FP6-IST-507219

PROSYD:

Property-Based System Design

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

Manual for Property-Based Synthesis Tool
(Deliverable 2.2/3)

Due date of deliverable: 1 August 2006
Actual submission date: 13 August 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	☒
PP	Restricted to other programme participants (including the Commission Services)	☐
RE	Restricted to a group specified by the consortium (including the Commission Services)	☐
CO	Confidential, only for members of the consortium (including the Commission Services)	☐

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 2.2/3 described in contract number 507219.

Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	June 6, 2006	Creation	Jobstmann	All
0.2	June 7, 2006	Using Lily	Jobstmann	Section 2
0.3	June 8, 2006	Syntax Summary	Jobstmann	Appendix
0.4	June 9, 2006	Wrote Introduction	Jobstmann	Section 1
0.5	June 9, 2006	Included Theory	Bloem, Jobstmann	Section 5
0.6	June 12, 2006	Extended User Manual	Jobstmann	Section 2
0.7	June 13, 2006	Completed syntax summary and Section 2	Jobstmann	Section A.4, 2
0.9	June 19, 2006	Wrote installation guide and technical details	Jobstmann	Section 3, 4
0.91	June 20, 2006	Front matters	Jobstmann	Front matters
0.92	June 30, 2006	Revised all sections	Bloem	All
1.0	July 17, 2006	Revision based on feedback from project manager	Bloem	All

Authors

Roderick Bloem
Barbara Jobstmann

Executive Summary

We present our property-based synthesis tool Lily. Given a set of properties written in the linear-time fragment of PSL and a partition of the signals used in those properties into input and output signals, Lily synthesizes a functionally correct design for the given properties. The synthesized design, a finite state machine, is provided as a VERILOG module or as a labeled directed graph in DOT format.

This document states how to use and install Lily and gives technical and theoretical details about the tool.

Purpose

The purpose of this document is to describe the effort done to develop a property-based synthesis tool for the linear-time fragment of PSL. Furthermore, it explains how to install and use this tool.

Intended Audience

This guide is intended for researchers working with PSL or a similar specification language, who want to use automatic synthesis. It is assumed that readers are familiar with the notions and terms related to PSL and VERILOG. In order to understand the underlying theory readers need to have a good understanding of model checking, of game theory, and of automata theory, including tree automata and alternating automata on infinite words.

Background

Synthesis of linear-time formulas is closely related to Church's problem of synthesis for S1S [Chu62]. It was formalized by Pnueli and Rosner [PR89]. There exist a few implementations covering subsets of LTL but to our knowledge no implementation for the complete language. Recent work of Amir Pnueli handles the most general subset. His approach is applicable to specifications expressible with a generalized Streett[1] acceptance condition. Those specifications have to be rewritten to a particular syntax in order to be synthesized. The work presented here is the first implementation of a synthesis algorithm for the linear-time fragment of PSL.

Contents

Table of Figures

Glossary

Acceptance Condition
A condition defining how an infinite automaton accepts an input object. We use Büchi and co-Büchi acceptance conditions both defined by a set of states F. An input word is Büchi accepted by an automaton, if the set of states visited infinitely often while reading the input word intersects the set F. Dually, a word is co-Büchi accepted if the set of states visited infinitely often does not intersect F.

Alternating Tree Automaton
An automaton with an arbitrary branching mode running on trees.

Atomic Proposition
An atomic proposition of a formula in a propositional logic corresponds to signals in a design or implementation.

AWT
Alternating Weak Tree Automaton. An alternating tree automaton with a particularly structured state space. The states are partitioned into partially ordered sets. Each set is classified as accepting or rejecting. The transition function is restricted so that in each transition, the automaton either stays at the same set or moves to a set smaller in the partial order.

Branching Mode
The branching mode is a way to classify automata. We distinguish between four branching modes: Deterministic, nondeterministic, universal, and alternating. In a deterministic automaton, the transition function maps from state and letter to a single state. The transition functions of nondeterministic and universal automata map to sets of states. The automata differ in the way they accept an input word or tree. In a nondeterministic automaton the suffix of the word or tree should be accepted by one of the states in the set. In the universal automaton all states in the set have to accept the suffix. An alternating automaton can have nondeterministic and universal edges.

Infinite Game
A finite state machine on which two players, the protagonist and the antagonist, determine the run, by each determining part of the input. The game comes with a winning condition and the task of the protagonist is to make sure that the run satisfies this condition.

Language Emptiness
The language of an automaton is empty iff the automaton accepts no input object (word or tree), that means there is no accepting run for this automaton.

LTL
Linear Temporal Logic or Linear-time temporal logic. LTL is a temporal logic for property specification in formal verification [Pnu77].

LTL Game
An infinite game where the winning condition is given as LTL formula. All plays in which the sequence of states visited fulfill the given formula are winning for the protagonist. Otherwise the antagonist wins.

Mu Calculus
A calculus of predicates and binary relations which enables writing and solving relational equations among states.

NBT
Nondeterministic Büchi Tree Automaton. An alternating tree automaton with Büchi acceptance condition and nondeterministic branching mode.

NBW
Nondeterministic Büchi Word Automaton. An alternating automaton with Büchi acceptance condition and nondeterministic branching mode. The automaton runs on words.

PSL
Property Specification Language, the language for specification of designs upon which PROSYD is based.

PSL Game
Similar to an LTL game but with a PSL formula as winning condition.

Realizable
A given formula ψ over a sets of input I and output O signal is realizable if there exists a strategy $f : (2^I)^* \rightarrow 2^O$ such that all the computations of the system generated by f satisfy ψ. Intuitively, a specification is realizable if there exists a system that can respond in such a way that independent of the input values the environment chooses the combination of inputs and outputs always fulfills the given formula.

Synthesis
The process of automatically generating a design from a given specification. Formally, check if the given specification is realizable and find a witness.

UCT
Universal co-Büchi Word Automaton. An alternating tree automaton with co-Büchi acceptance condition and universal branching mode.

Winning Strategy
A recipe with which a player is guaranteed to win an infinite game, no matter what the other player does. A finite state strategy may depend on a finite memory of the past, i.e., the move the strategy suggests can depend on previous moves of the two players. A memoryless strategy depends only on the current state of the game.

1 Introduction

In this document we introduce our tool Lily, a LInear Logic sYnthesizer. We describe what Lily is and what it can do. We explain how to use Lily and provide a running example. Furthermore, we explain some details on the implementation and on the test suite. Finally, we present the theoretical background [JB06a, JB06b].

1.1 What is Lily?

Lily is a linear logic synthesizer, which synthesizes a functionally correct design from a formal specification. Lily is a command-line tool written in Perl. Lily takes a set of PSL or LTL properties and a partition of the used signals into input and output signals. If the given specification is realizable, Lily provides a design with the stated input and output signals that fulfills the specification. The design is a state machine represented as a VERILOG module or as a directed graph in DOT format. Lily is implemented on top of Wring [SB00, GBS02], a toolkit for linear logics and automata on infinite words.

1.2 Why use Lily?

Writing both a specification and an implementation and subsequently checking whether the latter satisfies the former seems wasteful. A much more attractive approach is to automatically construct the implementation from the specification, leaving the designer with only the task of ensuring that the specification describes the intended behavior. The benefit is even more pronounced when one takes into effect the costs for debugging the manual implementation, and of redesigning it when the specification changes.

Due to the complexity of the problem the size of the specification is limited. Nevertheless, the ability to synthesize small specifications is also very useful. For instance, it can be used to synthesize functional models on the block level or it can help engineer to get familiar with properties more easily.

Our tool provides several optimizations to make synthesis more competitive. We have applied our optimizations to synthesize several examples and achieved a significant improvement over the straightforward implementation. Lily constitutes the first implementation of a synthesis algorithm for the linear-part of PSL. We believe that the optimizations implemented in our tool and discussed in Section 5 form an important step towards making linear-time synthesis practical.

1.3 Features List

Table 1 reports the status of the features stated in the Description of Work document for this tool.

The list contains *mandatory*, *desirable*, and *nice to have* features, with the intention that the minimal requirement for this deliverable is the implementation of all mandatory features.

Other features are not explicitly requested to fulfill the due of the deliverable. We implemented all mandatory and desirable features.

	Present	Reference
Mandatory Features		
Pointers to algorithms used	YES	5
List of target operating systems	YES	3.1
Explanation of coding standards	YES	4.1
Discussion of license issues	YES	3.2
User documentation, including documentation of user interface (command line switches) and imported/exported file formats	YES	2, Appendix
Test suite	YES	4.2
Standard Input Language - PSL	YES	2.1
Support for Verilog Flavor	YES	2.1, Appendix
Synthesis of the linear part of PSL(LTL-like)	YES	5
Outputs Verilog	YES	2.4
Desirable features		
Efficient for “weak” properties (weakness expresses an automata-theoretic notion of expressibility)	YES	5.3 (First Section)
Nice to have features		
Support for other flavors	NO	
Efficient synthesis for all of the linear subset of PSL	NO[a]	

Table 1: Table of features

---

[a] We have implemented optimizations to speed up the synthesis process for “strong” properties as well. Even though the optimizations work very well for many cases, there are still specifications where they do not help. (cf. First Section of 5.3)

1.4 History of Synthesis

LTL synthesis was proposed in [PR89]. The key to the solution is the observation that a program with input signals I and output signals O can be seen as a complete Σ-labeled D-tree with $\Sigma = 2^O$ and $D = 2^I$: the label of node $t \in D^*$ gives the output after input sequence t. The solution proposed in [PR89] is to build a nondeterministic Büchi word automaton for the specification and then to convert this automaton to a deterministic Rabin automaton that recognizes all Σ-labeled D-trees satisfying the specification. A witness to the nonemptiness of the automaton is an implementation of the specification.

There are two reasons that this approach has not been followed by an implementation. The first reason is that synthesis of LTL properties is 2EXPTIME-complete [Ros92]. The second is that the solution uses an intricate determinization construction [Saf88] that is hard to implement and very hard to optimize. The first reason should not prevent one from implementing the approach. After all, the bound is a lower bound and a manual implementation is also subject to it. (Cf. [Var05].) Thus, the complexity of verifying the specification on a manual implementation is not lower than that of automatically synthesizing the design. In combination with the second reason, however, the argument gains strength. For many specifications, a doubly-exponential blow up is not necessary but can only be avoided through careful use of optimization techniques. Safra's determinization construction turned out to be very resistant to efficient implementations[ATW05].

In order to deal with these complexity issues, previous implementations on LTL synthesis focuses on restricted subsets of LTL [WHT03, Har05, PPS06]. The approach of Piterman, Pnueli, and Sa'ar [PPS06] handles the most general subset. Their approach is applicable to specifications expressible with a generalized Streett[1] acceptance condition.

Recently, Kupferman and Vardi [KV05] proposed an alternative to the standard approach. Starting from a specification φ over $I \cup O$, they generate, through the nondeterministic Büchi word automaton for $\neg\varphi$, a universal co-Büchi tree automaton that accepts all trees satisfying φ. From that they construct an alternating weak tree automaton accepting at least one (regular) tree satisfying φ (or none, if φ is not realizable). Finally, the alternating automaton is converted to nondeterministic Büchi tree automaton with the same language. A witness for the nonemptiness of this automaton is an implementation of φ. The approach is applicable to any linear logic that is closed under negation and that can be compiled to nondeterministic Büchi word automata.

Our implementation is based on this approach. It is the first to handle the complete language and does not impose any syntactic requirements on the specification.

2 Usage

This sections explains how to use Lily. Lily takes a *specification* and a *partition file* as input and provides a VERILOG and a DOT version of the generated design. In the first two subsections, we explain the purpose and the syntax of the specification and the partition file. Then we show how to call Lily and explain the available command line options. Finally, we talk about the generated output files.

2.1 Specification File

The *specification file* holds a formal specification written in the linear-part of PSL or in LTL. The tool distinguishes between the language due to the file-extension. Files ending with ".psl" are recognized as PSL files. Files ending with ".ltl" are recognized as LTL files. Table 2 shows the Boolean and temporal operators recognized by Lily for the PSL and the LTL flavor.

The two flavors also differ in the way they handle variables. In LTL flavor, we have to assign a Boolean value (0 or 1) to each variable. In PSL flavor the assignment can be omitted. Those variable are assigned to 1 by default.

In both flavors the keywords `assert` and `assume` can be use to distinguish between assumptions on the environment and assertions the system has to fulfill. If the keywords are omitted we synthesize the conjunction of all formulas.

Table 2: Operators recognized by Lily

Boolean operator	**LTL flavor**	**PSL flavor**
And	`*`	`&,&&`
Or	`+`	`|,||`
Imply	`->`	`->`
Equivalent	`<->`	`<->`
Not	`!`	`!`

Temporal operator	**LTL flavor**	**PSL flavor**
Next	`X`	`next`
Existential Next	-	`next_e[n:m]`
Universal Next	-	`next_a[n:m]`
Strong Until	`U`	`until!`
Strong Release	`R,V`	-
Always	`G`	`always`
Strong Eventually	`F`	`eventually!`

We present an example to show what the formulas and the corresponding specification files look like. A detailed syntax description for the specification file can be found in Appendix A.1.

Example 1. *We specify a small traffic light system for a crossing of a highway and a farm road. The systems has only two lights, which are either green or red. Signals* hl *and* fl*, which are output signals, encode these two lights. The highway light is green iff* $hl = 1$*, and similarly for the crossing farm road and* fl*. The input signal* car *indicates that a car is waiting at the farm road and* $timer$ *represents the expiration of a timer. The specification assumes that the timer expires regularly. It requires that a green lamp stays green until the timer expires. Furthermore, one of the lamps must always be red, every car at the farm road is eventually allowed to drive on, and the highway lamp is regularly set to green. Below we show the specification file for Lily in* PSL *and* LTL *flavor.*

Specification file for Example 1 in PSL flavor

```
assume always(eventually!(timer));
assert always(!(hl & fl));
assert always(eventually!(hl));
assert always(car -> eventually!(fl));
assert always(hl -> (hl until! timer));
assert always(fl -> (fl until! timer));
```

Specification file for Example 1 in LTL flavor

```
G(F(timer=1)) -> (G(fl=1 -> (fl=1 U timer=1)) *
                  G(hl=1 -> (hl=1 U timer=1)) *
                  G(car=1 -> F(fl=1)) *
                  G(F(hl=1)) *
                  G(!(hl=1 * fl=1)));
```

2.2 Partition File

The partition file divides the signals used in the specification file into input and output signals. In Example 1 we have the four signals car, $timer$, fl, and hl. The first two are input signals, the later are output signals. The corresponding partition file is shown below and a detailed syntax description is provided in Appendix A.2.

Partition file for Example 1

```
.inputs timer car
.outputs hl fl
```

2.3 Command Line Options

Lily is invoke with the command `ltl2aut.pl`. All command line options valid in Wring are valid in Lily as well, since Lily uses Wring to construct a Büchi automaton in its first step. Below we show the original Wring command and the new Lily command.

Wring Command

```
ltl2aut.pl [-c {0,1}] [-f formula] [-h]  [-ltl file]
           [-m method] [-o {0,1}] [-p prefix] [-s {0,1}] [-v n]
           [-ver {0,1}] [-auto file] [-mon file]
```

Lily Command

```
ltl2aut.pl [-c {0,1}] [-f formula] [-h]  [-ltl file]
           [-m method] [-o {0,1}] [-p prefix] [-s {0,1}] [-v n]
           [-ver {0,1}] [-auto file] [-mon file]
           [-syn file] [-syndir synthesisDir] [-mc]
           [-ouct {0,1}] [-oawt {0,1}] [-owit {0,1}]
           [-omh {0,1}] [-omhc {0,1}]
           [-oedges {0,1}] [-orelease {0,1}] [-oreuse {0,1}]
```

With the command line options inherited from Wring the user can determine the name of the specification file, the prefix for the output files, verbosity, and parameters for the construction of Büchi automata provided by Wring. A detailed description of those options is shown in Table 3.

Table 3: Command line options inherited from Wring

Command	**Result**	**Example**
`-c num`	Iff `num` $\neq 0$, make the transition relation of the automaton complete. Off by default.	`-c 1`
`-comp`	Build Büchi automaton and its complement for the given LTL formula.	`-comp`
`-f formula`	The LTL formula to be translated. Use either `-ltl` or `-f`.	`-f '!(G(F(q=1)))'`
`-h`	Gives help on the usage.	`-h`
`-ltl file`	File containing the LTL formulae to be translated. Use either `-ltl` or `-f`.	`-ltl spec1.ltl`
`-m method`	Sets the method used in translation. Method ranges over GPVW, GPVW+, LTL2AUT, Boolean. Default is Boolean.	`-m LTL2AUT`
`-o {0,1}`	Optimize the automaton after translation, using simulation relations. On by default.	`-o 1`
`-p prefix`	Sets the prefix of the files that are written. Default values is `ltl2aut`.	`-p example1`
`-s num`	Iff `num` $\neq 0$, simplify the formula before translating it, using rewriting. On by default.	`-s 1`
`-v level`	Sets the verbosity level ($0 \leq$ `level` ≤ 4). Default is 1.	`-v 2`
`-ver num`	Iff `num` $\neq 0$, make an attempt at verifying the automaton. Off by default.	`-ver 1`
`-mon file`	Write a VERILOG monitor to file.	`-mon monitor.v`
`-auto file`	Read-in the automaton described in `file` and optimizes it. This automaton can be used as specification for the synthesis process of Lily as well. See Table 4 for a detailed description of using `-auto` option with Lily.	`-auto nbw1.aut`

Lily has new command line options to invoke the synthesis process, to define the name of the partition file, to specify an output directory, to verify the generated design, and to switch various optimizations on and off. By default all optimizations are turned on. The user need not care about those options. In Table 4 we list and describe all available options.

Let us continue the traffic light example. If the specification is stored in the file `tl.psl` and the partition is stored in the file `tl.part` the simplest way to call Lily is to use one of the following commands:

```
ltl2aut.pl -syn tl.part -ltl tl.psl  or
ltl2aut.pl -syn tl.part -ltl tl.psl -syndir trafficlight
```

The output file are stored in the current directory or in the new directory `trafficlight` depending on the chosen command.

Table 4: Command line options for Lily

Command	Result	Example
`-syn file`	Synthesizes the formula (given with `-f` or `-ltl`) to VERILOG code using the signal partition stored in `file`.	`-syn ex1.part`
`-syndir dir`	Only valid with `-syn` option. `dir` is the name of the directory in which all results of the synthesis process are stored. If `-syndir` is not set the result files are stored in the current directory.	`-syndir results`
`-auto file`	Read-in the automaton described in `file`. Use the following file-extensions to defined the type of automaton to read (see Appendix A.4 for a syntax description.) `aut` for a state labeled NBW (default) `l2a` for a transition labeled NBW `uct` for an UCT The automaton specifies the allowed behavior of the system to construct. This options overwrites the specification given with `-f` or `-ltl`.	`-auto count.l2a`
`-mc`	Only valid with `-syn` and `-ltl` option. Modelcheck the result of the synthesis process using the program Vis [B+96]. To use this option Vis has to be installed and in the search path.	`-mc`
`-ouct {0,1}`	Optimize the universal co-Büchi tree automaton, using game and simulation-based optimizations (see Section 5.3). On by default.	`-ouct 1`
`-oawt {0,1}`	Optimize the alternating weak tree automaton, using game and simulation-based optimizations (see Section 5.3). On by default.	`-oawt 1`
`-owit {0,1}`	Optimize the witness/strategy, using simulation relation (see Section 5.3.) On by default.	`-owit 0`
`-omh  {0,1}`	Use Fritz' optimizations (see Section 5.3) during Miyano and Hayashi's construction.	`-omh  1`
`-omhc {0,1}`	Combine Miyano and Hayashi's construction with language emptiness check (see Section 5.3.)	`-omhc 1`
`-oedges {0,1}`	Merge direction by applying Theorem 11 of Section 5.3.	`-oedges 1`
`-orelease {0,1}`	Restrict release function to stay in odd layer if possible (Theorem 13).	`-orelease 0`
`-oreuse {0,1}`	Reuse the result from previous computations with lower ranks (see Section 5.3.)	`-oreuse 1`

2.4 Output Files

Lily provides a VERILOG module and a graphical state diagram of the the generated design. We use DOT format to store the state diagram. Files in DOT format can be translated using `dot` [GVZ]. See Appendix A.3 for a syntax description of the generated DOT files.

By default Lily generated the following two files:

`ltl2vl-verilog.v`

`ltl2vl-synthesis.dot`

If the specification is realizable `ltl2vl-verilog.v` holds the VERILOG module of the generated design. The state diagram of the generated design is stored in `ltl2vl-synthesis.dot`. If the specification is not realizable both files state that the given specification is unrealizable. Note that the prefix `ltl2vl` can be replaced by a user defined prefix with the option `-p`.

The specification we used in our traffic light example is realizable and the design generated by Lily is shown in Figure 1. The corresponding state diagram is shown in Figure 2.

```
module synthesis(fl,hl,clk,car,timer);
  input  clk,car,timer;
  output fl,hl;
  wire clk,fl,hl,car,timer;
  reg [1:0] state;

   assign hl = (state == 0)||(state == 2);
   assign fl = (state == 1);

  initial begin
    state = 0; //n15_1n18_1
  end
  always @(posedge clk) begin
    case(state)
    0: begin //n15_1n18_1
      if (car==0) state = 0;
      if (car==1 && timer==1) state = 1;
      if (car==1 && timer==0) state = 2;
    end
    1: begin //n12_1n18_1
      if (timer==1) state = 0;
      if (timer==0) state = 1;
    end
    2: begin //n10_1n15_1n18_1
      if (timer==0) state = 2;
      if (timer==1) state = 1;
    end
    endcase
  end
endmodule //synthesis
```

Figure 1: Generated design for a simple traffic light

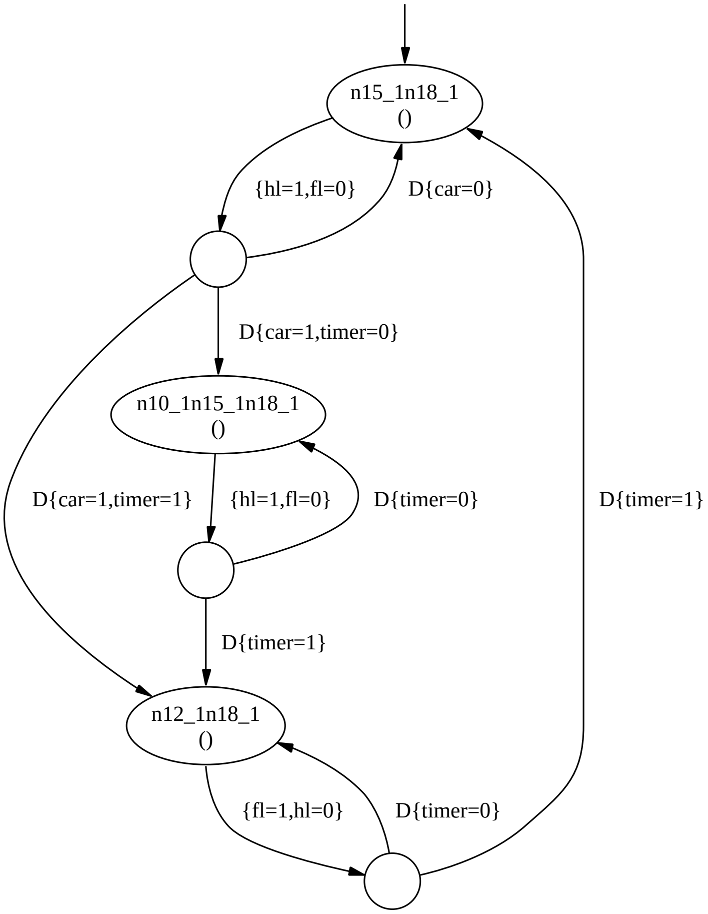

Figure 2: State diagram of the generated traffic light

3 Installation

In this section we provide information about installation related issues including system requirements, license issues, and a guide to install Lily.

3.1 System Requirements

Lily was developed on a Gentoo GNU/Linux based x86 machine with a 2.6.14 kernel using Perl 5. It should run on any similar machine that runs

- Perl 5.8.8 or higher [PRL].

If used with `-mc` option Lily also requires

- Vis release 2.1 or higher [VIS].

3.2 License Issues

3.3 Installing Lily

Follow the four steps below to install Lily.

1. Download Lily source files (`lily.tar.gz`) from
   `http://www.ist.tugraz.at/staff/jobstmann/lily/lily.tar.gz`
2. Unpack sources using
   `tar xvfz lily.tar.gz`
   to target directory (e.g., `/opt/lily`).
3. Add source directory to the perl library path, e.g.,
   `export PERL5LIB=/opt/lily:${PERL5LIB}` or
   `setenv PERL5LIB /opt/lily:${PERL5LIB}`
   Lily includes its own Wring version. If you have installed another version of Wring, add the source directory to the beginning of the library path to ensure that the right version is used. The same holds for setting the search path explained below.
4. Add source directory to the search path, e.g.,
   `export PATH=/opt/lily:${PATH}` or
   `setenv PATH /opt/lily:${PATH}`

4 Technical Details

In this section we give some details about how we implemented and tested Lily. In the first part, we talk about the programming language and provide a diagram of the program structure. In the second part, we discuss the examples we used to test our implementation.

4.1 Implementation

Lily is written in Perl 5. Perl is a dynamic procedural programming language, which summarizes features from C, shell scripting, AWK, sed, Lisp, and many other programming languages in an easy-to-use way. In Perl 5, features were added that support complex data structures, first-class functions and an object-oriented programming model. We make extensive use of these features. Lily is implemented according to the object-oriented paradigm.

Figure 3 shows a block diagram of the structure illustrating the major parts of Lily and the connection between Lily and Wring. The rounded rectangles represent the major functional parts and the wavelike rectangles represent the data structures. Rounded rectangles in grey belong to Lily. The single rounded rectangle in white represents Wring. Wring, from the University of Colorado [SB00, GBS02], is an academic toolkit for linear logics and automata on infinite words. It contains a translator from LTL to nonderministic Büchi word automata and various transformation and optimization algorithms for such automata which were of use for the synthesis tool.

The synthesis approach we implemented consists of a sequence of automata translations and corresponding optimizations (see Section 5 for more details). Each type of automata and the translations and optimizations applicable to it form a separated part of our tool.

- **Wring:** Block to construct and manipulate nondeterministic Büchi word automata.
- **BuildUCT:** Block to construct and manipulate universal co-Büchi tree automata.
- **BuildAWT:** Block to construct and manipulate alternating weak tree automata.
- **BuildNBT:** Block to construct and manipulate nondeterministic Büchi tree automata.
- **BuildFSM:** Block to construct and manipulate finite state machines.

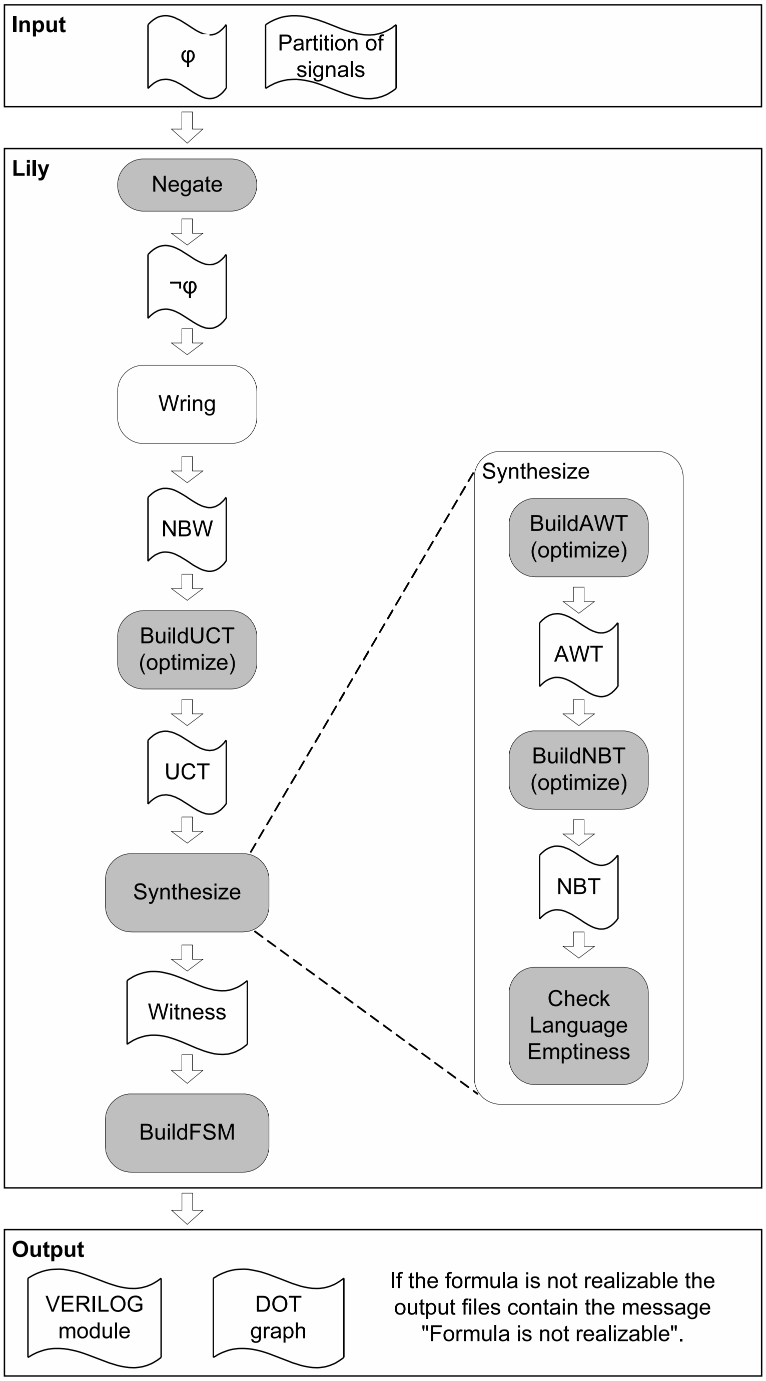

Figure 3: Blockdiagram of Lily

The block **Negate** takes an LTL formula and builds its negation. Finally, the block **Check Language Emptiness** takes a nondeterministic Büchi tree automata and check if the language of the automaton is empty and provides a witness if the language is not empty.

4.2 Test Suite

We have performed tests with formulas generated by the Wring random formula generator. Even though we used different partitions of the atomic propositions into input and output signals, only a few of these formulas could be synthesized. Most formulas were either unrealizable or Lily could not tell because the UCT was not weak and the bound on k was too high (see Section 5.3 for the meaning of k). Furthermore, we are interested in meaningful specifications to see the relation between our design intent and the generated design. Thus, we concentrated on hand-written formulas.

We show the effectiveness of the various optimizations by synthesizing 20 hand-written formulas. Our examples are small, but we show a significant improvement over the straightforward implementation.

For realizable formulas, we verified the output of our tool with a model checker. In the case of unrealizability we negated the formula, switched the input and output signals, and tried to synthesize an environment that forces any system to violate the formula. Since we synthesize Moore machines, this is not always possible. For instance, $\texttt{always}\,(r \leftrightarrow a)$ with input r and output a can not be realized as a Moore machine, and neither can $\neg\texttt{always}\,(r \leftrightarrow a)$ with input a and output r. In such cases, we have verified the result by hand, which is a tedious job even for small formulas.

5 Underlying Theory

In this section we explain the algorithms using in Lily [JB06a, JB06b]. We start with introducing the necessary definitions in Section 5.1. In Section 5.2 we describe a game-based and a simulation-based optimization that can be used on any tree automaton. In Section 5.3, we recall the construction of Kupferman and Vardi [KV05] and discuss how we implemented it efficiently.

5.1 Definitions

We assume that the reader is familiar with the μ-calculus and PSL. For an introduction see [MP91, CGP99]. We will use the linear time fragment of PSL to specify the behavior of a system. Properties will use the set $I \cup O$ of atomic propositions, where I denotes the input signals and O the output signals.

A Σ-labeled *D-tree* is a tuple (T,τ) such that $T \subseteq D^*$ is prefix-closed and $\tau : T \rightarrow \Sigma$. The tree is *complete* if $T = D^*$. The set of all Σ-labeled D-trees is denoted by $T_{\Sigma,D}$.

An *alternating tree automaton* for Σ-labeled D-trees is a tuple $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ such that Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \rightarrow 2^{2^{D \times Q}}$ is the *transition relation* (an element $C \in 2^{D \times Q}$ is called a *transition*) and $\alpha \subseteq Q$ is the *acceptance condition*. We denote by A^q, for $q \in Q$, the automaton A with the initial state q.

A run (R,ρ) of A on a Σ-labeled D-tree (T,τ) is a $T \times Q$-labeled $\mathbb{N}$-tree satisfying the following constraints:

1. $\rho(\varepsilon) = (\varepsilon, q_0)$.
2. If $r \in R$ is labeled (t,q), then there is a set $\{(d_1,q_1),\ldots,(d_k,q_k)\} \in \delta(q,\tau(t))$ such that r has k children labeled $(t \cdot d_1, q_1),\ldots,(t \cdot d_k, q_k)$.

We have two acceptance conditions: Büchi and co-Büchi. A run (R,ρ) of a Büchi (co-Büchi) automaton is accepting if all *infinite* paths of (R,ρ) have infinitely many states in α (only finitely many states in α). The language $L(A)$ of A is the set of trees for which there exists an accepting run.

An ABT induces a graph. The states of the automaton are the nodes of the graph and there is an edge from q to q' if (d',q') occurs in $\rho(q,\sigma)$ for some $\sigma \in \Sigma$ and $d' \in D$. The automaton is *weak* if each strongly connected component (SCC) contains either only states in α or only states not in α.

Intuitively, A is a top-down tree automaton for infinite trees. A run of A is also a tree. The nodes are labeled with pairs (t,q) meaning that A is in state q in node t

of T. Because A is alternating, it can be in multiple states simultaneously for any given node: For a given t there can be multiple q_i and nodes labeled (t, q_i) in R. The automaton starts at the root note in state q_0. If it is in state q in state t of the input tree, and t is labeled σ, then $\delta(q,\sigma)$ tells A what to do next. The automaton can nondeterministically choose a $C \in \delta(q,\sigma)$. Then, for all $(d', q') \in C$, A moves to node $t \cdot d'$ in state q'. (The transition relation $\delta(q,\sigma)$ can be considered as a DNF formula over $D \times Q$.) Note that there are no runs with a node (t,q) for which $\delta(q,\tau(t)) = \emptyset$. On the other hand, a run that visits a node t needs not visit all of its children; there are no restrictions on the subtrees rooted in a node that is not visited. In particular, a node (t,q) such that $\delta(q,\tau(t)) = \{\emptyset\}$ does not have any children, and there are no restrictions on the subtree rooted in t.

An automaton is *universal* if $|\delta(q,l)| = 1$. A universal automaton has at most one run for a given input. An automaton is *nondeterministic* if for all $q \in Q, \sigma \in \Sigma, C \in \delta(q,\sigma)$ and $(d_i, q_i), (d_j, q_j) \in C$ we have $d_i = d_j$ implies $q_i = q_j$. That is, the automaton can only send one copy in each direction and a run is isomorphic to the input tree. An automaton is deterministic if it is both universal and nondeterministic.

An automaton is a word automaton if $|D| = 1$. In that case, we can leave out D altogether.

We will abbreviate alternating/nondeterministic/universal/deterministic Büchi/co-Büchi/weak tree/word automaton as a three letter acronym: A/N/U/D B/C/W T/W.

We will use Σ-labeled D-trees to model programs with input alphabet D and output alphabet Σ. In order to establish a link with the PSL specification, we will assume that $D = 2^I$ and $\Sigma = 2^O$. Thus, a path of a Σ-labeled D-tree can be seen as a word over $(\Sigma \cup D)^\omega$: we merge the label of the node with the direction edge following it in the path. Given a word language $L \in (\Sigma \cup D)^\omega$, let $T(L) \subseteq T_{\Sigma,D}$ be the set of trees T such that all paths of T are in L. For a word automaton A we will write $T(A)$ for $T(L(A))$. Similarly, we will write $T(\varphi)$ for the set of trees T such that every path of T satisfies the PSL formula φ.

A *Moore machine* with output alphabet Σ and input alphabet D is a tuple $M = (\Sigma, D, S, s_0, T, G)$ such that S is a finite set of states, $s_0 \in S$ is the initial state, $T : S \times D \to S$ is the transition function, and $G : S \to \Sigma$ is the output function. We extend T to the domain $S \times \Sigma^*$ in the usual way. The *input/output language* $L(M)$ of M is

$$\{\pi \in (\Sigma \cup D)^\omega \mid \pi = ((\sigma_0, d_0), (\sigma_1, d_1), \ldots), \sigma_n = G(T(q_0, d_0 ... d_{n-1}))\}.$$

Every Moore machine corresponds to a complete Σ-labeled D-tree for which every node $t \in D^*$ is labeled with $G(T(q_0, t))$. Thus, every tree language $T \subseteq T_{\Sigma,D}$ defines a set $\mathcal{M}(T)$ of Moore machines: those machines M for which $T(L(M)) \in T$. (Note that not every tree can be defined by a Moore machine and thus there are T for which $\bigcup\{T(L(M)) \mid M \in \mathcal{M}(T)\}) \neq T$).

5.2 Simplifying tree automata

In this section we discuss two optimizations that can be used for any tree automaton.

Simplification Using Games

We define a sufficient (but not necessary) condition for language emptiness of A^q.

Our heuristic views the alternating automaton as a game which is played in rounds. In each round, starting at a state q, the protagonist decides the label $\sigma \in \Sigma$ and a set $C \subseteq \delta(q, \sigma)$ and the antagonist chooses a pair $(d, q') \in C$. The next round starts in q'. If $\delta(q, \sigma)$ or C are empty the play is finite and the player who has to choose from an empty set loses the game. If a play is infinite the winner is determined by the acceptance condition. For an ABT, the protagonist wins the play if the play visits the set of accepting states α infinitely often. For a ACT, the protagonist wins if from some point on the play avoids α. A strategy s maps a finite sequence of states $q_0, \ldots, q_k$ to a set $C \subseteq \delta(q_k, \sigma)$ for some a label $\sigma \in \Sigma$. A play $q_1, q_2, \ldots$ adheres to a strategy s if for every k, $s(q_0, \ldots, q_k) = C$ implies that there is a pair $(d, q_{k+1}) \in C$. The game A^q is won if there is a strategy such that all plays starting at q that adhere to the strategy are won. We call q a winning state and the set of all winning states is called the winning region.

If the game is lost, then $L(A^q)$ is empty. In the case of an NBT (NCT) the converse holds as well. However, in general it does not. A counterexample would be a word automaton such that (1) $\delta(q_0, \sigma) = q_1 \wedge q_2$ for all σ, (2) $L(A^{q_1}) \cap L(A^{q_2}) = \emptyset$, and (3) the games A^{q_1} and A^{q_2} are won. In this case, the game A^q is also won. Note that computing a necessary and sufficient condition in polynomial time is not possible as this would give us an EXPTIME algorithm for deciding realizability.

The game is computed as follows. Let

$$\begin{aligned} \langle \mathsf{P} \rangle \mathsf{X},(S) &= \{q \in Q \mid \exists \sigma \in \Sigma, C \in \delta(q, \sigma) : \forall (d, q') \in C : q' \in S\}, \\ W_B(S) &= \nu Y. \langle \mathsf{P} \rangle \mathsf{X},(\mu Z. Y \wedge (S \vee \langle \mathsf{P} \rangle \mathsf{X}, Z)), \text{ and} \\ W_C(S) &= \mu Y. \langle \mathsf{P} \rangle \mathsf{X},(\nu Z. Y \vee (S \wedge \langle \mathsf{P} \rangle \mathsf{X}, Z)). \end{aligned}$$

In an ABT (ACT) with acceptance condition α, we can discard the states outside of $W_B(\alpha)$ ($W_C(\alpha)$, resp.).

Theorem 2. *Given an ABT (ACT) $A = (\Sigma, D, Q, q_0, \delta, \alpha)$, let $W = W_B(\alpha)$. ($W = W_C(\alpha)$, resp.) Let the ABT (ACT) $A' = (\Sigma, D, Q', q'_0, \delta', \alpha')$ with $Q' = Q \cap W$, $\alpha' = \alpha \cap W$, and $\delta'(q, \sigma) = \{C \mid C \in \delta(q, \sigma), \forall (d, q') \in C, q \in W\}$. If $q_0 \in W$ then $q'_0 = q_0$, otherwise q'_0 is some state in Q' with an empty language.*

We have $L(A^q) = L(A'^q)$ for all $q \in Q'$ and in particular, $L(A) = L(A')$. ☐

Simplification Using Simulation Relations

The second optimization uses (direct) simulation minimization on alternating tree automata. Simulation minimization on nondeterministic word automata is well established. Our construction generalizes that for alternating word automata [AHKV98, FW02, GKSV03].

Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. The direct simulation relation $\preceq \subseteq Q \times Q$ is the largest relation such that $u \preceq v$ implies that

1. $u \in \alpha$ implies $v \in \alpha$, and
2. $\forall \sigma \in \Sigma, C_u \in \delta(u,\sigma)\ \exists C_v \in \delta(v,\sigma) : \forall d' \in D, (d',v') \in C_v\ \exists (d',u') \in C_u : u' \preceq v'$.

If $u \preceq v$, we say that u is simulated by v. If additionally, $u \succeq v$, we say that u and v are simulation equivalent, denoted $u \simeq v$.

Lemma 3. *If $u \preceq v$ then $L(A^u) \subseteq L(A^v)$.* ☐

The following theorems are tree-automaton variants of those presented in [GKSV03] for optimizing alternating word automata. The first theorem allows us to restrict the state space of an ABT to a set of representatives of every equivalence class under $\simeq$.

Theorem 4. *Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT, let $u, v \in Q$, and suppose $u \simeq v$. Let $A' = (\Sigma, D, Q \setminus \{u\}, q'_0, \delta', \alpha)$, where $q'_0 = v$ if $q_0 = u$ and $q'_0 = q_0$ otherwise, and δ' is obtained from δ by replacing u by v everywhere. Then, $L(A) = L(A')$.* ☐

The following two theorems allow us to simplify the relations of an NBT.

Theorem 5. *Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT, let $u, v \in Q$, and suppose $u \neq v$ and $u \preceq v$. For $C \subseteq D \times Q$, let*

$$C' = \begin{cases} C \setminus (d,v) & \text{if } \exists d : (d,u) \in C, \\ C & \text{otherwise.} \end{cases}$$

Let $A' = (\Sigma, D, Q, q_0, \delta', \alpha)$, where for all q and σ we have $\delta'(q,\sigma) = \{C' \mid C \in \delta(q,\sigma)\}$. We have $L(A) = L(A')$. ☐

Theorem 6. *Let $A = (\Sigma, D, Q, q_0, \delta, \alpha)$ be an ABT. Suppose $C, C' \in \delta(q,\sigma)$, $C \neq C'$, and for all d and $(d,q') \in C'$ there is a $(d,q) \in C$ such that $q \preceq q'$. Let $A = (\Sigma, D, Q, q_0, \delta', \alpha)$ be an ABT for which δ' equals δ except that $\delta'(q,\sigma) = \delta(q,\sigma) \setminus C$. We have $L(A) = L(A')$.* ☐

We can simplify an ABT by repeated application of the last two theorems and removal of states that are no longer reachable from the initial state. The simulation relation can be computed in polynomial time, as can the optimizations. (It should be noted that application of the theorems does not alter the simulation relation.)

5.3 Optimizations for Synthesis

Synthesis Algorithm

The goal of synthesis is to find a Moore machine M implementing a PSL specification φ (or to prove that no such M exists). Our approach follows that of [KV05], introducing optimizations that make synthesis much more efficient. The flow is as follows.

1. Construct an NBW A_{NBW} with $L(A_{\mathrm{NBW}}) = \{w \in (\Sigma \cup D)^{\omega} \mid w \not\models \varphi\}$. Let n' be the number of states of A_{NBW}. Note n' is exponential in $|\varphi'|$, if φ is expressible with an LTL formula of the same length and at most doubly-exponential otherwise [BDBF^{+}05].
2. Construct a UCT A_{UCT} with $L(A_{\mathrm{UCT}}) = T_{\Sigma,D} \setminus T(A_{\mathrm{NBW}}) = T(\varphi)$. Let n be the number of states of A_{UCT}; we have $n \leq n'$,
3. Perform the following steps for increasing k, starting with $k = 0$.
   (a) Construct an AWT $A_{\mathrm{AWT}k}$ such that $L(A_{\mathrm{AWT}k}) \subseteq L(A_{\mathrm{UCT}})$ and $L(A_{\mathrm{UCT}}) \neq \emptyset$ implies $L(A_{\mathrm{AWT}k}) \neq \emptyset$; $A_{\mathrm{AWT}k}$ has at most $n \cdot k$ states.
   (b) Construct an NBT $A_{\mathrm{NBT}k}$ such that $L(A_{\mathrm{NBT}k}) = L(A_{\mathrm{AWT}k})$; $A_{\mathrm{NBT}k}$ has at most $(k+1)^{2n}$ states.
   (c) Check for the nonemptiness of $L(A_{\mathrm{NBT}k})$. If the language is nonempty, proceed to Step 4.
   (d) If $k = 2n2^{2n+2}$, stop. Specification φ is not realizable. Otherwise, proceed with the next iteration of the loop. (The bound on k follows from [Pit06].)
4. Compute a witness for the nonemptiness of $A_{\mathrm{NBT}k}$ and convert it to a Moore machine.

If the UCT constructed in Step 2 is weak, synthesis is much simpler: we complement the acceptance condition of A_{UCT} turning it into a UWT, a special case of an AWT. Then, we convert the UWT into an NBT A_{NBT} as in Step 3b. If $L(A_{\mathrm{NBT}})$ is nonempty, the witness is a Moore machine satisfying φ, if it is empty, φ in unrealizable. In this case, we avoid increasing k and the size of the NBT is at most 2^{2n}.

It turns out that in practice, for realizable specifications, the algorithm terminates with very small k, often around three. It should be noted that if the the UCT is not weak it is virtually impossible to prove the specification unrealizable using this approach, because of the high bound on k.

In the following, we will describe the individual steps, discuss the optimizations that we use at every step, and show how to reuse information gained in one iterations of the loop for the following iterations.

NBW

We use Wring [SB00] to construct a nondeterministic generalized Büchi automaton for the negation of the specification. We then use the classic counting construction and the optimizations available in Wring to obtain a small NBW A_{NBW} with $L(A_{\text{NBW}}) = (D \cup \Sigma)^{\omega} \setminus L(\varphi)$.

UCT

We construct a UCT A_{UCT} over Σ-labeled D-trees with $L(A_{\text{UCT}}) = T((\Sigma \cup D)^{\omega} \setminus L(A_{\text{NBW}}))$.

Definition 7. *[KV05] Given an NBW $A_{NBW} = (\Sigma, D, Q, q_0, \delta, \alpha)$, let UCT $A_{UCT} = (\Sigma, D, Q, q_0, \delta', \alpha)$, with for every $q \in Q$ and $\sigma \in \Sigma$*

$$\delta'(q,\sigma) = \big\{\{(d,q') \mid d \in D, q' \in \delta(q, d \cup \sigma)\}\big\}.$$

□

We have $L(A_{\text{UCT}}) = T_{\Sigma,D} \setminus T(A_{\text{NBW}})$.

We can reduce the size of $L(A_{\text{UCT}})$ using game-based simulation and Theorem 2. Optimizing the UCT reduces the time spent optimizing the AWT and, most importantly, it may make the UCT weak, which means that we avoid the expensive construction of the AWT discussed in the next section. Because the UCT is small in comparison to the AWT and the NBT, optimization comes at little cost.

Specifications are often of the form $\varphi \rightarrow \psi$, where φ is an assumption on the environment and ψ describes the allowed behavior of the system. States necessary to ensure that the environment assumptions φ are fulfilled once the system assertion ψ is violated are not necessary. Such states, among others, are removed by the game-based optimization.

Example 8. *We give a small example to show which states will be removed by our algorithm. Let $\varphi =$ `always eventually!` timer $\rightarrow$ `always` (light $\rightarrow$ (light `until!` timer)). Fig. 4 shows a minimal NBW A_{NBW} accepting all words in $\neg\varphi$. Edges are labeled with cubes over the atomic propositions. We partition the atomic propositions into $I = \{light\}$ and $O = \{timer\}$. The UCT A_{UCT} that accepts all 2^O-labeled 2^I-trees not in $T(A_{NBW})$ is shown in Fig. 5. Circles denote states and boxes denote transitions. We label edges starting at circles with cubes over O and edges from boxes with cubes over I. The transition corresponding to a box C consists of all pairs (d,q) such that there is an edge from C to q such that d satisfies the label on the edge. In particular, if d satisfies none of the labels, the branch in direction d is finite, e.g., in state n_2 with light=0 and timer=1. Note that finite branches are accepting.*

Even though the NBW is optimized, the UCT is not minimal: The tree languages $L(A_{UCT}{}^{n3})$ and $L(A_{UCT}{}^{n4})$ are empty. Our algorithm finds both states and replaces them by transitions to false*, removing the part of A_{UCT} to the right of the dashed line. Note that the optimizations cause the automaton to become weak.* □

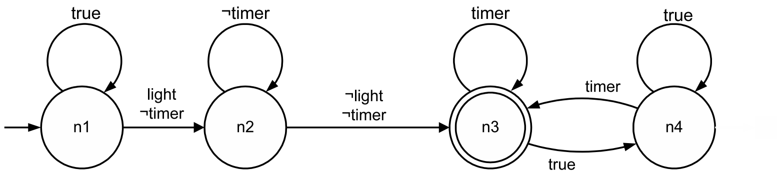

Figure 4: NBW for $\neg\varphi = \texttt{always}\,(\texttt{eventually!}\,(\text{timer})) \wedge \texttt{eventually!}\,(\text{light} \wedge (\neg\text{light}\,\mathsf{R}\,\neg\text{timer}))$

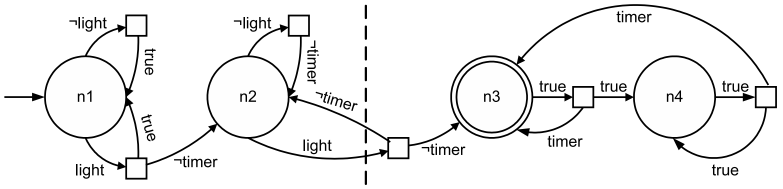

Figure 5: UCT for $\varphi = \texttt{always}\,(\texttt{eventually!}\,(\text{timer})) \rightarrow \texttt{always}\,(\text{light} \rightarrow (\text{light}\,\texttt{until!}\,\text{timer}))$

AWT

From the automaton A_{UCT} we construct an AWT $A_{\text{AWT}k}$ such that $L(A_{\text{AWT}k}) \subseteq L(A_{\text{UCT}})$

Definition 9. *[KV05] Let* $A_{UCT} = (\Sigma, D, Q, q_0, \delta, \alpha)$*, let* $n = |Q|$ *and let* $k \in \mathbb{N}$*. Let* $[k]$ *denote* $\{0, \ldots, k\}$*. We construct* $A_{AWTk} = (\Sigma, D, Q', q'_0, \delta', \alpha')$ *with*

$$\begin{aligned}
Q' &= \{(q,i) \in Q \times [k] \mid q \notin \alpha \text{ or } i \text{ is even}\},\\
q'_0 &= (q_0, k),\\
\delta'((q,i),\sigma) &= \big\{\{(d_1,(q_1,i_1)),\ldots,(d_k,(q_k,i_k))\} \mid \\
&\qquad \{(d_1,q_1),\ldots,(d_k,q_k)\} \in \delta(q,\sigma), i_1,\ldots,i_k \in [i], \forall j : (q_j,i_j) \in Q'\big\}\\
\alpha' &= Q \times \{1,3,\ldots,2k-1\}.
\end{aligned}$$

We call i the rank *of an AWT state (q,i).* ☐

If $k = 2n2^{n+2}$ we have $L(A_{\text{AWT}k}) = \emptyset$ implies $L(A_{\text{UCT}}) = \emptyset$ [KV05, Pit06].

We improve this construction in three ways: by using games, by merging directions, and by using simulation relations.

Game Simulation We can use Theorem 2 to remove states from $A_{\text{AWT}k}$.

Example 10. *Consider the UCT in Fig. 6 and the corresponding AWT in Fig. 7, using $k = 5$. The UCT has been optimized using the techniques discussed in Section 5.3, and the AWT has been optimized in three ways: We have removed states that are not reachable from the initial state, we have merged directions, and we have removed edges. (The last two optimizations are explained in the next sections). Still, there is ample room for improvement of the AWT.*

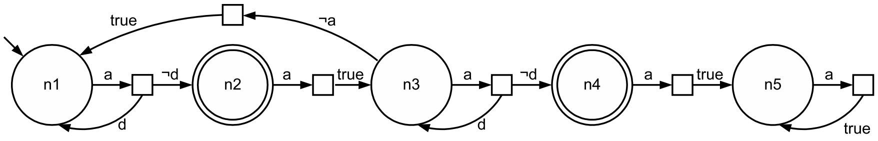

Figure 6: UCT that requires rank 5. Edges that are not shown (for instance from n_4 with label $\neg a$) correspond to labels that are not allowed.

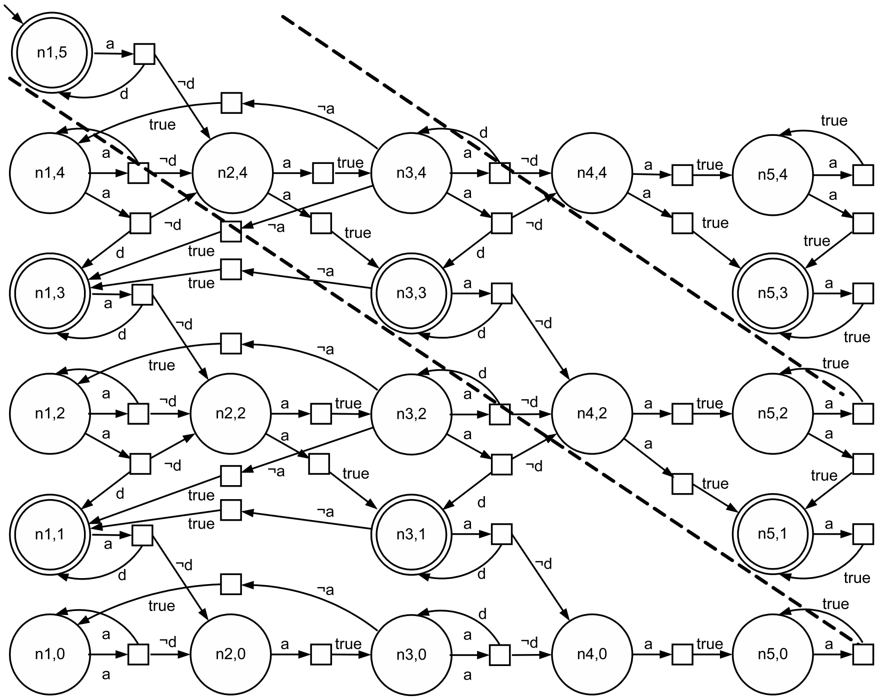

Figure 7: AWT for UCT in Figure 6.

Application of Theorem 2 removes the 12 states below the dashed line on the bottom left and the incident edges. This is a typical situation: each UCT state has an associated minimum rank. □

It should be noted that $A_{\mathrm{AWT}k}$ has a layered structure: there are no states with rank j with a transition back to a state with a rank $i > j$. Furthermore, $A_{\mathrm{AWT}k+1}$ consists of $A_{\mathrm{AWT}k}$ plus one layer of states with rank $k+1$. This implies that game information computed for $A_{\mathrm{AWT}k}$ can be reused for $A_{\mathrm{AWT}k+1}$. A play is won (lost) in $A_{\mathrm{AWT}k+1}$ if it reaches a states that is won (lost) in $A_{\mathrm{AWT}k}$. Furthermore, if (q,j) is won, then so is (q,i) for $i > j$ when i is odd or j is even, which allows us to reuse some of the information computed for states with rank k when adding states with rank $k+1$. This follows from the fact that (q,i) simulates (q,j), as will be discussed in Section 5.3.

Merging Directions Note that δ' may be drastically larger than δ: a single transition $C \in \delta(q,\sigma)$ yields $i^{|C|}$ transitions out of state $(q,i) \in Q'$. Often, however, the transitions in the UCT are the same for many directions, and this fact can be used for to optimize the transition relation.

Theorem 11. *Let* $A''_{AWTk} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ *be as in Definition 9, but with*

$$\delta''((q,i),\sigma) = \{C \in \delta'((q,i),\sigma) \mid \forall (d,(q,j)),(d',(q',j')) \in C : q = q' \rightarrow j = j'\}.$$

We have $L(A''_{AWTk}) = L(A_{AWTk})$. □

Proof. Because $\delta''(q,\sigma) \subseteq \delta'(q,\sigma)$, any tree accepted by $A''_{\mathrm{AWT}k}$ is also accepted by $A_{\mathrm{AWT}k}$.

Let r be a run of $A_{\mathrm{AWT}k}$, we will build a run r'' of $A''_{\mathrm{AWT}k}$. Run r'' is isomorphic to r, using a bijection that maps a node v of r to a node v'' of r''. Run r'' has the same labels as r with the following exception. If node v in r is labeled $(t,(q,i))$ and has children $(t',(q',i'))$ and $(t'',(q',i''))$ with $i' > i''$, then the corresponding children of node v'' of r'' are labeled $(t',(q',i'))$ and $(t'',(q',i'))$.

Because in $A_{\mathrm{AWT}k}$ state (q',i') has all transitions that (q',i'') has, r'' is a run of $A_{\mathrm{AWT}k}$, and because it satisfies the extra condition on δ'' it is also a run of $A''_{\mathrm{AWT}k}$. If r is accepting, then every infinite path π in r gets stuck in an odd rank w from some level l onwards. So starting from l, all children of nodes on π have rank at most w. That implies that the nodes on π in r'' have rank w starting at rank $l+1$ at the latest. Thus, π is still accepting, and since π is arbitrary, r'' is accepting as well. □

This theorem is key to an efficient implementation as it allows us to represent a set of pairs $\{(d_1,q),\ldots,(d_k,q)\}$ as $(\{d_1,\ldots,d_k\},q)$ whenever $\{d_1,\ldots,d_k\}$ can efficiently be represented by a cube over the input signals I.

Simulation minimization We compute the simulation relation on $A_{\mathrm{AWT}k}$ and use Theorems 4, 5, and 6 to optimize the automaton. We would like to point out one optimization in particular.

Lemma 12. *For* $(q,i),(q,j) \in Q'$ *with* $i \geq j$ *such that* i *is odd or* j *is even, we have* $(q,i) \succeq (q,j)$. □

Thus, for any σ, if i is even, we can remove all transitions $C \in \delta((q,i),\sigma)$ that include a pair (q',j) for $j \leq i-2$. If i is odd we can additionally remove all transitions that contain a pair (q',j) with $q \notin \alpha$ and $j = i-1$. That is, odd states become deterministic and for even states there are at most two alternatives to choose from.

Theorem 13. *Let* $A'_{AWTk} = (\Sigma, D, Q', q'_0, \delta'', \alpha')$ *as in Definition 9, but with*

$$\begin{aligned}\delta''((q,i),\sigma) = \Big\{C \in \delta(q,\sigma) \quad \mid \quad & \forall (d',(q',i')) \in C : i' \in \{i-1,i\}, \\ & (i \text{ is even} \vee q' \in \alpha \vee i' = i), \\ & \forall (d'',(q'',i'')) \in C : q' = q'' \rightarrow i' = i''\}.\end{aligned}$$

then $L(A'_{AWTk}) = L(A_{AWTk})$. □

Example 14. *States* $(n_4,4)$, $(n_5,4)$, *and* $(n_5,3)$ *(top right) are simulation equivalent with* $(n_4,2)$, $(n_5,2)$, *and* $(n_5,1)$, *respectively. Using Theorem 4, we can remove states* $(n_4,4)$, $(n_5,4)$, *and* $(n_5,3)$, *and redirect incoming edges to equivalent states.*

Furthermore, the previous removal of the states on the bottom left implies that $(n_3,4) \preceq (n_3,3)$. *Since* $(n_2,4)$ *has identical transitions to* $(n_3,4)$ *and* $(n_3,3)$, *Theorem 6 allows us to remove the transition to* $(n_3,4)$. *Thus,* $(n_3,4)$ *becomes unreachable and can be removed. The same holds for* $(n_5,2)$ *for a similar reason. (This optimization also allows us to remove states* $(n_4,4)$, $(n_5,4)$, *and* $(n_5,3)$, *but Theorem 6 is not in general stronger than Theorem 4.)*

The optimization of the edges due to Theorem 13 is already shown in Fig. 7. Consider, for instance, the transition from $(n_2,4)$ *to* $(n_3,4)$.

Altogether, we have reduced the number of states in the AWT from 22 to 5. The removal of edges is equally important as it reduces nondeterminism and makes the translation to an NBT more efficient. ☐

NBT

The next step is to translate $A_{\mathrm{AWT}k}$ to an NBT $A_{\mathrm{NBT}k}$ with the same language [KV05, MH84].

Assume that $A_{\mathrm{AWT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. We first need some additional notation. For $S \subseteq Q$ and $\sigma \in \Sigma$ let

$$sat(S,\sigma) = \{C \in 2^{D \times Q} \mid C \text{ is minimal set such that} \forall q \in S\, \exists C_q \in \delta(q,\sigma) : C_q \subseteq C\}.$$

For $(S,O) \in 2^Q \times 2^Q$, let

$$sat((S,O),\sigma) = \{(S',O') \in 2^Q \times 2^Q \mid S' \in sat(S,\sigma), O' \in sat(O,\sigma), O' \subseteq S'\}.$$

Furthermore, let $S_d = \{s \mid (d,s) \in S\}$, let $O_d = \{s \mid (d,s) \in O\}$. Let $C_N(S,O) = \{(d,(S_d, O_d \setminus \alpha)) \mid d \in D\}$ and let $C_\emptyset(S) = \{(d,(S_d, S_d \setminus \alpha)) \mid d \in D\}$.

Definition 15. *[KV05, MH84] Let* $A_{NBTk} = (\Sigma, D, 2^Q \times 2^{Q \setminus \alpha}, (\{q_0\}, \emptyset), \delta', 2^Q \times \emptyset)$ *with*

$$\delta'((S,O),\sigma) \quad = \quad \begin{cases} \{C_N(S',O') \mid (S',O') \in sat((S,O),\sigma)\} & \textit{if } O' \neq \emptyset \\ \{C_\emptyset(S') \mid S' \in sat(S,\sigma)\} & \textit{otherwise} \end{cases}$$

☐

We have $L(A_{\mathrm{NBT}k}) = L(A_{\mathrm{AWT}k})$.

We improve this construction in three ways. First, we make use of the simulation relation on the AWT to reduce the size of the NBT. Second, we remove *inconsistent states*, and third, we compute the NBT on the fly.

Simulation-Based Optimization We can use the simulation relation that we have computed on $A_{\mathrm{AWT}k}$ to approximate the simulation relation on $A_{\mathrm{NBT}k}$. This is a simple extension of Fritz' result for word automata [Fri03].

Given a direct simulation relation $\preceq_{\mathrm{AWT}}$ for $A_{\mathrm{AWT}k}$, we define the simulation relation $\preceq' \subseteq Q' \times Q'$ on $A_{\mathrm{NBT}k}$ as

$$(S_1, O_1) \preceq' (S_2, O_2) \text{ iff} \forall q_2 \in S_2 \; \exists q_1 \in S_1 : q_1 \preceq_{\mathrm{AWT}} q_2 \wedge (q_2 \in O_2 \rightarrow q_1 \in O_1).$$

Note that $\preceq'$ is a subset of the full (direct) simulation relation on $A_{\mathrm{NBT}k}$ and thus, the following lemma holds.

Lemma 16. *$(S_1, O_1) \preceq' (S_2, O_2)$ implies $L(A^{(S_1,O_1)}) \subseteq L(A^{(S_2,O_2)})$.* □

In particular, for a state $(S, O) \in Q'$, if $q, q' \in S$, $q \preceq_{\mathrm{AWT}} q'$, and $q' \in O \rightarrow q \in O$, then $(S, O) \simeq (S \setminus \{q'\}, O \setminus \{q'\})$. Thus, by Theorem 4, we can remove q' from such sets. Likewise, if $A_{\mathrm{NBT}k}$ contains two simulation equivalent states (S, O) and (S', O') we keep only one (preferring the one with smaller cardinality). Finally, we can use Theorem 6 to remove states that have a simulating sibling.

Removing Inconsistent States In [KV05], it is shown that it is not necessary to include states (S, O) such that (q, i) and $(q, j) \in S$ with $i \neq j$. This implies that we can use the following optimization.

Theorem 17. *Let $A'_{NBTk} = (\Sigma, D, Q'', (\{q_0\}, \emptyset), \delta'', 2^Q \times \emptyset)$ be as in Definition 15, with $Q'' = Q \setminus \{(S, O) \mid \exists (q, i), (q, j) \in S : i \neq j\}$. The transition relation δ'' is obtained from δ' by replacing, for all $C \in \delta'(q, \sigma)$ and all $(S, O) \in C$, state (S, O) by (S', O') where S' is obtained from S by removing all states (q, j) with j not minimal and O' is obtained from O by replacing $(q, j) \in O$ by (q, j') if $(q, j) \notin S'$ and $(q, j) \in S'$.*

We have $L(A'_{NBTk}) = L(A_{NBT})$. □

This is an important theorem as it reduces the number of states in the NBT to $(k+1)^{2n}$ instead of 2^{nk}, where n is the number of states in A_{UCT}.

On-the-Fly Computation Suppose $A_{\mathrm{NBT}k} = (\Sigma, D, Q, q_0, \delta, \alpha)$. Instead of building $A_{\mathrm{NBT}k}$ in full, we construct an NBT $A'_{\mathrm{NBT}}[k] = (\Sigma, D, Q', q_0, \delta', \alpha \cap Q')$ such that $q_0 \in Q' \subseteq Q$ and for $q \in Q'$, either $\delta'(q, \sigma) = \delta(q, \sigma)$ for all σ or $\delta'(q, \sigma) = \emptyset$ for all σ. Thus, $L(A'_{\mathrm{NBT}k}) \subseteq L(A_{\mathrm{NBT}k})$. If $L(A'_{\mathrm{NBT}k}) \neq \emptyset$, the witness of nonemptiness of $L(A'_{\mathrm{NBT}k})$ is a witness of nonemptiness of $L(A_{\mathrm{NBT}k})$. Otherwise, we select a state $q \in Q'$ with $\delta'(q, \sigma) = \emptyset$ and *expand* it, setting $\delta'(q, \sigma) = \delta(q, \sigma)$, introducing the necessary states to Q'.

Our current heuristic expands states in a breadth first manner, which is quite effective. It may be beneficial to expand certain state first, say states with a low cardinality or with high ranks.

Moore Machine

We use the game defined in Section 5.2 to compute language emptiness on the $A_{\mathrm{NBT}k}$. Since $A_{\mathrm{NBT}k}$ is nondeterministic, all states in the winning region have a nonempty language. If the initial state is in the winning region, the language of $A_{\mathrm{NBT}k}$ is not empty and we extract a witness.

Since $A_{\mathrm{NBT}k}$ is a subset of $A_{\mathrm{NBT}k+1}$, we can reuse all results obtained when computing language emptiness on $A_{\mathrm{NBT}k}$ to compute language emptiness on $A_{\mathrm{NBT}k+1}$.

Moreover, it follows from Miyano and Hayashi's construction that if $L(A^{(S,O)}) \neq \emptyset$ and $S \subseteq S'$, then $L(A^{(S',O')}) \neq \emptyset$. We may use this fact to further speed up the computation of language emptiness, and especially to reuse information obtained computing language emptiness on $A_{\mathrm{NBT}k}$ for larger k.

A witness for nonemptiness corresponds to a winning *attractor strategy* [Tho95]. The winning strategy follows the μ-iterations of the final ν-computation of $W_B(\alpha)$: From a state $q \notin \alpha$ we go to a state q' from which the protagonist can force a shorter path to an accepting state. In an accepting state we move back to an arbitrary state in the winning region.

If a strategy exists, it corresponds to a complete Σ-labeled D-tree and thus to a Moore machine M. The states of M are the states of $A_{\mathrm{NBT}k}$ that are reachable when the strategy is followed, and the edges are given by the strategy.

To minimize the strategy, we compute the simulation relation and apply Theorem 4, which is equivalent to using the classical FSM minimization algorithm [HU79]. Thus, the optimized strategy is guaranteed to be minimal with respect to its given I/O language. The output of our tool is a state machine described in VERILOG that implements this strategy.

6 References

[AHKV98] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *Proc. 9th Conferance on Concurrency Theory*, pages 163–178, Nice, September 1998. Springer-Verlag. LNCS 1466.

[ATW05] C. Schulte Althoffa, W. Thomas, and N. Wallmeier. Observations on determinization of buchi automata. In *International Conference on the Implementation and Application of Automata*, 2005.

[B+96] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eighth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, 1996. LNCS 1102.

[BDBF+05] Shoham Ben-David, Roderick Bloem, Dana Fisman, Andreas Griesmayer, Ingo Pill, and Sitvanit Ruah. Automata construction algorithms optimized for PSL, 2005. Prosyd D.3.2/4.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[Chu62] A. Church. Logic, arithmetic and automata. In *Proceedings International Mathematical Congress*, 1962.

[DOT] The DOT Language. http://graphviz.org/doc/info/lang.html.

[Fri03] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In O. H. Ibarra and Z. Dang, editors, *Conference on Implementation and Application of Automata (CIAA'03)*, pages 35–48, 2003. LNCS 2759.

[FW02] C. Fritz and T. Wilke. State space reductions for alternating Büchi automata. In *Foundations of Software Technology and Theoretical Computer Science*, pages 157–168, Kanpur, India, December 2002. Springer-Verlag. LNCS 2556.

[GBS02] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In E. Brinksma and K. G. Larsen, editors, *Fourteenth Conference on Computer Aided Verification (CAV'02)*, pages 610–623. Springer-Verlag, Berlin, July 2002. LNCS 2404.

[GKSV03] S. Gurumurthy, O. Kupferman, F. Somenzi, and M. Y. Vardi. On complementing nondeterministic Büchi automata. In *Correct Hardware Design and Verification Methods (CHARME'03)*, pages 96–110, Berlin, October 2003. Springer-Verlag. LNCS 2860.

[GVZ] Graphviz - Graph Visualization Software. http://graphviz.org/.

[Har05] A. Harding. *Symbolic Strategy Synthesis For Games With LTL Winning Conditions*. PhD thesis, University of Birmingham, 2005.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[JB06a] B. Jobstmann and R. Bloem. Game-based and simulation-based improvements for LTL synthesis. In *Third Workshop on Games in Design and Verification*, 2006. To Appear.

[JB06b] B. Jobstmann and Roderick Bloem. Optimizations for LTL synthesis. In *6th Conferences on Formal Methods in Computer Aided Design (FMCAD '06)*, 2006. To Appear.

[KV05] O. Kupferman and M. Vardi. Safraless decision procedures. In *Symposium on Foundations of Computer Science (FOCS'05)*, pages 531–542, 2005.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems *Specification**. Springer-Verlag, 1991.

[Pit06] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *21st Symposium on Logic in Computer Science (LICS'06)*, 2006. To appear.

[Pnu77] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, 1977.

[PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, pages 364–380, 2006.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. Symposium on Principles of Programming Languages (POPL '89)*, pages 179–190, 1989.

[PRL] Perl. http://www.perl.com/ or http://www.perl.org/.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, 1992.

[Saf88] S. Safra. On the complexity of ω-automata. In *Symposium on Foundations of Computer Science*, pages 319–327, October 1988.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 248–263. Springer-Verlag, Berlin, July 2000. LNCS 1855.

[Tho95] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science*, pages 1–13. Springer-Verlag, 1995. LNCS 900.

[Var05] M. Vardi. A game-theoretic approach to automated program generation. Presentation at IFIP Working Group 2.11 Second Meeting. Available from http://www.cs.rice.edu/ taha/wg2.11/m-2/, 2005.

[VIS] URL: http://vlsi.colorado.edu/∼vis.

[WHT03] N. Wallmeier, P. Hütten, and W. Thomas. Symbolic synthesis of finite-state controllers for request-response specifications. In *Proceedings of the International Conference on the Implementation and Application of Automata*. Springer-Verlag, 2003.

A Syntax Rule Summary

A.1 Syntax of the Specification File

```
SPECFILE    ::= FORMULALIST
FORMULALIST ::= FORMULA POSTFIX |
                FORMULA POSTFIX FORMULALIST
                PREFIX FORMULA POSTFIX |
                PREFIX FORMULA POSTFIX FORMULALIST
PREFIX      ::= assert | assume
POSTFIX     ::= ;NEWLINE
NEWLINE     ::= \n
```

LTL Flavor

```
FORMULA     ::= TERM { BINARYOP TERM }
TERM        ::= ATOM | (FORMULA) |
                UNARYOP (FORMULA) | TEMPORALOP (FORMULA)
BINARYOP    ::= * | + | ˆ | -> | <-> | U | R | V
UNARYOP     ::= !
TEMPORALOP  ::=  G|F|X
ATOM        ::= VARIABLE = VALUE
VARIABLE    ::= \w+
VALUE       ::= 0 | 1
```

PSL Flavor

```
FORMULA     ::= TERM { BINARYOP TERM }
TERM        ::= ATOM | (FORMULA) |
                UNARYOP (FORMULA) | TEMPORALOP (FORMULA)
BINARYOP    ::= & | && | ’|’ | ’||’ | -> | <-> | until! | release!
UNARYOP     ::= !
TEMPORALOP  ::=  always|eventually!|NEXT
NEXT        ::= next | next_e[COUNT] | next_a[COUNT]
COUNT       ::= NUM:NUM | NUM
ATOM        ::= VARIABLE = VALUE | VARIABLE
```

```
VARIABLE    ::= \w+
VALUE       ::= 0 | 1
NUM         ::= \d+
```

A.2 Syntax of the Partition File

```
PARTFILE   ::= PARTITION
PARTITION  ::= INPUTS NEWLINE OUTPUTS NEWLINE
INPUTS     ::= .inputs SIGNALLIST
OUTPUTS    ::= .outputs SIGNALLIST
SIGNALLIST ::= SIGNAL | SIGNAL SIGNALLIST
SIGNAL     ::= \w+
```

A.3 Syntax of the generated DOT Files

```
GRAPH          ::= digraph NAME { HEADER BODY }
NAME           ::= "\w+"
HEADER         ::= OPTIONLIST
OPTIONLIST     ::= OPTION | OPTION OPTIONLIST
OPTION         ::= KEY = VALUE;
KEY            ::= \w+
VALUE          ::= \w+ | \d+ | ".+"
BODY           ::= NODEEDGELIST
NODEEDGELIST   ::= NODE | EDGE | NODE NODEEDGELIST | EDGE NODEEDGELIST
NODE           ::= NAME [NODEOPTIONLIST];
EDGE           ::= NAME -> NAME [NODEOPTIONLIST];
NODEOPTIONLIST ::= NODEOPTION | NODEOPTION, NODEOPTIONLIST
NODEOPTION     ::= KEY = VALUE
```

See [DOT] for the completed definition of the DOT language.

A.4 Syntax of the Automata Files

```
AUTOMATON    ::= STATES ARCS FAIR
STATES       ::= States NEWLINE STATEDEFS
```

```
STATEDEFS    ::= STATE NEWLINE | STATE NEWLINE STATEDEFS
NAME         ::= \w+
DESCRIPTION  ::= { LTLFORMULAE }
LABEL        ::= { LTLFORMULAE }
LTLFORMULAE  ::= FORMULA | FORMULA, LTLFORMULAE
ARCS         ::= Arcs NEWLINE ARCSDEFS
ARCSDEFS     ::= ARC NEWLINE | ARC NEWLINE ARCSDEFS
ARCS         ::= [->] NAME -> { ARCLIST }
FAIR         ::= { LISTOFSTATES }
LISTOFSTATES ::= NAME | NAME, LISTOFSTATES
```

Note that the syntax for a valid LTL-formula is shown in Section A.1. The syntax of `STATE` and `ARCLIST` depend on the automaton.

State-labeled Nondeterministic Büchi Word Automaton

```
STATE        ::= NAME: DESCRIPTION label: LABEL
ARCSLIST     ::= LISTOFSTATES
```

Transition-labeled Nondeterministic Büchi Word Automaton

```
STATE        ::= NAME: DESCRIPTION
ARCLIST      ::= ARC | ARC, ARCLIST
ARC          ::= [ LABEL , NAME ]
```

Universal co-Büchi Tree Automaton

```
STATE        ::= NAME: DESCRIPTION
ARCLIST      ::= ARC | ARC, ARCLIST
ARC          ::= [ LABEL , { DIRSTATELIST } ]
DIRSTATELIST ::= DIRSTATE | DIRSTATE, DIRSTATELIST
DIRSTATE     ::= [ LABEL , NAME ]
```