



FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Property-based Synthesis of Optimized Büchi Automata (Deliverable 3.2/10)

Due date of deliverable: September 1, 2006
Actual submission date: September 3, 2006

Start date of project: January 1, 2004

Duration: Three years

Organisation name of lead contractor for this deliverable: TU-Graz

Revision 1.0

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	<input checked="" type="checkbox"/>
PP	Restricted to other programme participants (including the Commission Services)	<input type="checkbox"/>
RE	Restricted to a group specified by the consortium (including the Commission Services)	<input type="checkbox"/>
CO	Confidential, only for members of the consortium (including the Commission Services)	<input type="checkbox"/>

Notices

For information, contact Roderick Bloem rbloem@ist.tugraz.at.

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 3.2/10 described in contract number 507219.

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

© Copyright PROSYD 2006. All rights reserved.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	May, 2006	Creation	Pill	All
0.2	June, 2006	Draft	Pill	All
0.3	June, 2006	Glossary, Introduction, Summary, Automata Synthesis related sections	Pill	1, 2.1, 3.1, 4.1, 5
0.4	June, 2006	Revision	Pill, Bloem	All
0.5	June, 2006	Translator related sections	Sheinvald	2.2, 3.2, 4.2, 4.3
0.6	July, 2006	Revision, Version sent to partners for review	Bloem, Pill, Sheinvald	All
0.7	August, 2006	Incorporated comments, further enhancements	Pill	All
1.0	August, 2006	Final approval by project coordinator	Eisner	All

Authors

Ingo Pill
Sarai Sheinvald
Roderick Bloem

Part of the work was done in joint research with Alessandro Cimatti, Marco Roveri, and Simone Semprini from our partner ITC-irst.

Executive Summary

In this deliverable we present an implementation of an optimized PSL automata construction. With our tool a designer can derive alternating and nondeterministic explicit automata from PSL formulae. Furthermore our tool offers the option of deriving a symbolic representation in the SMV format (used e.g. by the NuSMV [10] model checker) using a new approach we presented at the *11th International Conference on Implementation and Application of Automata, 2006* in [8]. We performed extensive experiments with our optimizations and symbolic representations, shown in [8].

We give installation instructions in Section 2. In Section 3 we give available syntax and options. Then, in Section 4 we give the technical details of the syntax translation and automata synthesis optimizations implemented. Finally we draw conclusions in Section 5.

Purpose

This document describes the implementation of our tool for synthesizing optimized Büchi automata from PSL properties.

Intended Audience

This document is aimed at people who want to build Büchi automata for PSL properties. This includes people using formal verification as well as engineers from the Electronic Design Automation (EDA) industry or scientific community working on tools for formal verification.

Background

This work is based on the theory presented in Deliverable 3.2/4 [6] for synthesis of automata from PSL. This document focuses on the implementation of an optimized synthesis. An unoptimized translation was presented in Deliverable 3.3/1 [18].

Contents

Table of Revisions	iii
Authors	iii
Executive Summary	iii
Purpose	iii
Intended Audience.....	iv
Background	iv
Contents	v
Table of Figures	vi
Glossary	vii
1 Introduction	1
2 Installation.....	3
2.1 Automata Synthesis.....	3
Software Requirements	3
Licensing and Warranty	4
How to obtain the Automata Synthesis Tool	4
Integration of our Synthesis Library into Private VIS Sources.....	5
2.2 Syntactic Sugar Translation	5
Software Requirements	6
Licensing and Warranty	6
How to obtain the PSL Translation Component.....	6
Integration of our Translation Library into Private Sources.....	7
3 Usage.....	9
3.1 Automata Synthesis.....	9
sugar_to_aut Command and Options	9
Example.....	10
3.2 Syntactic Sugar Translation	14
PSLRW Program and Options	15
Example of the file to file usage	15
Example of the char* to char* usage.....	16
4 Technical Details.....	17
4.1 Automata Synthesis.....	17
Automata Construction Optimizations	17
4.2 Syntactic Sugar Translation	18
4.3 Testing.....	18
5 Summary.....	21
6 References.....	23

Table of Figures

Glossary

ABA, Alternating Büchi Automaton

Alternating Büchi automata feature both existential and universal choice for the transition relation. See *nondeterminism* and *universality* for information on existential and universal choice.

Kleene Closure

The Kleene closure for regular expressions is a unary operator on single syntactical elements or sets of syntactical elements of a language. S^* describes the set of words built by concatenating zero or more elements of S .

LTL, Linear Temporal Logic, Linear-Time Temporal Logic

Linear temporal logic is a temporal logic for property specification in formal verification [20].

NBA, Nondeterministic Büchi Automaton

Non-deterministic Büchi automata feature existential choice only for the transition relation.

Non-determinism

Non-determinism or existential choice for automata describes the possibility of choosing one of several possible successors, defining a different run for every possible successor.

Property

A collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behaviors.

PSL

Property specification language. This is the specification language used throughout and building the basis for the PROSYD project. Refer to the Language Reference Manual [4] for more details on the language.

Regular Expression

A regular expression is a pattern or expression that describes a set of finite words.

SERE

Sugar Extended Regular Expressions, as featured by the PSL language.

UFA, Universal Büchi Automaton

Universal Büchi automata feature universal choice for the transition relation only.

Universalism, Universality

Universalism or universal choice for automata describes the property that all possible successors have to be chosen simultaneously, splitting a run into several branches.

1 Introduction

In this deliverable we provide a tool for the optimized synthesis of Büchi automata from PSL [5] properties. With our tool a designer can derive alternating and non-deterministic explicit automata from PSL formulae. Furthermore we offer the option of generating a symbolic representation in the SMV format (used e.g. by the NuSMV [10] model checker) using a new approach we presented at the *11th International Conference on Implementation and Application of Automata, 2006* in [8]. We performed extensive experiments with our optimizations and symbolic representation, shown in [8].

PSL is a temporal logic standardized by the Accellera standards organization. The core of PSL is an extension of the linear temporal logic LTL [20] adding new basic formulae like SEREs and operators.

For the linear temporal logic LTL the standard approach for formal verification methods like model checking is to generate nondeterministic Büchi automata for the properties [23, 22, 11, 16]. LTL formulae can be translated to one-weak automata, a restrictive subclass of alternating automata. A one-weak alternating automaton with n states can then be translated to a nondeterministic automaton with $n \cdot 2^n$ states [16].

PSL includes the Kleene Closure and thus is not star free like LTL. Therefore PSL cannot be translated to one-weak automata, but to (weak) alternating automata as shown in Deliverables 3.2/4 [6] and 3.3/1 [18]. In general, the conversion of a (weak) alternating automaton to a nondeterministic version requires the use of Miyano and Hayashi's construction [17] resulting in a blowup to $O(3^n)$ states.

An efficient implementation of the construction and conversion algorithms is a key to the successful application of PSL in formal verification. In this deliverable we present a tool capable of providing optimized alternating and nondeterministic Büchi automata for PSL formulae.

The tool consists of a translator and a synthesizer. The translator takes an arbitrary PSL property and converts it into *core PSL*. This property is then synthesized into an automaton by the synthesizer.

The document is organized as follows. In the next section we will discuss installation related topics. Then in Section 3 we discuss available commands and their usage. Technical details are given in Section 4, including code structure and underlying theory. In Section 5 we draw conclusions, and show implicit benefits to other Deliverables.

In Table 1 we report the status of the features of this tool and deliverable as from the Description of Work document. The list contains *mandatory* and *nice to have* features, with the intention that the minimal requirement for this deliverable is the implementation of all mandatory features. We implemented all mandatory and one of two nice to have features. There were no features that were rated *desirable*.

	Present	Ref.
Mandatory Features		
Pointers to algorithms used	YES	4
List of target operating systems	YES	2.1, 2.2
Explanation of coding standards	YES	4
Discussion of license issues	YES	2.1, 2.2
User documentation, including documentation of user interface (command line switches) and imported/exported file formats	YES	2, 3
Test suite	YES	4.3
Standard Input Language - PSL	YES	3.1, 3.2
Support for Verilog Flavor	YES	3.1, 3.2
Output explicit representation of automaton	YES	3.1
Nice to have features		
Support for other flavors	NO	
Output symbolic representation of automaton	YES	3.1, 4.1

Table 1: Table of features

We use "Yes" and "No" respectively to state whether the feature has been implemented or not. In the third column we show references to related sections in this document.

2 Installation

This section gives information on installation related issues such as software requirements, licensing, and installation procedure for both the automaton synthesizer and the syntactic sugar translator. These two components come with different licenses and can be independently integrated to other programs. Therefore this section contains two subsections focusing on the specific details of the two components.

We provide our tool via two packages available from the PROSYD website [1].

- *Binary Package:* This package contains pre-compiled binaries. See Sections 2.1 and 2.2 for details on the compilation platform.
- *Library Package:* In this section we provide our sources such that designers may integrate our code into private sources according to the licenses shown in Sections 2.1 and 2.2. The *src/synthesis* and *src/translator* contain the sources of our two components.

All packages are provided as compressed archives. To decompress the archives perform the following steps:

- Create a new directory to contain provided data, and copy the archive to this directory.
- Issue the following command to decompress the contents of the archive into the current directory:

```
tar zxvf BuechiTool.tgz or tar zxvf BuechiTool_library.tgz
```

2.1 Automata Synthesis

Software Requirements

The provided automata synthesis functionality has been developed as a library for VIS [21] version 2.1. VIS-2.1 has been built and tested on the following platforms, and may work on others:

- IBM RISC System/6000 / AIX Version 4.3.3 / gcc, g++

- Intel ix86 / Linux / gcc, g++
- Intel ix86 / Windows98SE with Cygwin 1.3.2 / gcc, g++
- Sun Sparc/ Solaris 2.8 / gcc, cc, g++

The provided executable of the binary package was compiled with gcc 3.3.5-20050130 on a Gentoo Linux based 32bit x86 machine running a 2.6.14 series kernel. It should run on any similar machine.

Licensing and Warranty

The following copyright notice (with varying copyright holders and dates) holds for the VIS source code, the source code of the provided automata synthesis library, and the pre-compiled automata synthesis tool.

Copyright (c) 2004-2006 Graz University of Technology. All rights reserved.

Permission is hereby granted, without written agreement and without license or royalty fees, to use, copy, modify, and distribute this software and its documentation for any purpose, provided that the above copyright notice and the following two paragraphs appear in all copies of this software.

In no event shall Graz University of Technology be liable to any party for direct, indirect, special, incidental, or consequential damages arising out of the use of this software and its documentation, even if the Graz University of Technology has been advised of the possibility of such damage.

Graz University of Technology specifically disclaims any warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. the software provided hereunder is on an “as is” basis, and the Graz University of Technology has no obligation to provide maintenance, support, updates, enhancements, or modifications.

Note that the license allows for commercial use, and that such use currently takes place.

How to obtain the Automata Synthesis Tool

We provide our automata synthesis functionality via two packages available from the PROSYD website [1]. There you can find the package named Buechi Tool in the software category on the links page. These packages also include the translator component, whose specific details are covered in Section 2.2.

- *Binary Package:* This package contains a pre-compiled binary. See Section 2.1 for details on the compilation platform.
- *Library Package:* We provide the sources of our library for the integration into a private VIS-2.1 source tree. With this library a user can compile binaries for platforms other than that of the pre-compiled version, or modify the code according to the license shown in Section 2.1.

The executable package contains an executable of the synthesizer in the *bin* subdirectory. For other platforms, and to support engineers we provide a library package containing the source code for integration into local VIS-2.1 source trees.

Integration of our Synthesis Library into Private VIS Sources

This section describes how to integrate our synthesis library contained in the library package into private VIS sources, which can be downloaded from [24]. It is assumed that the reader is familiar with compiling VIS and has a working VIS source tree, whereby related information can be found in the documentation provided with VIS. In the following we describe the necessary steps to perform. Additional information regarding general development and integration of new VIS packages can be found in the VIS Engineering Manual [12]. The location of the VIS source tree is assumed to be */vis*. Please adjust given directory names to suit your installation.

The following steps integrate the library into private VIS sources.

- Create a new directory in */vis/common/src* to contain the library:
/vis/common/src/sugar. The chosen subdirectory name *sugar* refers to the package name used in VIS.
- Copy the provided source code files (*src/synthesis/.**) to this directory.
- Add the newly introduced package *sugar* to the list of packages in */vis/common/src/Makefile* : *PKGS= sugar ...*
- Make a symbolic link in */vis/common/include* to point to */vis/common/src/sugar/sugar.h*.
- Add *#include "sugar.h"* in */vis/common/src/vm/vm.h*.
- Add calls to the library's initialization and termination functions *Sugar_Init()* and *Sugar_End()* in */vis/common/src/vm/vmVinit.c* for correct initialization and termination.
- Edit the Makefile of your architecture, e.g. */vis/i686-g/Makefile* for x86 machines, to add the sugar package to the list of packages; *PKGS= sugar*

An invocation of “make” within the directory containing the makefile for your architecture, e.g. */vis/i686-g/Makefile* for newer x86 machines, creates a new binary capable of synthesizing Büchi automata from PSL formulae.

2.2 Syntactic Sugar Translation

Software Requirements

The provided PSL syntactic sugar translation functionality has been built and tested on the following platforms, and may work on others:

- Intel ix86 / Linux / gcc, g++

The provided executable of the binary package was compiled with gcc 3.2.3-2 on a Red Hat Linux based 32bit x86 machine. It should run on any similar machine.

Furthermore, the following software requirements should be met:

- Bison, version 1.875 or later
- Flax, version 2.5.4 or later
- Perl, for the build process

Licensing and Warranty

The following copyright notice (with varying copyright holders and dates) holds for the provided PSL translation library source code, and the pre-compiled PSL rewriter binary.

The information in this document is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Note that the license allows for commercial use.

How to obtain the PSL Translation Component

The PSL translation component is contained in the two versions (library/binary) of the automata synthesis tool package. The package can be downloaded from the PROSYD website [1]. There you can find the package named Buechi Tool in the software category on the links page.

- *Binary Package*: This package contains a pre-compiled binary called *pslrw* for the translation. See Section 2.2 for details on the compilation platform.

- *Library Package:* We provide the sources of our translation library for the integration into private sources. With this library a user can compile binaries for platforms other than that of the pre-compiled version, or modify the code according to the license shown in Section 2.2.

The executable package contains an executable for the translation in the *bin* subdirectory. For other platforms, and to support engineers, we provide a library package containing the source code. With these sources a user may compile system specific executables, or integrate them into private projects.

Integration of our Translation Library into Private Sources

This section describes how to integrate our PSL translator library contained in the library package into private sources.

We provide two “main” files, with the following features.

- A main file called *main.cpp* which demonstrates the basic input-file to output-file version of the translator.
- A main file called *main_string.cpp_* which demonstrates a basic char* to char* version of the translator.

Note that only one of these files can be used in compilation. Therefore, according to your needs, name the main file you wish to use *main.cpp* and run the compilation process.

The following steps integrate the library into your private sources.

- Choose a directory for your sources and copy the provided source code of the translation component (*src/translator/*.**) to this directory.
- As mentioned, the default main file is *main.cpp*. If you wish to make use of the other provided main file, *main_string.cpp_*, rename it *main.cpp* before compiling (you may want to rename *main.cpp* beforehand, to avoid running over it).
- Edit the *makefile.settings* file to the proper settings. Set the TOP variable to the path of the *pslrw* directory.
- Issue the following command: *make*

Once the build process is done, you should see the module file *pslrw* in your directory. This file is a driver for PSLRW, and is based on the source *main.cpp* in the same directory.

3 Usage

This section offers detailed information on the usage of the provided functionality. For both the automata synthesis library and the syntactic sugar translator we show available commands, and depict their options as well as input and output formats.

3.1 Automata Synthesis

In this section we outline the usage of the automata synthesis library to synthesize Büchi automata from PSL formulae, which shall be given in Verilog flavor. With our library a designer can create alternating, nondeterministic, and deterministic automata from PSL formulae.

sugar_to_aut Command and Options

Using the *sugar_to_aut* command the user can create explicit and symbolic automata from PSL formulae. Information on the resulting automata can be directed to the standard output or stored in given files. We implemented several automatic and transparent optimizations.

Please note that the *-m* switch only affects the translation from the PSL package automata format to that used by the LTL package. The PSL formulae shall be given in an input file using PSL Verilog flavor.

The output files generated with the *-o* switch use the *.dot* file format as described in [2]. The symbolic representations obtained with the *-s* switch generate *SMV* files as supported by the NuSMV [10] model checker. Documentation on this file format can be found at the NuSMV homepage [3].

```
sugar_to_aut [-h] [-v level] [-r] [-d] [-l] [-m value]
             [-o out] [-s sym] <input_file>
```

-h	Prints the help message containing parameter information to the standard output.
-v level	This option determines the verbosity level given by a number from 0 to 3. The default value is 0. Information provided by a lower number is also provided by higher numbers. 0: Print out the formula. 1: Print out automata statistics. 2: Give a transition list of all edges. 3: Give vertex list.
-r	Generate a non-deterministic automaton instead of an alternating one.
-d	Generate a deterministic automaton.
-l	Generate a Buechi automaton usable by the LTL package ignoring options -d or -r.
-m value	Generate a minimized Buechi automaton usable by the LTL package. The option value can be any sum of the following values to derive any minimization combination: 1: Prune fair sets 2: Direct-simulation 4: Reverse-simulation 8: I/O compatible simulation
-o out	Write the automata produced for the input formulas to separate output files. The file name for each automaton is derived by value “out” followed by the formula number and the extension “.dot”.
-s sym	Derive symbolic representations for the produced automata and write them to separate output files. The file name for each automaton is the value of “sym” followed by the formula number and the extension “.smv”.
input_file	The given file should contain the input formulae for which automata shall be derived. The formulae should be written in the Verilog flavor of PSL.

Example

This section shows a simple example scenario for deriving automata from PSL formulae. We show an example input file containing a PSL formula to be synthesized, and show the necessary commands to derive corresponding alternating and nondeterministic Büchi automata, and create an SMV file.

- **Write an input file containing the formula:** Use your favorite editor to write an input file containing one or more properties in Verilog PSL flavor. A small example could be:

```
{a;bi{c}[*]} |-> {d;e;f};
```

- **Start VIS:** Execute the VIS binary. You will receive a VIS command prompt with a greeting message indicating the VIS version used.

```
vis/i686-g> ./vis
vis release 2.1 (compiled 28-Jun-06 at 10:35 AM)
vis>
```

- **Derive automata:** With the command *sugar_to_aut* you can derive automata from PSL formulae. In this example we will derive an alternating and a nondeterministic explicit automaton for the given example formula. The switch *v* “verbose level” controls the amount of information printed on the screen. For this example we choose level three.

First we derive the alternating Büchi automaton:

```
vis> sugar_to_aut -v3 input_file
```

Automaton:

```
-----
Name: {a;b;{c}[*]} |-> {d;e;f}
-----
```

States:

```
q0:  IN 0 / OUT 2 (138005896)
q1:  IN 1 / OUT 2 (138021904)
q2:  IN 2 / OUT 2 (138006088)
q5:  IN 2 / OUT 1 (138035312)
q6:  IN 1 / OUT 1 (138035368)
q7:  IN 5 / OUT 1 (138035456)
```

Arcs:

```
-> q0(138005896)
q0 -> q1  Label: 'a'
q7 -> q7  Label: TRUE
q5 -> q6  Label: 'e'
q6 -> q7  Label: 'f'
q1 -> q2,q5 Label: 'b'&'d'
q2 -> q2,q5 Label: 'c'&'d'
q0 -> q7  Label: '!a'
q1 -> q7  Label: '!b'
q2 -> q7  Label: '!c'
```

Accept States:

```
{q0(138005896),q1(138021904),q2(138006088),
q5(138035312),q6(138035368),q7(138035456)}
```

```
Number of states:      6
Number of symbols:    3
Number of arcs:       9
Number of initial states: {1}
Number of accept states: 6
```

Now we derive a nondeterministic version by adding the switch *r*.

```
vis> sugar_to_aut -v3 -r input_file
```

Automaton:

```
-----  
Name: {a;b;{c}[*]} |-> {d;e;f} <EXPANDED>  
-----
```

States:

```
q0:  IN 0 / OUT 2 (138070944)  
q1:  IN 1 / OUT 2 (138071200)  
q2:  IN 4 / OUT 1 (138071376)  
q3:  IN 1 / OUT 2 (138071696)  
q4:  IN 2 / OUT 2 (138072048)  
q5:  IN 2 / OUT 1 (138072272)
```

Arcs:

```
-> q0(138070944)  
q0 -> q1 Label: 'a'  
q0 -> q2 Label: '!a'  
q2 -> q2 Label: TRUE  
q1 -> q3 Label: 'b'&'d'  
q1 -> q2 Label: '!b'  
q3 -> q4 Label: 'c'&'d'&'e'  
q3 -> q5 Label: '!c'&'e'  
q5 -> q2 Label: 'f'  
q4 -> q4 Label: 'c'&'d'&'e'&'f'  
q4 -> q5 Label: '!c'&'e'&'f'
```

Accept States:

```
{q0(138070944),q1(138071200),q2(138071376),q3(138071696),  
q4(138072048),q5(138072272)}
```

```
Number of states:      6  
Number of symbols:    3  
Number of arcs:       10  
Number of initial states: {1}  
Number of accept states: 6
```

Please note that our example shows no significant blowup in the state space for the nondeterministic version, but was chosen for lucidity.

- **Create an SMV file:** Using the switch *s* you can generate a symbolic representation in SMV format and write the result to a given file. This time we omit the *v* switch and thus we are not presented with the details of the automaton.

```
vis> sugar_to_aut -s output_file inku input_file
```

```
Automaton:
```

```
-----  
Name: {a;b;{c}[*]} |-> {d;e;f} <EXPANDED>  
-----
```

The SMV file generated for the example looks like the following, whereby we added some line-breaks for the output to fit into this document.

```
-- Symbolic representation generated by VIS for PSL formula  
-- {a;b;{c}[*]} |-> {d;e;f} <EXPANDED>
```

```
MODULE main
```

```
--variables for symbols:
```

```
VAR d : boolean;
```

```
VAR e : boolean;
```

```
VAR f : boolean;
```

```
--variables for L and R:
```

```
VAR qL_1_0 : boolean;
```

```
VAR qL_1_1 : boolean;
```

```
VAR qL_1_3 : boolean;
```

```
VAR qL_1_4 : boolean;
```

```
VAR qL_1_5 : boolean;
```

```
VAR qL_1_2 : boolean;
```

```
DEFINE __Rempty__1 := TRUE;
```

```
--T_LC for Q_WN
```

```
--T_LC for Q_WA
```

```
TRANS(qL_1_0 -> ((a & next(qL_1_1)) | (!a & next(qL_1_2))))
```

```
TRANS(qL_1_1 -> ((b & d & next(qL_1_3)) |  
    (!b & next(qL_1_2))))
```

```
TRANS(qL_1_3 -> ((c & d & e & next(qL_1_4)) |  
    (!c & e & next(qL_1_5))))
```

```
TRANS(qL_1_4 -> ((c & d & e & f & next(qL_1_4)) |  
    (!c & e & f & next(qL_1_5))))
```

```
TRANS(qL_1_5 -> ((f & next(qL_1_2))))
```

```
TRANS(qL_1_2 -> ((TRUE & next(qL_1_2))))
```

```

--T_LC for Q_SA

--T_LC for Q_SN

--T_I

--T_F

--T_N

--Encode preimage function as we are strict

--Q_WN

--Q_WA
TRANS(next(qL_1_1) -> (qL_1_0))
TRANS(next(qL_1_3) -> (qL_1_1))
TRANS(next(qL_1_4) -> (qL_1_3 | qL_1_4))
TRANS(next(qL_1_5) -> (qL_1_3 | qL_1_4))
TRANS(next(qL_1_2) -> (qL_1_0 | qL_1_2 | qL_1_1 | qL_1_5))

--Q_SA

--Q_SN

--Initial values:

INIT ((qL_1_0 & (!qL_1_1) & (!qL_1_2) & (!qL_1_3) &
      (!qL_1_4) & (!qL_1_5)));

--Fairness constraints:

FAIRNESS __Rempty__1;

```

As will be described in Section 4.1 the construction of the SMV files follows the approach we presented in [8]. Please note the comments, which refer to our approach and explain from which parts of the construction the different parts of the representation originate.

3.2 Syntactic Sugar Translation

In this section we outline the usage of the sugar translation component to translate PSL formulae into core PSL. The translation component offers the option to compile an executable as well as provides an interface for the integration into private sources. In this section we depict both the command line interface of the executable, and the parameters of the interface for the integration into private sources.

PSLRW Program and Options

Running the *pslrw* program the user can create core PSL formulae translated from an input file containing PSL formulae in Verilog flavor. The output is a file containing the translated formulae.

All translations are according to the rewriting rules mentioned in [5].

The formulae to be translated are divided into several groups. The user may choose not to translate one of the groups by switching on its flag.

```
./pslrw [-bfr] [-nxt] [-wtn] [-vnt] [-fa] [-sere]
        [-none] <input_file>
```

-bfr	Switches off the translation of the 'before' operators: before, before!, before_, before!_ .
-nxt	Switches off the translation of the 'next_a/e' operators: next_a, next_a!, next_e, next_e! .
-wtn	Switches off the translation of the 'within' operators: within, within!, within_, within!_ .
-vnt	Switches off the translation of the 'next_event' operators: next_event, next_event!, next_event (ternary), next_event! (ternary), next_event_a, next_event_a!, next_event_e, next_event_e! .
-fa	Switches off the translation of the 'forall' operator.
-sere	Switches off the translation of the operators mentioned in section 3.1 of [5]
-none	Switches off all translations.
input_file	The given file should contain the input formulae for translation. The formulae should be written in the Verilog flavor of PSL.

Example of the file to file usage

This section shows a simple example scenario for translating PSL formulae. We show an example input file containing a PSL formula to be translated, and show the necessary commands to create a translated output file.

- **Write an input file containing the formula:** Use your favorite editor to write an input file containing one or more properties in Verilog PSL flavor. A small example could be:

```
// comment
(a before b) before!_ (a until b);
```

- **Run PSLRW:** Execute the `./pslrw` program. It produces an output file containing the translated results.

```
./pslrw myfile.txt
```

PSLRW will produce a file called `out_myfile.txt`, containing the translated formula:

```
// comment
[(![a W b]) U [(!b) W (a && (!b))]];
```

Example of the char* to char* usage

This section shows a simple example scenario for using the provided `TranslateString` function, which receives a char buffer containing a single PSL formula, flags to optionally suppress translation of special formulae, and returns a translated formula.

- Add the following to your source file

```
#include "rwstring.hpp"
```

An example of the usage of `TranslateString` is provided in the file `TranslateString.cpp`. See 2.2 for further details. An example of usage is:

```
bool bfrFlag=true, nxtFlag=true, wtnFlag = true,
    vntFlag=true, faFlag=true, sereFlag=true;
char* formula = "always (a before c)";
const char* result = TranslateString(formula, bfrFlag, nxtFlag,
    wtnFlag, vntFlag, faFlag, sereFlag);
```

Once executed, `result` will hold the translated formula. The flags of the provided interface offer the option to deactivate the translation of several groups of PSL formulae. The flags used for the function interface are the same as those available on the command line for the executable depicted before.

4 Technical Details

This section covers the technical details of our tools. We outline source code structure, show results of our test phases, and depict implemented algorithms and underlying theory.

4.1 Automata Synthesis

The provided library integrates into the VIS model checker. Our code has been developed in compliance to the VIS Engineering Manual [12] framing special filenames and a specific code structure. As the library evolved from that of Deliverable 3.3/1 [18] and uses the code structure declared there, we refer the reader in this respect to that deliverable. A research report on the theoretical background can be found in Deliverable 3.2/4 [6]. The underlying theory of our approach to derive a symbolic representation can be found in [8]. In the following we depict implemented optimizations not covered by earlier deliverables, whereas information on tests performed can be found in Section 4.3.

Automata Construction Optimizations

In this section we cover the optimizations implemented for our library. We implemented optimizations for both the alternating Büchi automaton (ABW) and the translation into an nondeterministic Büchi automaton (NBW).

Implemented optimizations on the alternating automaton include heuristics and the optimizations presented in [15].

The resulting algorithm basically consists of six stages and is performed iteratively throughout the construction:

- Optimize edge labels and remove edges labeled *false*.
- Remove universal edges containing a loopback if the state is non-accepting, as outgoing paths cannot be accepting.
- Remove states which cannot reach an accepting state.
- Remove states unreachable from the initial states.
- Simplify the edge set of a state by removing redundant edges and combine edges where possible.

- Use direct simulation relation to minimize state space and edges.

For the explicit construction of the nondeterministic automaton as of Miyano and Hayashi's approach [17], we implemented on-the-fly optimizations presented in [14]. We compute the direct simulation relation on the alternating automaton and use it to prune both the state space and edges of the non-deterministic automaton on-the-fly. Thus we avoid the computation of parts of the automaton which could be pruned afterwards by optimizations in the first place.

Last but not least we developed an approach to derive a symbolic representation directly from the alternating automaton [8]. Thus we can derive a symbolic representation avoiding the explicit construction of the nondeterministic automaton. This explicit construction has shown to be a bottleneck, as is shown by our experiments in [8].

Furthermore we combined the approach of Miyano and Hayashi [17] for general alternating automata with that of Gastin and Oddoux for one-weak automata [16]. As mentioned in the Introduction there is a gap between the complexity of these algorithms. For an alternating automaton with n states Miyano and Hayashi's construction results in an automaton with size $O(3^n)$. For the subset of one-weak alternating automata (of size n) we can produce nondeterministic versions of size $O(n \cdot 2^n)$.

In our approach we combine the generality of Miyano and Hayashi's approach with the efficiency of Gastin and Oddoux' technique: We use the general approach for those parts of the automaton where it is necessary and use the more efficient one whenever possible. Thus besides avoiding the bottleneck of an explicit construction of an intermediate nondeterministic automaton we also increase performance. Our work was presented at the "11th International Conference on Implementation and Application of Automata". Please refer to our paper [8] for detailed information.

4.2 Syntactic Sugar Translation

The syntactic Sugar Translator is based on the IBM Parser PSLP - a Front-End for PSL/Sugar.

The parser builds a parse tree from an input file containing PSL formulae. The PSLRW tool then traverses the parse tree, rearranging its construction according to the desired translations. All translations are according to the rewriting rules mentioned in [5]. An additional traversal is then performed in order to output the resulting parse tree.

For further details regarding the PSLP Parser, please see [13].

4.3 Testing

Testing has been carried out repeatedly throughout the whole development and implementation process. For our tests we generated several dozens of formulae. To detect problems in the synthesizer with interference of optimization methods we performed regression tests whenever a new optimization method was added or improved. Furthermore we utilized several combinations of optimizations for the tests. Due to the iterative use of optimizations in the synthesis algorithm, this approach enlarged the number of (intermediate) automata provided to each optimization stage.

With our tests we found a few bugs resulting in the design of additional related tests. For example, we found and fixed two critical bugs related to the interference of optimization methods which caused the construction to segfault for complex properties.

The testing activity reported here relates to the library development for VIS. As we are working together with our colleagues at ITC-irst on integrating the library to NuSMV [10] as well, additional tests have been performed in this setting. As of the date of this report there are no known bugs.

For the syntactic sugar translation, extensive tests have been carried out by generating several dozens of formulae consisting of all translation groups, and running the synthesizer on the derived translations.

5 Summary

In this deliverable we present the implementation of a tool that translates PSL formulae into automata. For the automata synthesis library we implemented state-of-the-art technology and developed new technologies addressing performance bottlenecks for a further enhancement in efficiency. We presented these new algorithms to the scientific community at the *11th International Conference on Implementation and Application of Automata, 2006* in [8]. For the construction of a nondeterministic automaton, we combined the generality of Miyano and Hayashi's construction with the efficiency of that of Gastin and Oddoux, and provide a symbolic implementation. Thus we are able to derive a symbolic representation of an alternating automaton avoiding an explicit construction of an intermediate explicit nondeterministic automaton, but can perform that directly. During our tests shown in [8], we found this construction of an intermediate automaton to be one of the bottlenecks of formal verification. In combination with our efficiency enhanced translation this shows a significant speedup [8].

The RAT tool provided with Deliverable 1.2/4-5 [7], and presented to the scientific community at the *43rd Design Automation Conference, 2006* in [19] and the *17th European Conference on Artificial Intelligence, 2006* in [9], benefits from our new library with a significant speedup in Property Simulation. Although our library was developed for VIS [21], we are currently working on its integration to the NuSMV [10] model checker, such that we provide an efficient implementation for both the free VIS and NuSMV model checkers.

Our implementation of the translator from syntactic sugar to the core syntax of PSL offers benefits for a wide range of tools. Using the translator as a preprocessor, tool designers can focus on supporting the core syntax while the preprocessor adds support for syntactic sugar. As a side-effect, a designer can use shorter syntactic sugar and ensure confidence in the written properties by translating them to the core syntax and compare the representations against each other.

6 References

- [1] Prosyd public deliverables. <http://www.prosyd.org/twiki/view/Public/PublicDeliverables>.
- [2] The DOT Language. <http://graphviz.org/doc/info/lang.html>.
- [3] The NuSMV homepage. <http://nusmv.irst.itc.it>.
- [4] Accellera. Property specification language — reference manual, version 1.1, June 2004.
- [5] Accellera Organization, Inc. Formal semantics of Accellera property specification language. In *Appendix B of <http://www.eda.org/vfv/docs/PSL-v1.1.pdf>*, pages 109–119, January 2004.
- [6] S. Ben-David, R. Bloem, D. Fisman, A. Griesmayer, I. Pill, and S. Ruah. Automata construction algorithms optimized for PSL. <http://www.prosyd.org>, 2005. Prosyd D 3.2/4.
- [7] R. Bloem, R. Cavada, C. Eisner, I. Pill, M. Roveri, and S. Semprini. Manual for Property Simulation and Assurance Tool. <http://www.prosyd.org>, 2005. Prosyd D 1.2/4-5.
- [8] R. Bloem, A. Cimatti, I. Pill, M. Roveri, and S. Semprini. Symbolic implementation of alternating automata. In *Conference on Implementation and Application of Automata*, 2006. To Appear.
- [9] R. Bloem, A. Cimatti, I. Pill, M. Roveri, S. Semprini, and A. Tchaltsev. RAT: A Tool for Formal Analysis of Requirements. ECAI Systems demonstration, 2006.
- [10] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new Symbolic Model Verifier. In N. Halbwachs and D. Peled, editors, *Proceedings of the Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633, pages 495–499, Trento, Italy, July 1999. Springer-Verlag.
- [11] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, (1):47–71, 1997.
- [12] S. Edwards and G. Swamy. *The VIS Engineering Manual*. The VIS Group. http://vlsi.colorado.edu/vis/doc/VisEngineering/vis_eng.ps.
- [13] M. Farkash, L. Gluhovsky, A. Orni, and S. Rabinovich. PSLP - a Front-end for PSL/Sugar. <http://www.prosyd.org/twiki/view/Public/DeliverablePageWP1>. Prosyd D 1.2/3.
- [14] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In *Conference on Implementation and Application of Automata*, pages 35–48, 2003. LNCS 2759.
- [15] C. Fritz and T. Wilke. Simulation relations for alternating Büchi automata. *Theoretical Computer Science*, 338:275–314, 2005.
- [16] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Thirteenth Conference on Computer Aided Verification (CAV '01)*, pages 53–65. Springer-Verlag, 2001. LNCS 2102.
- [17] S. Miyano and T. Hayashi. Alternating finite automata on ω -words. *Theoretical Computer Science*, 32:321–330, 1984.

- [18] I. Pill, A. Griesmayer, and R. Bloem. Manual for VIS Tool Port. <http://www.prosyd.org>, 2005. Prosyd D 3.3/1.
- [19] I. Pill, S. Semprini, R. Cavada, M. Roveri, R. Bloem, and A. Cimatti. Formal Analysis of Hardware Requirements. In *Design Automation Conference*, 2006. To Appear.
- [20] A. Pnueli. The temporal semantics of concurrent programs. *Theoret. Comput. Science*, 13:45–60, 1981.
- [21] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. -T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.
- [22] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263, 2000.
- [23] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Logic in Computer Science*, pages 322–331, 1986.
- [24] The VIS Group. *The VIS Home Page*. <http://vlsi.colorado.edu/vis/>.