '$$(CPFSPDROOT)/include'

# Full paths to extra libraries
# EXTRA_LIBRARIES = '$$(VYALIBDIR)/libvya.a' \
#                   '$$(CPFSPDLIBDIR)/libcpfspd.a'

# Default minimum channel size
# DFLT_MIN_CHANNEL_SIZE = 1

# Default actual channel size
# DFLT_CHANNEL_SIZE = 2048

# Default maximum channel size
# DFLT_MAX_CHANNEL_SIZE = 2048

###################################### Test Related Part ###########

# Original source root directory of application files (Makefile is
  assumed to be present as well)
# ORIG_SRC_ROOT =

# Original source its executable name
# ORIG_EXE_NAME =

# Original source its executable arguments
# ORIG_EXE_ARGS =

# Name of output file
# NAME_OUTPUT_FILE =

################### Nothing needs to be configured beyond this line
```

In principle one need not fill in anything here. However, due to an unsolved bug one needs to edit line 15 to fix the default output API, whenever this API is different from YAPI. So in case one iterates over TTL or translates to TTL one has to uncomment this line and fill in TTL:

```
DEFAULT_OUTPUT_API = TTL
```

We now assume for the moment the user is satisfied with all the options provided to the "Makefile-ScateSetup". Hereafter, one can generate the file "Makefile-Scate" by issuing the command (step 2 figure 4):

```
make -f Makefile-ScateSetup
```

As a result "Makefile-Scate" is generated. Execution of this "Makefile-Scate" will actually transform the input process network according to the transformations specified in the "Makefile-ScateSetup" (step 3 figure 4).

```
make -f Makefile-Scate
```

During every run all the source code files are parsed by the *SCATE* tool. Special provisions are made to garantee that the output sources made from this multi file AST can still be compiled in the single file compilation domain. This is acheived by using a specific order in which all the preprocessed files are read. This order is automatically determined and written into a load order file at the end of the very first run of the *SCATE* tool on a fresh process network. In a consecutive second run (does not require human intervention) this load order is used to not miss any information that could otherwise be lacking. Thus, in general one will experience two consecutive runs when starting with a fresh process network. Hereafter, one run is sufficient as long as the load order file is not removed.

As a result of the previous command a set of intermediate files have been generated [4]. These files can be composed into a compilable process network by running the generated "Makefile-Mpc" (step 4 figure 4).

```
make -f Makefile-Mpc
```

The process network output source code is generated in directory:

```
$SRCROOT/$SRCLIB/apiname/
```

by default SRCROOT will expand into '../../Design' (as defined in "Makefile-ScateSetup"), SRCLIB is determined by the basename of the current directory, and 'apiname' is currently either Yapi, TTL, or TTLC.

[4] The default behavior is that intermediate files are overwritten. In section 2 a flag is introduced that could prevent this.

4 Example transformations

In this chapter a number of source code transformation examples are explained in detail. If possible simple Producer-Consumer examples are used, see figure 5. By using small examples all (relevant) source code can be incorporated into the text.

In the Producer-Consumer examples we have two processes, a *producer* that writes integer values to a channel, and a *consumer* that reads the values from the channel. The number of values written is initialized by the network process *pc*.

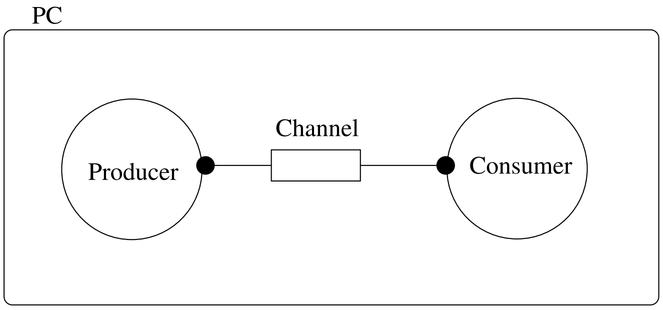

Figure 5: Producer-Consumer.

The source code of the initial process networks for section 4.1 (TTL), section 4.2 (YAPI) and section 4.3 (flattening) can be found in the release directory '~/Triple-M/release/examples/pc/'. Manually adjust (when needed) in each example the 'Makeflags' file to represent your local settings.

In the first example (section 4.1) we demonstrate the effects of a source-to-source transformation on the file and directory structure, which allways occurs even without specifying any particular transformation. This kind of restructuring of the source code is very usefull since it has the advantage that the output conforms to one programming style. One could now start collecting all the applications modeled as process networks and put them in one central database to be used by the Philips community at large. Browsing through such a database of applications that confirm to one programming style would make life easier.

In the second example (section 4.2) we demonstrate howto incorporate API transformations. The specification of a process network API is not hard coded in the tooling. Rather it should be provided to the tooling as (programmable) input. This allows one - to a certain extend - to specify the syntax and semantics of the input API and the output API. Currently the APIs of YAPI, TTL and TTLC can be captured.

In the third example (section 4.3) (partial) flattening of hierarchical process networks is explained. Hierarchy is very well suited at functional level, however at the implementation level it can lead to too much overhead. Therefore this kind of transformation is usefull when going from a functional model to an implemention model.

In the final example (section 4.4) we present a solution approach to deal with iterations over a number of related models. In particular we focus on the problem of late changes in the initial model that should ripple through all derived models. An generic solution approach is presented that can be used in all cases where one has to deal with a mixture of generated source code (or copied source code) and manually modified source code.

4.1 File and directory restructuring

The example described here requires a TTL runtime environment installation. Unfortunately, at the time of writing this is only available to the Philips community. As an alternative one can use the source code of '˜/Triple-M/release/pc/Generate/yapi/' as the initial process network. These sources only require a YAPI runtime environment. The text below is not completely useless since the procedure remains the same, and only minor changes in the Makefiles are required. However, it is probably best to start with example 4.2 first. That example describes howto handle YAPI input sources.

Below the Producer-Consumer example is explained in a top-down way. The source code of the initial process network can be found in '˜/Triple-M/release/pc/Generate/ttl/'. We start at the main program in which the process network is created and started. Next, we describe the process network, followed by the producer process and the consumer process.

In this example we focus on a TTL to TTL transformation without specifying any functional or structural transformation. Even in such a case, file and directory restructuring is enforced to the output files. Moreover, macros are expanded and at the moment (unfortunately) comments are removed from the source code.

First, the input process network source code is introduced. After this the makefile hierarchy of the toolflow is explained. We will see that only one of these makefiles is used to configure and steer the desired transformations. Finally, after transformation the output source code are presented.

The Producer-Consumer process network is represented by the class *PC*. The declaration of this class can be found in the header file 'pc.h':

```
#include "process.h"
#include "network.h"
#include "cb_in_port.h"
#include "cb_out_port.h"
#include "channel.h"

#include "producer.h"
#include "consumer.h"

class PC : public Network
{
public:
        PC(const Id& n, int length);

        const char* type() const;

private:
        Channel<int> a;

        Producer p;
        Consumer c;
};
```

The implementation can be found in the source file ‘pc.cc’:

```
#include "pc.h"

PC::PC(const Id& n, int length) :
        Network(n),
        a(id("a"), 256),
        p(id("p"), a, length),
        c(id("c"), a)
{ }

const char* PC::type() const
{
        return "PC";
}

int main(int argc, char *argv[])
{
        PC pc(id("pc"), 1000);
        run(pc);
        printf("%s","The end.");

        return 0;
}
```

The producer process is represented by the class *Producer*. This class is declared in the header file ‘producer.h’:

```
#ifndef PRODUCER_H
#define PRODUCER_H

#include "process.h"
#include "cb_out_port.h"

using namespace ttl;

class Producer : public Process
{
public:
    Producer(const Id& n, CbOut<int>& o, int length);

    const char* type() const;
    void main();

private:
    Port< CbOut<int> > p;
    int n;
};

#endif
```

The implementation can be found in the source file ‘producer.cc’:

```
#include "producer.h"

Producer::Producer(const Id& n, CbOut<int>& o, int length) :
        Process(n),
        p(id("p"), o),
        n(length)
{ }

const char* Producer::type() const
{
    return "Producer";
}

void Producer::main()
{
        write(p, n);

        for (int i=0; i<n; i++)
        {
                write(p, i);
        }
}
```

The consumer process is represented by the class *Consumer*. This class is declared in the header file ‘consumer.h’:

```
#ifndef CONSUMER_H
#define CONSUMER_H

#include "process.h"
#include "cb_in_port.h"

using namespace ttl;

class Consumer : public Process
{
public:
  Consumer(const Id& n, CbIn<int>& i);

  const char* type() const;
  void main();

private:
        Port< CbIn<int> > p;
};

#endif
```

The implementation can be found in the source file ‘consumer.cc’:

```
01 #include "consumer.h"
02 #include "assert.h"
03 #include <iostream>
04
05 using namespace std;
06
07 Consumer::Consumer(const Id& n, CbIn<int>& i) :
08         Process(n),
09         p(id("p"), i)
10 { }
11
12 const char* Consumer::type() const
13 {
14     return "Consumer";
15 }
16
17 void Consumer::main()
18 {
19         int n;
20         read(p, n);
21
22         for (int i=0; i<n; i++)
23         {
24                 int j;
25                 read(p, j);
26                 assert(i==j);
27                 printf("Value i=%d, j=%d\n", i, j);
28         }
29 }
```

In line 27 of 'consumer.cc' we use the C programming style rather than the C++ programming style to print the values of the integer variables. This choice has been a deliberate choice, since the *SCATE* tool does not support transformations from IO stream to *printf()*. Such kind of transformations would be needed when transforming from the C++ language to the C language. Taking small things like this into account when writing an application models makes it more reusable.

Makefile-ScateSetup As explained above the common way of working (section 3.3) is to feed commands to the *SCATE* tool via a file called "Makefile-ScateSetup". An initial version of this file can be generated using the command:

```
scate -setup
```

Let us for the moment take a closer look at the toplevel file. Edit the mandatory part of the generated "Makefile-ScateSetup" until it resembles:

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

To recapitulate lines 01 up to 18 constitute the mandatory part. Here you configure the *SCATE* (line 03) and MPC (line 06) versions you want to use. Here we assume MPC has been defined as an environment variable line 06 therefore this line is commented out.

The variable CPP_INCLUDES in line 14 specifies the include path(s) for the preprocessor. We assume the environment variable TTLROOT has been defined.

The variable APISPECIFICATION in line 17 specifies files that define the API's needed during program transformation. When iterations over TTL C++ are performed, only the TTL API definition is required (can be found in the file 'ttlc++.dat' provided in the (Triple-M release). At this point the reader is encouraged to look into the APISPECIFICATION file and look for the definitions of channels, interfaces, and functions. The syntax used in this file is quite straightforward and intuative to understand.

In principle one need not fill in anything in the optional part since in the example at hand no transformations are required. However, due to an unsolved bug one needs to edit line 15 of the optional part to fix the default output API, whenever this API is different from YAPI. So in our case iterating over TTL we have to fill in::

```
15 DEFAULT_OUTPUT_API = TTL
```

We now assume for the moment the user is satisfied with all the options provided to the "Makefile-ScateSetup". Hereafter, one can generate the file "Makefile-Scate" by issuing the command (step 2 figure 4):

```
make -f Makefile-ScateSetup
```

As a result "Makefile-Scate" is generated. Execution of this "Makefile-Scate" will actually transform the input process network according to the transformations specified (none in our case) in the "Makefile-ScateSetup" (step 3 figure 4).

```
make -f Makefile-Scate
```

As a result of the previous command a set of intermediate files have been generated. These files can be composed into a compilable process network by running the generated "Makefile-Mpc" (step 4 figure 4).

```
make -f Makefile-Mpc
```

The process network output source code is generated in directory:

```
../../Design
```

The output sources are included in appendix A.1.

From comparison between the input sources (shown above) and the output sources (in appendix A.1) one can tell that the file and directory structure of the process network has been modified. In the transformed process network there are two source files ('*Main.cc' and '*.cc'). Moreover, the original (input) process network files were all collected in one directory. While the transformed process network uses a seperate directory for each process and each network.

This file and directory restructuring occurs irrespective of other transformations. The main advantage of this is that one programming style is enforced for all process networks in an automated manner.

Next we will compile the transformed process network. But, let us first check if the correct functional simulation environment is used:

```
cd ../../Design/ttl/TTL/
```

The correct value of the variables $SRCROOT$ and $SRCLIB$ can be found in the "Makefile-ScateSetup".

Edit 'Config.h' when necessary [5]:

[5]When ED&T version 1.3 of YAPI is used, one needs to set the include path (line 11) to 'YAPI_INC = $(YAPIROOT)/include/yapi'.

```
XCOMM NOTE : This is an automatically generated overwrite protected
file,
XCOMM ====   It is OK TO EDIT this file
XCOMM
XCOMM Config.h, generated by Mpc for user Triple-M
XCOMM
XCOMM Creation Date : Wed May 11 11:27:32 2005
XCOMM

#include "../../Config.h"

TTL_INC = $(TTLROOT)/include
TTL_LIB = $(TTLROOT)/lib

TTL_INC_ROOT = $(TTL_SYSTEM_ROOT)/TTL
TTL_LIB_ROOT = $(TTL_INC_ROOT)
```

Due to another unsolved bug one needs to comment out manually the line “using namespace ttl;” in the generated output files “Consumer.h” line 14 and “Producer.h” line 14.

After this one can compile the application:

```
../../Compile
```

If all went fine the result can be executed:

```
cd TestBench/toplevelname/;
./executablename [inputfiles]
```

in our case:

```
cd TestBench/PC/;
./PC
```

4.2 Api transformation

The example shown here only requires a TTL runtime environment installation to run the final output code. Unfortunately, at the time of writing this is only available to the Philips community.

In this section we demonstrate the transformation of one API into another API. Similar to section 4.1 we use a Producer-Consumer process network example. Only this time we use the YAPI syntax as a starting point. The process network after transformation will be in TTL C++ syntax.

A YAPI to TTL transformation is just one demonstration of this kind of transformation. Other examples of why one could want to modify the syntax of an API are:

- Keeping up with changes in the syntax of an API (convert legacy code).
- Intentionally make changes in the syntax of an API (for efficiency reasons or extensions to the API).
- Conversion from C++ to C[6].

An example triggered by efficiency reasons is shown in section 4.2.1, where the definition of a read function is changed.

The source code of the initial process network can be found in '~/Triple-M/release/pc/Generate/yapi/'. The Producer-Consumer process network is represented by the class *PC*. The declaration of this class can be found in the header file 'pc.h':

```
#include "yapi.h"

#include "producer.h"
#include "consumer.h"

class PC : public ProcessNetwork
{
public:
    PC(Id n, int length);

    const char* type() const;

private:
        Fifo<int> fifo;

        Producer p;
        Consumer c;
};
```

The implementation can be found in the source file 'pc.cc':

[6]Transformations from the C language to C++ are and will not be supported

```
#include "pc.h"

PC::PC(Id n, int length) :
        ProcessNetwork(n),
        fifo( id("fifo")),
        p( id("p"), fifo, length),
        c( id("c"), fifo)
{ }

const char* PC::type() const
{
    return "PC";
}

int main(int argc, char *argv[])
{
    RTE rte;

    PC pc(id("pc"), 1000);

    rte.start(pc);
    printCommunicationWorkload(pc);
    printComputationWorkload(pc);
    printDotty(pc);

    return 0;
}
```

The producer process is represented by the class *Producer*. This class is declared in the header file ‘producer.h’:

```
#ifndef PRODUCER_H
#define PRODUCER_H

#include "yapi.h"

class Producer : public Process
{
 public:
    Producer(Id n, Out<int>& o, int length);

    const char* type() const;
    void main();

 private:
    OutPort<int> p;
    int n;
};

#endif
```

The implementation can be found in the source file ‘producer.cc’:

```
#include "producer.h"

Producer::Producer(Id n, Out<int>& o, int length) :
    Process(n),
    p( id("p"), o),
    n(length)
{ }

const char* Producer::type() const
{
    return "Producer";
}

void Producer::main()
{
        write(p, n);

        for (int i=0; i<n; i++)
        {
                write(p, i);
        }
}
```

The consumer process is represented by the class *Consumer*. This class is declared in the header file ‘consumer.h’:

```
#ifndef CONSUMER_H
#define CONSUMER_H

#include "yapi.h"

class Consumer : public Process
{
 public:
    Consumer(Id n, In<int>& i);

    const char* type() const;
    void main();

 private:
    InPort<int> p;
};

#endif
```

The implementation can be found in the source file ‘consumer.cc’:

```
#include "consumer.h"
#include <iostream>

using namespace std;

Consumer::Consumer(Id n, In<int>& i) :
    Process(n),
    p( id("in"), i)
{ }

const char* Consumer::type() const
{
    return "Consumer";
}

void Consumer::main()
{
        int n;
        read(p, n);

        for (int i=0; i<n; i++)
        {
                int j;
                read(p, j);
                assert(i==j);
                printf("Value i=%d, j=%d\n", i, j);
        }
}
```

Makefile-ScateSetup As explained above (section 3.3) an initial version of the file "Makefile-ScateSetup" can be generated using the command:

```
scate -setup
```

Given this process network in YAPI syntax we will now translate it into TTL syntax. The first step is to create an initial version of "Makefile-ScateSetup" as depicted in step 1 figure 4 and edit its content, until its resembles:

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

We have commented out line 06, assuming MPC has been defined as a environment variable. If this is not the case fill in the desired executable name.

The variable CPP_INCLUDES in line 14 specifies the include paths for the preprocessor. In this example CPP_INCLUDES should be set to *'$$(YAPIROOT)/include/'* including the quotes[7].

The variable APISPECIFICATION in line 17 specifies those files that define the API's needed during program transformation. When translating from YAPI to TTL C++, both API definitions are required ('yapi.dat' and 'ttlc++.dat'). You can specify the absolute path or the relative path. We recommend a third alternative using the environment variable TRIPLEMROOT as shown.

This is enough info for the moment. Continue with step 2 and 3 of figure 4. As indicated in the figure '*.map' files will be created. For translation to TTL C++ you need to copy the toplevel map file *PC.map* into *PC.ttl*. The *SCATE* tool regenerates the '*.map' files whenever you run the tool. So you need to make a copy to avoid losing your changes during an iteration.

The content of the file *PC.ttl* should be changed to resembles this:

[7] Unless the ED&T release 1.3 of YAPI is used. In this case you should set CPP_INCLUDES to *'$$(YAPIROOT)/include/yapi/'*.

```
01 MAPPING PC
02 {
03    TRANSPORT
04    {
05       mem_fifo["256"]
06       {
07          TTL::Channel<int>
08       }
09    }
10    NET
11    {
12       fifo -> mem_fifo
13    }
14    MAPPING Consumer : c
15    {
16    }
17    MAPPING Producer : p
18    {
19    }
20 }
```

This MAPPING specification binds transformations to YAPI / TTL design entities. As shown these specifications can be nested. The TRANSPORT section defines synchronized token containers. In practice such a container is defined by selecting a YAPI FIFO or a TTL CB channel [8].

The value 256 (line 05) between brackets specifies the size of the channel in number of tokens. So in this case, the channel can contain 256 integers.

The prefix ‘TTL::’ (line 07) 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 NET section describes that FIFO instance fifo will be mapped onto TRANSPORT instance ‘mem_fifo’.

Subsequently we have to instruct *SCATE* via the “Makefile-ScateSetup” to use the mapping file:

```
MAPPINGFILES = PC.ttl
```

Moreover we can reuse the symbol table [9] that has been built in the second run by copying the file *symtab* into *symtab.this* (the *symtab* file is overwritten each run). To speed up parsing we can now add the following information to the “Makefile-ScateSetup” file:

[8]See reference [6] for more details.

[9]The compiler uses a symbol table to keep track of the user-defined symbols in the program.

```
EXTRA_FLAGS = -symtab symtab.this
```

To mimic the YAPI equivalent of the *printDotty(pc)* toplevel main function one can add the flag '-dottyView' to the "Makefile-ScateSetup":

```
EXTRA_FLAGS = -symtab symtab.this \
              -dottyView
```

This will produce a graphical view on the process network after transformation in 'dotty' format (file 'dottyview.dot'). This graphical view is created after static analysis of the source code, i.e., it does not require running the process network executable. Moreover it is independent of the API that is used.

The dotty format is a textual format that can be used to view graphs. The program *dotty* [10] 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 > dottyview.ps
```

We now return to step 2 and 3 of figure 4 and recreate a new "Makefile-Scate" file that will transform the process network as instructed. If all went well the output text will clearly state that the YAPI processes have been recognized and translated to TTL processes:

```
========================== Transform Api Recognition ============
- PROCESS Consumer has been transformed from YAPI to TTL
- NETWORK PC has been transformed from YAPI to TTL
- PROCESS Producer has been transformed from YAPI to TTL
=================================================================
```

At this point in the flow *SCATE* has produced all the required input files for MPC, see step 3 figure 2. Next, we combine them into a TTL C++ process network using MPC. This is done in step 4 of figure 4.

The output files are not shown here explicitly, but they should closely resemble the files shown in appendix A.1. However, since we performed an API to API translation, the calls with which the process network is created and started could be different for the two APIs involved. These kind of transformations are not supported by the *SCATE* tool. Edit the toplevel main function manually (file 'main.cc') by inserting lines 17–19 and commenting out line 20–24 as shown:

[10] Note, *dotty* is available for users at the Nat.Lab. through the cadappl tree (*cadenv graphviz*).

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// PCMain.cc, generated by Mpc for user Triple-M
//
// Creation Date : Fri May 13 14:33:29 2005
//

#include "PC.h"

const char* PC::type() const
{
   return "PC";
}
int main(int argc, char* argv[])
{
   PC pc(id("pc"), 1000);

   run(pc);
//    RTE rte;
//    rte.start(pc);
//    printCommunicationWorkload(pc);
//    printComputationWorkload(pc);
//    printDotty(pc);
   return 0;
}
```

In order to prevent that these manual changes will be lost during another run of the tooling add the flag '-ignoreMain' to make the toplevel main function overwrite protected:

```
EXTRA_FLAGS = -symtab symtab.this \
              -dottyView \
              -ignoreMain
```

There is another more generic and powerful - but also more complex - manner to prevent the loss of manual changes, see section 4.4.

The TTL process network is now ready to be compiled and executed:

```
cd ../../Design/ttl/TTL/;
```

Edit 'Config.h' when necessary.

```
../../Compile
```

4.2.1 Api definition

The example shown here requires a TTL runtime environment installation. Unfortunately, at the time of writing this is only available to the Philips community. Nevertheless, the example shown indicates the advantage of having a configurable API, which is also the case for a YAPI API.

In this section we take a closer look at the API definition language. This language instructs *SCATE* about the channels and interfaces that are available, and which services are provided by the interfaces in the form of functions.

As an illustrating example we will change the syntax of the read function provided by TTL. Copy the existing 'ttlc++.dat' into a new file called 'new_ttlc++.dat'. Edit this file by changing the API name into 'NEW_TTL', and adopt the definition of the read function from:

```
FUNCTION read(PORT, VALUE)
{
   IN:ENTRY
   IN:DATA
   IN:LEAVE
}
```

into:

```
FUNCTION read(PORT) RETURNS &VALUE
{
  IN:ENTRY
  IN:DATA
  IN:LEAVE
}
```

Copy the mapping file *PC.ttl* created in the previous section into *PC.new_ttl*, and modify:

```
TTL::Channel<int>
```

into:

```
NEW_TTL::Channel<int>
```

Makefile-ScateSetup Edit “Makefile-ScateSetup” and change *PC.ttl* into *PC.new_ttl*.

Now rerun step 2–4 of figure 4. After this all read functions in the output ‘NEW_TTL’ source code are transformed. Unfortunately we do not have a runtime environment to execute the final code, nevertheless this example is shown here as an illustration.

4.3 Network transformations

The example shown here requires a TTL runtime environment installation. Unfortunately, at the time of writing this is only available to the Philips community. As an alternative one can use the source code of ‘˜/Triple-M/release/jpeg/Generate/yapi/’ as the initial process network.

In this section network transformations are introduced and explained. For this purpose the Producer-Consumer example is slightly modified into a process network with (artificial) hierarchy. The consumer process is embedded into a consumer network *cn*, see figure 6. The initial souce code can be found in directory ‘˜/Triple-M/release/pc/Generate/flatten/’.

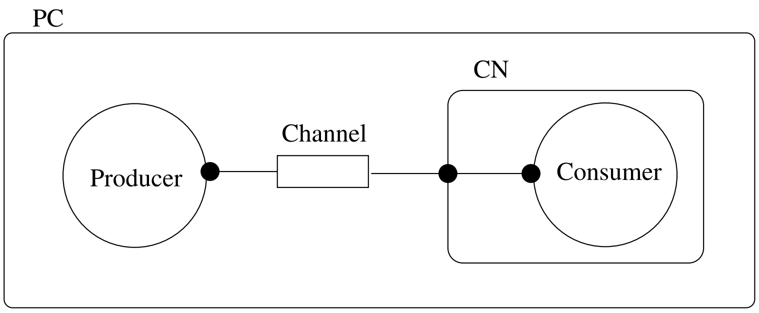

Figure 6: Producer-Consumer with hierarchy.

The consumer network is represented by the class *CN*. The declaration of this class can be found in the header file ‘cn.h’:

```
#include "process.h"
#include "network.h"
#include "cb_in_port.h"
#include "cb_out_port.h"
#include "channel.h"

#include "consumer.h"

class CN : public Network
{
public:
    // constructor
    CN(const Id& n,
       CbIn<int>& CNin);

        const char* type() const;

private:
    // input ports
    NetworkPort< CbIn<int> > CNinP;

    // channels

    //processes and networks
        Consumer c;
};
```

The implementation can be found in the source file ‘cn.cc’:

```
#include "cn.h"

CN::CN(const Id& n,
       CbIn<int>& CNin) :
        Network(n),
    // ports
        CNinP(id("CNinP"), CNin),
    // processes and networks
    c
    (
        id("c"),
        CNinP
    )
{ }

const char* CN::type() const
{
        return "CN";
}
```

The introduction of the network described above lead to changes in the toplevel network file ‘pc.h’:

```
#include "process.h"
#include "network.h"
#include "cb_in_port.h"
#include "cb_out_port.h"
#include "channel.h"

#include "producer.h"
#include "cn.h"

class PC : public Network
{
public:
        PC(const Id& n, int length);

        const char* type() const;

private:
        Channel<int> a;

        Producer p;
        CN cn;
};
```

Likewise also some changes in the toplevel implementation network file ‘pc.cc’:

```
#include "pc.h"

PC::PC(const Id& n, int length) :
        Network(n),
        a(id("a"), 256),
        p(id("p"), a, length),
        cn(id("cn"), a)
{ }

const char* PC::type() const
{
        return "PC";
}

int main(int argc, char *argv[])
{
        PC pc(id("pc"), 1000);
        run(pc);
        printf("%s","The end.");

        return 0;
}
```

Generate a fresh "Makefile-ScateSetup" as described several times above, step 1 figure 4. Besides the usual configuration of this file - MPC, CPP_INCLUDES, and APISPECIFICATION (see section 3.3) - one may appreciate the current example more by generating a dotty file before the network transformation (see section 4.2) [11].

A quick view on the *dotty* output reveals that the network class *CN* has been added to the process network structure as also shown in figure 6.

4.3.1 Mapping file

Flattening of a process network can be described in a mapping file:

```
MAPPING *
{
      FLATTEN
}
```

The precise syntax of the MAPPING specification is described in detail in report [6]. In words, "what is specified here is that the structural transformation statement FLATTEN will be applied to each unit". Flattening of a process network is a generic transformation for which the mapping file is provided by the Triple-M release (file 'flatten.dat').

Add the mapping file that flattens the process network to "Makefile-ScateSetup":

```
MAPPINGFILES = '$$(TRIPLEMROOT)/share/mapping/flatten.dat'
```

Rerun the *SCATE* tooling as usual, and generate output sources using *MPC*. The output sources after transformation can be found in appendix A.2. Look again at the graphical view (file 'dottyview.dot') after transformation, and confirm that the network *CN* has indeed been removed from the output sources.

4.4 Consistency between various models

Keeping iterations over a number of related source code models consistent is a real challenge. This problem would be less difficult when all design decisions could be automated from start to finish. Currently this is not possible - and most likely never will be - since many design decisions are based on some creative process that is hard / impossible to automate.

The second best thing one can do is to help the user as much as possible in keeping the various models consistent. For example consider what happens when after creating a number of derived process network models a change is made to the initial model. These

[11] Note, in order to create 'dottyview.dot' there is no need to run MPC.

changes should then ripple through all the derived models. The requirements that are defined to help the user are the following:

- Automatically patch when possible.
- Never loose information.
- Provide a stepwise approach.

Some changes in the source code are too disruptive to be handled by the solution approach chosen. For those cases there is no other option than to continue manually. While still obeying the requirement not to loose any information. Currently there is little experience where exactly the borderline is of what type of changes can and what type of changes cannot be handled in practice.

4.4.1 Source code patching

The example described here requires a TTL runtime environment installation. Unfortunately, at the time of writing this is only available to the Philips community. As an alternative we stated in example 4.1 to use the source code of '~/Triple-M/release/pc/Generate/yapi/' as the initial process network. These sources only require a YAPI runtime environment. The text below is not completely useless since the procedure remains the same, and only minor changes in the Makefiles are required.

The basic idea of a patch mechanism is to add or remove source code based on a context difference between a copy of the original file and the latest version of this file. Each hunk of the patch can be applied without any problem to a file that is a copy of the original file. In some cases one can even apply these patches without any problems to a modified version of a copy of the original file. This holds as long as the context of where patches occur can still be recognized. If the tool 'patch' cannot find a place to install a hunk of the patch, it puts the hunk out to a reject file, which normally is the name of the output file plus a .rej suffix. The line numbers on the hunks in the reject file may be different than in the patch file: they reflect the approximate location patch thinks the failed hunks belong in the new file rather than the old one.

These ideas have been implemented and extended in the following manner that allows for automatic generation of patches as well. Suppose a (sub)set of files in a directory (called 'patchDir') have the following properties. They start from an initial version that is from then on solely modified via the patch mechanism explained above. In order to put the idea of automatic generation of patches into practice one needs a separate directory (called '.cvsPatchDir') in which copies of the initial versions are stored under version management. New versions of these files are simply copied or generated into this '.cvsPatchDir', thereby overwriting the old versions. Version management can now help to create a patch by running a context diff w.r.t. the versions stored in the repository. This patch can now be used to automatically remove or add code in the 'patchDir' directory.

After this has been done the '.cvsPatchDir' directory should be checked in into the repository, since they now form a new frame of reference for the next iteration. At this point one may wonder what the use of the patch mechanism is since one could simply overwrite the files in the 'patchDir'. The neat thing is that with the extra machinery described - that can be automated with the help of scripts - it also allows for manual editing of the exact same files located in the 'patchDir'.

The approach described in the previous paragraph to source code patching results in a manageable system in which a (sub)set of files may contain a mixture between (automatically) generated source code and handwritten source code [12]. All requirements formulated are fulfilled to manage consistency between various related source level models and iterations over them, i.e. output models of one transformation can be used as input for another.

For a user working with the *SCATE* tool, source code patching as described can be useful in two ways:

1 Patch **input** application code.

2 Patch generated *SCATE* **output** code to:

- manually correct bugs in the transformed source code.
- edit source code for transformations that have not been implemented yet.
- edit source code fragments that are not be supported (e.g. stop and start functions in the toplevel main function, see section 4.2).

Source code patching as described in item 1 enables one to perform iterations over various related versions of source level models without loosing their dependencies. A change in any of the intermediate models can now ripple through all models that depend on it, assuming one sequentially transforms each of those models. With source code patching as described in item 2 in place there is no or little need to edit the transformed process network as generated by *MPC* [13]. One could even consider not putting the MPC generated output sources under version management.

In the following we demonstrate howto setup source code patching as described in item 1 and 2 for process network transformations. Even though both items are discussed in one example they can be used independently.

We reuse the source code from the 'ttl' example (section 4.1) for the initial setup. Shadow directories are created, sources are put under version management, and scripts are explained to enable patching. As an example we will simulate a late change in the tokensize - from int to short - of the channel in the initial process network. Thereafter additional steps are introduced that allow automated testing.

We assume in the following a cvs repository is available [14]. For those who do not have a repository at hand can create one on their private disk:

[12] Note, it is also allowed to manually change generated source code.

[13] Assuming there are no bugs in *MPC*.

[14] For details we refer to the cvs manual [9].

```
cvs -d ~/cvsroot init
```

This repository will be used throughout the example presented below.

Manual setup Create a new directory - within the 'Generate' directory - called 'patch' and enter this directory. Create a project in the repository of cvs and update to this project:

```
export CVSROOT=~/cvsroot
cvs import -m 'Created patch project.' patch patch start
cd ..
rmdir patch
cvs co patch
cd patch
```

Run the following command in the 'patch' directory [15]:

```
cvsCreateApplPatchDir ../ttl/
```

The argument of the script should be an absolute or relative path to the process network that is transformed. The script creates, in the current directory, a hidden directory '.cvsApplPatchDir', and copies the input application sources in there [16].

One should now manually copy those sources in the current directory:

```
cp .cvsApplPatchDir/* .
```

This suffices for the patch mechanism of the input application code. We now continue with the patch mechanism of the *scate* output sources.

Create a fresh toplevel 'Makefile', see step 1 of figure 4:

```
scate -setup
```

Create the file "Makefile-ScateSetup":

```
make -f Makefile-ScateSetup
```

Edit the generated "Makefile-ScateSetup" and define MPC, CPP_INCLUDES as described in section 4.1. Create intermediate files:

```
make -f Makefile-Scate
```

Run the script:

[15] Note, with slight modification one could also apply the procedure described here to an already existing transformation. This is left for the reader as an exercise.

[16] '*.cc', '*.c' and '*.h', with the exception of '*Options.h' and 'Config.h'

```
cvsCreateScateOutputPatchDir
```

This creates a '.cvsScateOutputPatchDir' and copies the intermediate sources in there.

Edit the generated "Makefile-ScateSetup" and add the flag:

```
EXTRA_FLAG = -cvsOutputPatch
```

This treats intermediate files produced by *SCATE* in the current directory as overwrite protected [17], while still generating (and overwriting) code in the shadow directory '.cvsScateOutputPatchDir'. One may now add other transformations as desired.

Recreate intermediate files in the hidden directory, see step 2–3 of figure 4:

```
make -f Makefile-ScateSetup
make -f Makefile-Scate
```

Look at the output to confirm that files are created in the shadow directories and not directly in the current directory.

Add the directories '.cvsScateOutputPatchDir' and '.cvsApplPatchDir' plus all files to your project in the repository:

```
cvs add *
cvs add .cvsScateOutputPatchDir
cd .cvsScateOutputPatchDir
cvs add *
cd ..
cvs add .cvsApplPatchDir
cd .cvsApplPatchDir
cvs add *
cd ..
cvs ci -m 'Add initial files and directories to project patch.'
```

When multiple developers are working on the same sources one should advert to the use of branches [9]. This prevents content interference in the hidden directories '.cvsScateOutputPatchDir' and '.cvsCiApplPatch'. Each developer is then able to generate code into these hidden directories, and to maintain it independently using version management. This is how to create a branch with name 'tagname' - suggestion: use for this a numbered three letter acronym of your name -:

```
cvs tag -b tagname `find . -type d -name '.cvsScateOutputPatchDir'`
```

When these commands are issued all '.cvsScateOutputPatchDir' directories are put on a private branch. Similarly:

```
cvs tag -b tagname `find . -type d -name '.cvsApplPatchDir'`
```

[17] 'installfile.yapi.*', '*.net', and '*.MPC'.

Then update to this branch:

```
cvs update -r tagname
```

Continue as usual by adding additional transformations when appropriate. A thing or two have changed now, i.e., one should patch the input application code whenever it has changed **before** a new iteration is started. Application code can be patched using:

```
cvsApplPatch
```

Likewise, one should patch intermediate output code of *scate* right **after** it has been generated in every iteration cycle using:

```
cvsScateOutputPatch
```

Note, with this mechanism is place one is allowed - at any time - to edit the intermediate sources as required. After patches have been applied on needs to check the new files into the repository:

```
cvsCiScateOutputPatch
cvsCiApplPatch
```

We are now set to apply source code patching on the example at hand. Suppose the type of the channel in the example at hand has been changed by the application developer from *int* to *short* for efficiency reasons. Such changes to the input code can now be handled without problem. For this exercise the transformation to *short* has been prepared and can be imported as follows:

```
cvsCreateApplPatchDir ../short/
```

This overwrites the files in the '.cvsApplPatchDir'.

Before starting a second iteration over the sources one needs to patch the input code for *scate* in the current directory as stated above:

```
cvsApplPatch
```

Create new intermediate output as usual:

```
make -f Makefile-ScateSetup
make -f Makefile-Scate
```

Finally, patch the output produced by *scate*:

```
cvsScateOutputPatch
```

Before checking in it is instructive to look at the differences w.r.t. repository:

```
cvsDiffScateOutputPatch
cvsDiffApplPatch
```

Before the next iteration is started one should check in the current set of files as a new frame of reference:

```
cvsCiScateOutputPatch
cvsCiApplPatch
```

As an assignment one could now edit the top level main function as described in section 4.2, without using the '-ignoreMain' flag.

Automate testing The procedure described in the previous section can be extended and automated such that a complete test flow is obtained including the source code patching mechanism as described above. The result of such a test is summarized in a HTML file. One scenario could be to periodically run such automated test for error detection, and to send the resulting file to all members of the team involved. Another complementary scenario could be to use the patching mechanism to keep all related models upto date.

Edit the "Makefile-ScateSetup" in the previous 'patch' example and add two flags:

```
EXTRA_FLAG = -cvsOutputPatch \
             -mkImportMakefile \
             -mkTestMakefile
```

The '-cvsOutputPatch' has already been described, see section 4.4.1. The '-mkImportMakefile' and the '-mkTestMakefile' flags instruct *SCATE* to generate a "Makefile-Import" and a "Makefile-Test", respectively.

Furthermore, fill in the test related part of the "Makefile-ScateSetup" file as much as possible. At least fill in the directory where the "Makefile" of the original application can be found, and the name of the executable that is created by this "Makefile":

```
##################################### Test Related Part #

# Original source root directory of application files
# (Makefile is assumed to be present as well)
ORIG_SRC_ROOT = '../short'

# Original source its executable name
ORIG_EXE_NAME = pc

# Original source its executable arguments
# ORIG_EXE_ARGS =

# Name of output file
# NAME_OUTPUT_FILE =
```

Run:

```
make -f Makefile-ScateSetup
```

Besides a “Makefile-Scate” file this now also generates a “Makefile-Import” file, see step 2 of figure 4.

Edit “Makefile-Import” such that the directory(ies) of the (original) application sources is / are known. In our case this information can be extracted from the ‘Test Related Part‘ of the toplevel “Makefile-ScateSetup”. Things could be more tricky when the original sources are spread over multiple directories, when the case adjust the “Makefile-Import” accordingly. One could now run the command:

```
make -f Makefile-Import
```

This will execute script ‘cvsCreateApplPatchDir’ which - if it does not exist - creates ‘.cvsApplPatchDir’, and imports (copies) the input application sources. It also executes the script called ‘cvsApplPatch’. This patches the input code when it has been changed w.r.t. repository. When CVS related errors occur at this stage one should check whether CVSROOT has been set in the current shell. Those errors should then disappear in a next iteration. In a moment we will see that it is not needed to call ‘make -f Makefile-Import’ explicitly from the command line as shown. Rather this call will be done by the script *scate_test*.

Run *SCATE* with the all its arguments:

```
make -f Makefile-Scate
```

This creates intermediate files, and a “Makefile-Test” file, see step 3 of figure 4. When the test related part of the“Makefile-ScateSetup” is filled in appropriately one generally need not edit this file anymore. Otherwise make the necessary corrections in the “Makefile-Test” or alternative delete this file [18] and correct “Makefile-ScateSetup” and continue from there as usual.

[18]“Makefile-Test” is treated as overwrite protected by *SCATE*.

We are all set now. In order to further automate 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 this strategy the following tests are automatically performed:

- Build native application
- Run native application
- Import and patch input application
- Apply transformation
- SCATE output patch
- Create application with MPC
- Build transformed application
- Run transformed application
- File output diff

All these tests produce a status like 'passed' or 'failed'. The outcome of the *scate _test* script can be visualized in the form of an HTML page, see figure 7.

Run the following commands:

```
cd ~/Triple-M/release/pc/Generate/
scate_test patch > test.html
```

Providing one or more test cases - 'patch' in our case - is optional. The script will otherwise traverse the directory tree in search of a 'Makefile-Test'. In the example at hand one will get an error during compilation of the generated sources due to a known bug, see also page 19. The line "using namespace ttl;" in the generated output files "Consumer.h" line 14 and "Producer.h" line 14 should be commented out. As described above on page 34 source code patching can also be used to manually correct bugs in the transformed source code. We can fix this bug in *SCATE* by commenting out the content in the intermediate files 'Consumer.preClass.h.MPC' and 'Producer.preClass.h.MPC'. These '*.MPC' are produced by *SCATE*, and will be used as input by MPC. Changes to any of these intermediate files will therefore end up in the final process network. Rerunning *scate _test* should now be succesfull. The 'File output diff' still test ends with not equal, because the test is designed to perform a diff on an output file generated by the application. In the present example no output file is generated, instead a comparison is done on all files in the directory. This test can be suppressed by commenting out the target 'has-run-cmp' in the 'Makefile-Test'.

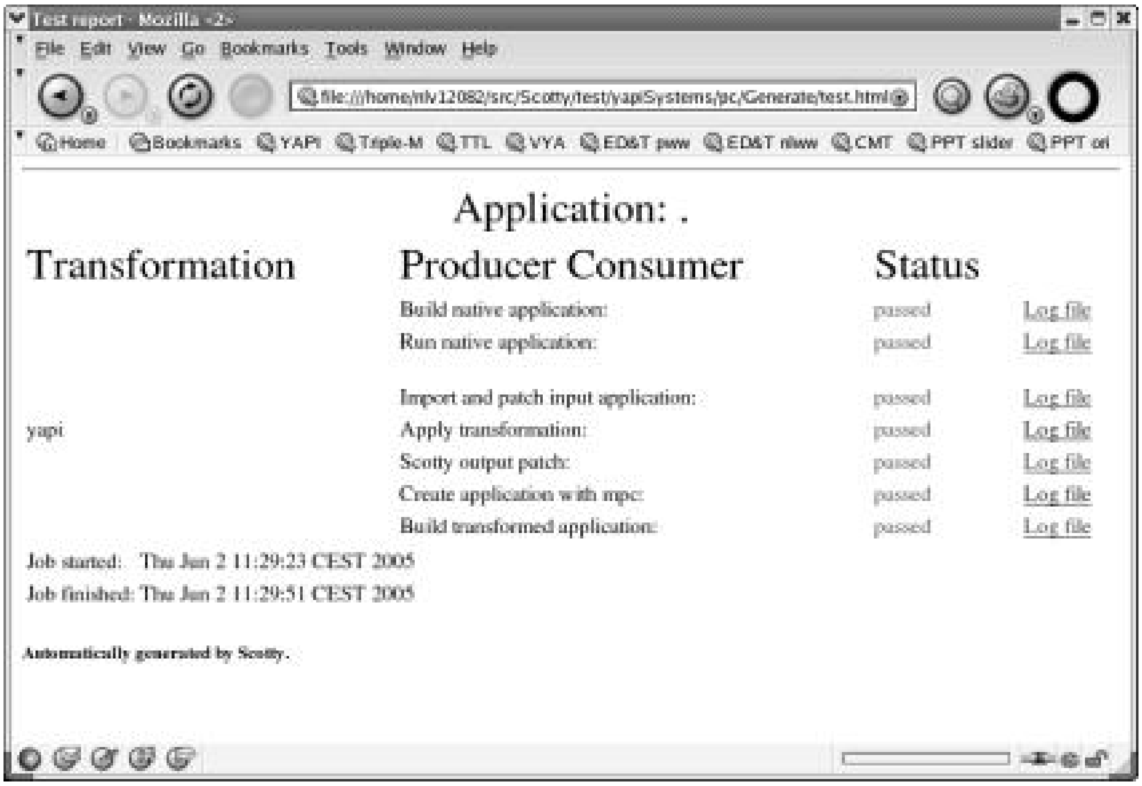

Figure 7: Test report.

5 Problems, Troubleshooting

In case of trouble one could try to remove all the intermediate files produced by *SCATE* using the command [19]:

```
make -f Makefile-Scate clean
```

or

```
make -f Makefile-Scate very_clean
```

This latter command also removes the preprocessed application files, 'Make.SrcVars', and the load order file. Note, after this step it requires an extra (automatic) *SCATE* run to re-determine the load order.

Whenever possible use version control. This allows one to use:

```
cvs diff
```

to figure out what has changed since the previous iteration..

[19]This command should only be used when *SCATE* output patching is NOT used. Otherwise one could unintentionally remove manual changes as well.

5.1 FAQ

unexpected token When the CPP_INCLUDES path is not set correctly one will get an error: 'No such file or directory'. Still a *.cpp file is produced containing wrong information. One should delete this file before continuing:

```
make -f Makefile-Scate very_clean
```

Continuing without removing the *.cpp file, while fixing the CPP _INCLUDES path will lead to an error: 'unexpected token:'

symtab The '-symtab *symtabfilename*' should only be used as an EXTRA_FLAGS option in "Makefile-ScateSetup". Otherwise an error message will occur:
'unexpected token: API'

cvs For questions about cvs we refer to the manual [9].

5.2 Bugs

Bugs can be reported at sourceforge [4].

5.3 Wishlist

All remarks are welcome.

A Examples: source code

A.1 File and directory restructuring

This section contains the output source code of the example described in section 4.1. The declaration of class *PC* can be found in the header file ‘PC.h’:

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// PC.h, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#ifndef PC_INCLUDED
#define PC_INCLUDED

#include "channel.h"
#include "network.h"
#include "ttl_use_namespace.h"

#include "ttlTypes.h"
#include "cb_channel.h"
#include "rb_channel.h"
#include "rn_channel.h"
#include "dbi_channel.h"
#include "dbo_channel.h"
#include "dno_channel.h"
#include "dni_channel.h"
#include "cb_in_port.h"
#include "cb_out_port.h"

#include "Consumer.h"
#include "Producer.h"

class PC: public Network
{
   public:
      // constructor
      PC
         (Id n,
          int length);

      // extra public member function declarations
      const char* type() const;

   private:

      // input ports
      int length_param;

      // channels
      Channel<int> a;

      // processes and networks
      Consumer i0_Consumer;
      Producer i0_Producer;
};

#endif // PC_INCLUDED
```

The declaration of this class can be found in the header file ‘PC.cc’:

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// PC.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#include "PC.h"

PC::PC
   (Id n,
    int length)
:
   Network(n),
   // ports
   length_param(length),
   // channels
   a(id("a"), 256),
   // processes and networks
   i0_Consumer
   (
      id("i0_Consumer"),
      a
   ),
   i0_Producer
   (
      id("i0_Producer"),
      length_param,
      a
   )
{}
```

The toplevel main function can be found in file ‘PCMain.cc’:

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// PCMain.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#include "PC.h"

const char* PC::type() const
{
   return "PC";
}
int main(int argc, char* argv[])
{
   PC pc(id("pc"), 1000);
   run(pc);
   printf("%s", "The end.");
   return 0;
}
```

The producer process is represented by the class *Producer*. This class is declared in the header file ‘Producer.h’:

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// Producer.h, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#ifndef Producer_INCLUDED
#define Producer_INCLUDED


// pre class items
//using namespace ttl;

#include "cb_in_port.h"
#include "cb_out_port.h"
#include "process.h"
#include "ttl_use_namespace.h"

#include "ttlTypes.h"
#include "cb_out_port.h"


class Producer: public Process
{
   public:
      // constructor
      Producer
         (Id n,
          int length,
          CbOut<int>& o);

      // extra public member function declarations
      const char* type() const;
      void main();

   private:

      // input ports
      int n;

      // output ports
      Port< CbOut<int> > p;
};

#endif // Producer_INCLUDED
```

The constructor can be found in the source file ‘Producer.cc’:

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// Producer.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#include "Producer.h"

Producer::Producer
   (Id n,
    int length,
    CbOut<int>& o)
:
   Process(n),
   // ports
   n(length),
   p(id("p"), o)
{}
```

The implementation of the member functions can be found in the source file 'Producer-Main.cc':

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// ProducerMain.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#include "Producer.h"

const char* Producer::type() const
{
   return "Producer";
}
void Producer::main()
{
   write(p, n);
   for (int i = 0; i < n; i++)
   {
      write(p, i);
   }
}
```

The consumer process is represented by the class *Consumer*. This class is declared in the header file 'Consumer.h':

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// Consumer.h, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#ifndef Consumer_INCLUDED
#define Consumer_INCLUDED


// pre class items
//using namespace ttl;

#include "cb_in_port.h"
#include "cb_out_port.h"
#include "process.h"
#include "ttl_use_namespace.h"

#include "ttlTypes.h"
#include "cb_in_port.h"


class Consumer: public Process
{
   public:
      // constructor
      Consumer
         (Id n,
          CbIn<int>& i);

      // extra public member function declarations
      const char* type() const;
      void main();

   private:

      // input ports
      Port< CbIn<int> > p;
};

#endif // Consumer_INCLUDED
```

The implementation of the constructor can be found in the source file ‘Consumer.cc’:

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// Consumer.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//


// include items
#include <assert.h>
#include <iostream>

#include "Consumer.h"

Consumer::Consumer
    (Id n,
     CbIn<int>& i)
:
   Process(n),
   // ports
   p(id("p"), i)
{}
```

The implementation of the member functions can be found in the source file ‘Consumer-Main.cc’:

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// ConsumerMain.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed May 11 16:48:56 2005
//

#include <assert.h>
#include <iostream>
#include "Consumer.h"

using namespace std;
const char* Consumer::type() const
{
   return "Consumer";
}
void Consumer::main()
{
   int n;
   read(p, n);
   for (int i = 0; i < n; i++)
   {
      int j;
      read(p, j);
      assert(i == j);
      printf("Value i=%d, j=%d\n", i, j);
   }
}
```

A.2 Network transformations

This section contains the output source code of the example described in section 4.3.

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// PC.h, generated by Mpc for user Triple-M
//
// Creation Date : Wed Jun 15 10:49:01 2005
//

#ifndef PC_INCLUDED
#define PC_INCLUDED

#include "channel.h"
#include "network.h"
#include "ttl_use_namespace.h"

#include "flattenTypes.h"
#include "cb_channel.h"
#include "rb_channel.h"
#include "rn_channel.h"
#include "dbi_channel.h"
#include "dbo_channel.h"
#include "dno_channel.h"
#include "dni_channel.h"
#include "cb_out_port.h"
#include "cb_in_port.h"

#include "Producer.h"
#include "Consumer.h"

class PC: public Network
{
   public:
      // constructor
      PC
         (Id n,
          int length);

      // extra public member function declarations
      const char* type() const;

   private:

      // input ports
      int length_param;

      // channels
      Channel<int> a;

      // processes and networks
      Producer i0_Producer;
      Consumer i0_Consumer;
};

#endif // PC_INCLUDED
```

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// PC.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed Jun 15 10:49:01 2005
//

#include "PC.h"

PC::PC
   (Id n,
    int length)
:
   Network(n),
   // ports
   length_param(length),
   // channels
   a(id("a"), 256),
   // processes and networks
   i0_Producer
   (
      id("i0_Producer"),
      length_param,
      a
   ),
   i0_Consumer
   (
      id("i0_Consumer"),
      a
   )
{}
```

```
// WARNING : This file has been automatically generated,
// =======   DO NOT EDIT unless you know what you are doing...
//
// PCMain.cc, generated by Mpc for user Triple-M
//
// Creation Date : Wed Jun 15 10:49:01 2005
//

#include "PC.h"

const char* PC::type() const
{
   return "PC";
}
int main(int argc, char* argv[])
{
   PC pc(id("pc"), 1000);
   run(pc);
   printf("%s", "The end.");
   return 0;
}
```

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] 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.

[4] http://sourceforge.net/projects/scate/ .

[5] http://sourceforge.net/projects/mpsc/ .

[6] D. Alders and O. Popp. The Triple-M software infrastructure for YAPI and TTL source code analysis and transformations (http://sourceforge.net/projects/scate/) . Technical report, July 2005.

[7] *Software Portability with imake*. Number ISBN: 1-56592-226-3. O'Reilly & Associates, Inc., 1996.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[9] *https://www.cvshome.org/docs/manual/* .

Author(s)	D. Alders
Title	Scate Manual