User Manual of Extended NuSMV

Wanwei Liu, Ji Wang and Zhaofei Wang

National Laboratory of Distributed and Parallel Processing – China

Email: {wwliu,wj,zfwang}@nudt.edu.cn

This document is part of the distribution package of the extended NuSMV model checker, available at `http://nlp.nudt.edu.cn/~lww`.

Contents

Chapter 1

Introduction

NuSMV[CCGR99] is a model checking tool originated by CMU and ICT-irst. And It is a re-implementation and extension of CMU SMV [McM93]. The major adaptation arises from Clarke *et al*'s idea of tableau based LTL model checking [CGH94], which directly leads to a BDD [Bry86] based symbolic LTL model checking.

Presently, NuSMVsupports verification of various temporal logics, including: CTL, LTL, PSL (OBE part), and RTCTL.

As indicated by Vardi [Var98, Var01, NV07], temporal logics based on linear structures benefit both from the syntactical and semantical perspective. Therefore, LTL is a much more popular specification language and, its verification problem seems more fundamental.

However, LTL has its own inadequate feature — not all ω-regular properties can be specified by LTL. For example, Wolper [Wol83] pointed out that the *periodic* properties like "p `holds at least in every even moment`" does not have a peer expression in LTL.

To settle this, various temporal logics are presented to enhance LTL's expressive power. Vardi and Wolper [VW94] suggested three kinds of Extended Temporal Logics (ETL, for short), namely ETL_l, ETL_f and ETL_r, respectively. ETLs employ automata as temporal connectives. e.g., ETL_l, ETL_f and ETL_r respectively uses looping, finite and repeating (Büchi, cf. [Büc62]) automata. It is known that all of these three kinds of ETLs are as expressive as full ω-regular expressions.

ETL induces many variants [KPV01] and it is also the basis of temporal logics like ForSpec [AFF$^+$02] and IBM Sugar (the precursor of PSL). Moreover, LTL can be viewed as a proper fragment of ETL.

Another useful extension is PSL [Acc04], which uses Sequential Extended Regular Expressions (SEREs, for short) as additional formula constructs. And this logic involves both linear features (i.e., FL formulas) and branching features (OBE formulas, which are roughly CTL formulas). PSL has been accepted as an industrial standard (IEEE-1850). Liu *et al* proposed a variant of PSL, namely APSL [LWCM08]. The linear part of APSL, namely AFL, replaces SERE constructs with finite automata. In addition, AFL also employs some operators (such as **`abort`**, **`trigger`**) which are frequently used in the industrial hardware design languages (e.g. VHDL).

As an exercise, we extended Clarke's idea to that of ETL_f. We adapted NuSMVand made it support symbolic model checking of ETL_f. With the extended version of NuSMV, users may customize temporal connectives other than the built-in operators in LTL. Version 1.0 of the extended NuSMVallows users verifying ETL specifications,

and Version 1.1 further supports AFL verifications.

This manual is a supplementary material of NuSMVuser manual. Users are strongly recommended to refer to [CCJ+07] for basic notions and infrastructures in NuSMV. If you don't know how to write an SMV module, or don't know how to verify LTL/CTL properties with NuSMV, please read [CCJ+07] first. This manual puts emphasis on the newly introduce features in the extended version. In order not to annoy the readers, we try to make notations and symbols coherent with the original manual.

The extended NuSMVis also an open source software, and it can be freely downloaded from `http://nlp.nudt.edu.cn/~lww/enusmv`. Its compilation and installation are exactly the same as NuSMV 2.4.

This is a very short user manual, and the rest is organized as follows. Chapter 2 revisits some basic notions of ETL, and then introduces the extended language features used in defining ETL specifications. Chapter 3 briefly defines the logic of AFL, and then illustrates how to define AFL specifications in extended NuSMV. Chapter 4 gives some examples w.r.t. the newly introduced features. Then Chapter 5 illustrates how to perform the ETL/AFL verification with extended NuSMV.

Chapter 2

Specifying ETL Properties in Extended NuSMV

2.1 Extended Temporal Logic

It is Wolper [Wol83] who firstly extended LTL's expressive power by adding ω-grammars as additional formula constructs. Later, Vardi and Wolper [VW94] employed automata as temporal connectives, and they named this family of temporal logics ETLs.

Various ETLs can be defined depending on the type of automata used in the logic. In this document, main focus is put on ETL$_f$ — i.e., ETL with automata on finite words as temporal operators.

Why choose this kind of extend temporal logic? First of all, it is well known that all kinds of ETLs are preciously the same in expressiveness. Secondly, in comparison to other types of automata, automata on finite words is much more familiar to most users. Last but not least, ETL$_f$ has a natural correspondence to some of the most important specification languages. For example, the logic PSL has become an industrial standard specification language. Its core part, namely LTL_WR [BFH05], can be viewed as a proper fragment of ETL$_f$. i.e., each LTL_WR formula can be equivalently translated to an ETL$_f$ formula with the same length.

In the sequel, without explicit declaration, the word "ETL" specially refers to ETL$_f$.

2.1.1 Automata on Finite Words

An `automaton` is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q, F \rangle$, where Σ is an `alphabet`; Q is a finite set of `states`; $\delta : Q \times \Sigma \to 2^Q$ is a `transition function`; $q \in Q$, is an `initial state`; and $F \subseteq Q$, is the set of `final states`.

Given a word $w = a_0 a_1 \dots a_n$. A `run` of w over $\mathcal{A}$ is sequence $q_0 q_1 \dots q_{n+1} \in Q^*$ where q_0 is the initial state, $q_{n+1} \in F$ and $q_{i+1} \in \delta(q_i, a_i)$ for each $0 \leq i \leq n$. w can be `recognized` by $\mathcal{A}$, if there is a run of w over $\mathcal{A}$. The class of finite words recognized by $\mathcal{A}$ is denoted by $L(\mathcal{A})$. In addition, we denote by $PreL(\mathcal{A})$ the `prefixes` of $L(\mathcal{A})$. That is, $PreL(\mathcal{A}) = \{w \mid \exists w', \text{ s.t. } ww' \in L(\mathcal{A})\}$.

2.1.2 ETL, the Syntactical and Semantical Definition

Fix a set AP of atomic propositions, the formal definition of ETL formulas is inductively given as follows.

- Each atomic proposition $p \in AP$ is an ETL formula.
- If φ is an ETL formula, then $\neg\varphi$ is an ETL formula.
- If φ_1 and φ_2 are ETL formulas, then $\varphi_1 \wedge \varphi_2$ and $\varphi_1 \vee \varphi_2$ are ETL formulas.
- If $\mathcal{A}$ is an automaton whose alphabet is $\{a_1, \ldots, a_n\}$, and $\varphi_1, \ldots, \varphi_n$ are ETL formulas, then $\mathcal{A}(\varphi_1, \ldots, \varphi_n)$ is an ETL formula.

Semantics of ETL formulas are defined w.r.t. linear `models`, each model π is an ω word in $(2^{AP})^\omega$. For the model π, let $\pi(i)$ be its i-th letter and let $\pi[i]$ be its suffix starting from the i-th letter. Clearly, $\pi[0] = \pi$. Inductively, .

- $\pi \models p$ if and only if $p \in \pi(0)$.
- $\pi \models \neg\varphi$ if and only if $\pi \not\models \varphi$.
- $\pi \models \varphi_1 \vee \varphi_2$ if and only if either $\pi \models \varphi_2$ or $\pi \models \varphi_2$; $\pi \models \varphi_1 \wedge \varphi_2$ if and only if both $\pi \models \varphi_2$ and $\pi \models \varphi_2$.
- Suppose that $\mathcal{A}$ is an automaton whose alphabet is $\{a_1, \ldots, a_n\}$. Then, $\pi \models \mathcal{A}(\varphi_1, \ldots, \varphi_n)$ if and only if there is some $w \in L(\mathcal{A})$, such that for each $i < |w|$, if the i-th letter of w is a_k, then $\pi[i] \models \varphi_k$.

Note. An n-letter automaton is viewed as an n-ary temporal connective, and each letter acts as a place holder. The order among letters is extremely important, hence, it is more proper to view the alphabet as a vector, other than a set.

2.2 Customizing Temporal Connectives with Automata

The extended NuSMV allows users to define their own temporal connectives by writing finite automata.

An automaton definition consists of the declaration of alphabet, state set and transition relations. The following shows how these features described by the extended NuSMVlanguages.

2.2.1 Alphabet Declaration

The alphabet of an automaton is a non-empty list of `letter_names`, each `letter_name` can be an arbitrary meaningful `identifier`. The definition of legal `identifier` can be found in [CCJ+07].

```
alphabet_decl ::
     letter_list

letter_list ::
      letter_name
    | letter_list,letter_name

letter_name :: identifier
```

2.2.2 State Set Declaration

For declaring the state set, one also need a list of `identifiers`. Moreover, it is also necessary to identify the initial and final states.

```
state_set_decl :: STATES: state_list

state_list :: state_decl
    | state_list, state_decl

state_decl :: state_name
    | >state_name
    | state_name<

state_name :: identifier
```

State declaration starting with "$>$" indicates an initial state; and state declaration ending with "$<$" indicates a final state. Notice that the symbol of "$>$" or "$<$" is not a part of state name.

In the `state_set` declaration, a unique initial state is required (otherwise, NuSMVreports an error), and at least one final state is required (otherwise, NuSMVproduces a warning).

2.2.3 Transition Relation Declaration

The core part of connective definition is the declaration of `transition_relation`. In NuSMV, transitions are defined using the following grammar.

```
transitions_decl :: transition_decl;
    | transitions_decl; transition_decl;

transition_decl :: TRANSITIONS (state_name) transition_body

transition_body :: case transition_def_list esac

transition_def_list :: transition_def;
    | transition_def_list; transition_def;

transition_def :: letter_name : state_name |
    letter_name : {state_name_list}

state_name_list :: state_name | state_name_list, state_name
```

Notice that for each individual state, at most one `transition_decl` can be defined w.r.t. it.

2.2.4 Automaton Connective Declaration

An automaton connective declaration encapsulates the alphabet, state set and transition relation. Automata connectives are used to "glue" other ETL formulas.

```
automaton_connective_decl ::
        CONNECTIVE  connective_name (alphabet_decl)
        connective_body
```

```
connective_name :: identifier

connective_body :: state_set_decl    transitions_decl
```

For example, consider the automaton *demo_conn* depicted in Figure 2.1, its NuSMVdescription can be written as follows.

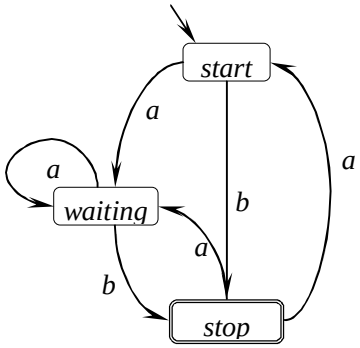

Figure 2.1: An example automaton connective

```
CONNECTIVE demo_conn(a,b)
STATES:
    >start, waiting, stop<
TRANSITIONS (start)
case
    a: waiting;
    b: stop;
esac;
TRANSITIONS (waiting)
case
    a: {waiting,stop};
    b: stop;
esac;
TRANSITIONS (stop)
case
    a: {start, waiting};
esac;
```

2.3 ETL Specifications

ETL specifications are introduced by the keyword **ETLSPEC** . The syntax of this specification is:

```
etl_specification :: ETLSPEC etl_expr [; ]
```

The ETL formulas recognized by NuSMVare as follows:

```
etl_expr ::
```

```
    simple_expr                                 --  a simple Boolean expression
    | ( etl_expr)
    | ! etl_expr                                --  logical not
    | etl_expr & etl_expr                       --  logical and
    | etl_expr | etl_expr                       --  logical or
    | etl_expr xor etl_expr                     --  logical exclusive or
    | etl_expr -> etl_expr                      --  logical implies
    | etl_expr <-> etl_expr                     --  logical equivalence
    | X etl_expr                                --  temporal next
    | automaton_name( etl_expr_list)            --  automaton invocation

etl_expr_list :: etl_expr
    | etl_expr_list, etl_expr
```

Notice For ETL formulas of automaton invocations, the number of operators (i.e., the length of `etl_expr_list`) must match the size of the automaton connective's alphabet. The operator **`X`** is preserved in ETL specification.

All LTL specifications can be equivalently translated to ETL specifications. For example, consider the following connective *until*:

```
CONNECTIVE until(a,b)
STATES: >st_1, st_2<
TRANSITIONS (st_1)
case
    a: st_1;
    b: st_2;
esac;
```

Then, the ETL specification `until(prop_1,prop_2)` is exactly the same as the LTL specification `prop_1` **`U`** `prop_2`.

Chapter 3

Specifying AFL Properties in Extended NuSMV

3.1 What Is AFL?

AFL (Automatarized Fundamental Logic) is the linear part of of APSL [LWCM08]. In comparison to standard FL, additional formula constructs is not SEREs, but finite automata.

There are two major reasons for why replacing SEREs in the formulas with finite automata:

- Firstly, though SEREs are much convenient to write, they sometimes lack of intuitiveness. For, the transition structures of SEREs/REs are not as clear as that of finite automata. The combining use of these operators makes the decidability of the decidability of SERE extremely hard — e.g., transforming SEREs like $(r_1 \& r_2) : r_3$ into an equivalent automaton is not a trivial task. This makes the symbolic verification of standard FL formulas is hard to implement.
- Secondly, an SERE expressive may involve some newly introduced operators. Adding these operators may make the expressing flexible, however, it would not change the expressive power. Hence, change SEREs with finite automata will not change the essence of the logic.

3.1.1 Syntax

Fix a set AP of atomic propositions, **Boolean formulas** are formulas built up from AP and Boolean connectives. Formally,

- Each $p \in AP$ is a Boolean formula.
- If φ, φ_1, φ_2 are Boolean formulas, then both $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ are Boolean formulas.

Subsequently, we inductively define AFL formulas.

- Every Boolean formula is an AFL formula.
- If φ, φ_1, φ_2 are AFL formulas, then both $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ are AFL formulas.
- If φ is an AFL, then $\mathsf{X}\varphi$ is an AFL formula.
- If φ_1 and φ_2 are AFL formulas, then $[\varphi_1 \mathsf{U} \varphi_2]$ is an AFL formula.
- If $\mathcal{A}$ is an automaton with alphabet $\{a_1, \ldots, a_n\}$, $\varphi_1, \ldots, \varphi_{n+1}$ are Boolean formulas, then $[\mathcal{A}(\varphi_1, \ldots, \varphi_n)\ \mathsf{abort}\ \varphi_{n+1}]$ is an AFL formula.
- If $\mathcal{A}$ is an automaton with alphabet $\{a_1, \ldots, a_n\}$, $\varphi_1, \ldots, \varphi_n$ are Boolean formulas, and ψ is an AFL formula, then $[\mathcal{A}(\varphi_1, \ldots, \varphi_n)\ \mathsf{T}\ \psi]$ is an AFL formula.

3.1.2 Semantics

Unlike standard FL, semantics of AFL formulas are given only w.r.t. infinite linear models. Inductively, given a linear model $\pi \in (2^{AP})^{\omega}$:

- For each $p \in AP$, $\pi \models p$ if and only if $p \in \pi(0)$.
- $\pi \models \neg\varphi$ if and only if $\pi \not\models \varphi$.
- $\pi \models \varphi_1 \wedge \varphi_2$ if and only if both $\pi \models \varphi_1$ and $\pi \models \varphi_2$ hold.
- $\pi \models \mathsf{X}\varphi$ if and only if $\pi[1] \models \varphi$.
- $\pi \models [\varphi_1 \mathsf{U} \varphi_2]$ if and only if $\exists i \geq 0$, such that $\pi[i] \models \varphi_2$ and for each $0 \leq j < i$, $\pi[j] \models \varphi_1$.
- Suppose that the alphabet of $\mathcal{A}$ is $\{a_1, \ldots, a_n\}$, then $\pi \models [\mathcal{A}(\varphi_1, \ldots, \varphi_n)\ \mathsf{abort}\ \varphi_{n+1}]$ (resp. $[\mathcal{A}(\varphi_1, \ldots, \varphi_n)\ \mathsf{T}\ \varphi_{n+1}]$) if and only if $\exists w \in PreL(\mathcal{A})$ (resp. $\exists w \in L(\mathcal{A})$), s.t. for each $i < |w|$, if the i-th letter of w is a_k, then $\pi[i] \models \varphi_k$; moreover, we require $\pi[|w|] \models \varphi_{n+1}$.

Note. There's a slight different between abort and T: the former must be immediately followed by a pure Boolean formula, whereas any AFL may appear after the latter.

3.2 AFL Specifications

AFL specifications are introduced by the keyword **AFLSPEC** .

```
afl_specification :: AFLSPEC afl_expr [; ]
```

AFL formulas that can be recognized by NuSMVare as follows:

```
afl_expr ::
    simple_expr                              --  a simple Boolean expression
    | ( afl_expr)
    | ! afl_expr                             --  logical not
    | afl_expr & afl_expr                    --  logical and
    | afl_expr | afl_expr                    --  logical or
    | afl_expr xor afl_expr                  --  logical exclusive or
    | afl_expr -> afl_expr                   --  logical implies
    | afl_expr <-> etl_expr                  --  logical equivalence
    | X afl_expr                             --  temporal next
    | afl_expr U afl_expr                    --  temporal until
    | afl_expr V afl_expr                    --  temporal releases
    | atom_etl_expr abort simple_expr        --  temporal abort
    | atom_etl_expr monitor simple_expr      --  temporal monitor
    | atom_etl_expr T afl_expr               --  temporal trigger
    | atom_etl_expr |-> afl_expr             --  temporal leadsto

atom_etl_expr ::
    automaton_name( simple_expr_list)        --  automaton invocation

simple_expr_list ::                          --  list of Boolean expressions
    simple_expr
    | simple_expr, simple_expr_list
```

Notice that we here introduce two new keywords **abort** and **monitor** ; and the operator **|->**. The operators of **V**, **monitor** and **|->** are respectively the negative dual of **U**, reserved-abort and **T**. In detail, the specifications of

```
AFLSPEC prop_1 V prop_2
AFLSPEC expr monitor prop
AFLSPEC expr |-> prop
```

are respectively the same as

```
AFLSPEC ! (! prop_1 U ! prop_2)
AFLSPEC ! (expr abort ! prop)
AFLSPEC ! (expr T ! prop).
```

Using these derived operators, one may put negations inward into parenthesis and hence acquire the positive normal form of AFL specifications.

Notice For AFL specifications, one should pay attention to the following issues when defining/using automaton connectives.

1. The way to define automaton connectives is the same as in ETL (cf. Section 2.2.4).
2. Unlike ETL specifications, parameters of automaton connectives can only be pure Boolean expressions.
3. In AFL specifications, an automaton formula must be immediately followed by one of the operators of **abort**, **monitor**, **T**, or **|->**.
4. More importantly, when giving names to a user-defined automaton connective, please DO NOT chose strings starting with _sys_.

Chapter 4

Examples

In this chapter, we demonstrate how to use the extended version of NuSMVto verify ETL specifications by giving some simple examples.

4.1 A Synchronous Example

This first example is a synchronous model: *octad counter*. It consists of three instances of *cells*, and each cell is a binary counter (mod 2) having an input signal "*carry_in*" and an output signal, "*carry_out*".

Counter cells can be modeled by the following SMV code.

```
MODULE cell (carry_in)
VAR
    pre_value : boolean;
    value : boolean;
ASSIGN
    init (pre_value):=0;
    init (value):=0;
    next (pre_value):=value;
    next (value) := (value+carray_in) mod 2;
DEFINE
    carry_out : = pre_value & carry_in;
```

Subsequently, we create three instances of `cell`, namely `bit_0`, `bit_1`, `bit_2`, respectively.

`bit_0.carry_in` is always set to 1, `bit_1.carry_in` is `bit_0.carry_out` and `bit_2.carry_in` is `bit_1.carry_out`. This can be described as Figure 4.1.

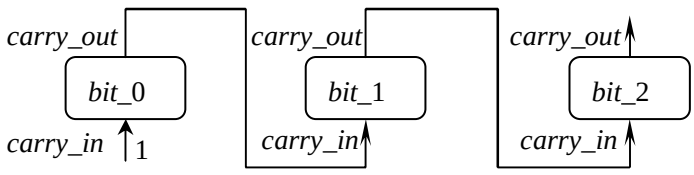

Figure 4.1: The Octad Counter

Thus, the system is declared as follows.

```
MODULE main
VAR
    bit_0 : cell(1);
    bit_1 : cell(bit_0.carry_out);
    bit_2 : cell(bit_1.carry_out);
```

Subsequently, we need to verify the property "`bit_2 carries out infinitely often`" (equals to the LTL Specification **G F** `bit_2.carry_out`). To fulfill this, we borrow the previously defined automaton connective *until* (cf. Section 2.3, Page 9) and write the specification as follows.

```
ETLSPEC ! until(TRUE ,! until(TRUE ,bit_2.carry_out))
```

4.2 An Asynchronous Example

In this section, we give an asynchronous example — *inverter_ring*. An inverter is nothing else but a NOT gate, and an inverter ring is a circle of serial inverters.

An inverter can be modeled as follows.

```
MODULE inverter(input)
VAR
    output : boolean;
ASSIGN
    init (output) := 0;
    next (output) := ! input;
FAIRNESS
    running
```

An inverter ring consists of n instances of inverters `gate_0`, ..., `gate_n-1` and `gate_i.input` is set to `gate_j.output` if $j = (i+1) \bmod n$. Thus, the system can be modeled as follows.

```
MODULE main
VAR
    gate_0 : process inverter(gate_n-1.output);
    gate_1 : process inverter(gate_0.output);
    ......
   gate_n-1: process inverter(gate_n-2.output);
```

Note. the keyword **process** indicates that each instance executes in an asynchronous way, and **FAIRNESS running** guarantees that each instance is scheduled infinitely often.

Here an interesting property is that "`This ring will not lead to a stable status`" — i.e., each inverter outputs 0 and 1 infinitely often. The ETL specification for verifying this property is:

```
ETLSPEC !until(TRUE ,!until(TRUE ,bit_0.output=0))
                & !until(TRUE ,!until(TRUE ,bit_0.output=1))
```

The equivalent LTL specification is:

```
LTLSPEC G F bit_0.output=0 & G F bit_0.output=1
```

One may exam that this property holds when n is an odd number, and it is violated when n is an even number.

4.3 Example of Non-Star-Free Properties

In this section, we would verify some properties that cannot be expressed in LTL/CTL.

Still consider the example of *Octad counter*, the following properties are obviously held, however, cannot be specified by LTL/CTL.

- "`bit_0.carry_out is evaluated to 1 at moments of 2, 4, 6,...`".
- "`There is a quad moment having bit_1.output=1`".

Consider the following automaton connective *eventually_2* and *eventually_4*. It is clear that the above two properties can be respectively sepcified as

```
ETLSPEC X X ! eventually_2(TRUE,!bit_0.carry_out)
```

and

```
ETLSPEC eventually_4(TRUE,bit_1.carry_out)
```

```
-------------- eventually_2 ---------------
CONNECTIVE eventually_2(a,b)
STATES:
    >st_1, st_2, fin<
TRANSITIONS (st_1)
case
    a : st_2;
    b : fin;
esac;
TRANSITIONS (st_2)
case
    a : st_1;
esac;
-------------- eventually_4 ---------------
CONNECTIVE eventually_4(a,b)
STATES:
    >st_1, st_2, st_3,st_4, fin<
TRANSITIONS (st_1)
case
    a : st_2;
    b : fin;
esac;
TRANSITIONS (st_2)
case
    a : st_3;
esac;
TRANSITIONS (st_3)
case
    a : st_4;
esac;
TRANSITIONS (st_4)
case
    a : st_1;
esac;
-----------------------------------------
```

Analogously, one can define the temporal connectives of *eventually_k* for every natural number k.

Chapter 5

Performing ETL/AFL Verification

NuSMV 2.4 supports two verification modes: the *batch mode* and the *interactive mode*. We apologize to the users that the extended version **only supports the batch mode** when doing ETL/AFL verification.

The major difficulty lies from the inconvenience of defining automaton connective in a single command line. This might be settled by providing additional command line `read_connective` (like `read_model`) to read a connective from a file, yet, not implemented presently.

To execute the batch verification, just runs NuSMVwithout `-int` option. That is, input the following command in the system prompt window.

```
system_prompt> ./NuSMV [command line options] input_file <RET>
```

Usage of command line options can be found in [CCJ+07].

Bibliography

[Acc04] Accellera. Accellera property languages reference manual. http://www.eda.org/vfv/docs/PSL-v1.1.pdf, June 2004.

[AFF+02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The forspec temporal logic: A new temporal property-specification language. In *8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2002)*, volume 2280 of *Lecture Notes in Computer Science*, pages 296–311. Springer, 2002.

[BFH05] D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. Technical Report MCS05-04, IBM Haifa Research Lab, May 2005.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congr. Method and Philosophy of Science 1960*, pages 1–12, Palo Alto, CA, USA, 1962. Stanford University Press.

[CCGR99] A. Cimatti, E. Clarke, F. Giunchiglia1, and M. Roveri. Nusmv: A new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *Proc. 11th Computer Aided Verfication (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer-Verlag, 1999.

[CCJ+07] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. Nusmv 2.4 user manual. http://nusmv.fbk.eu/NuSMV/userman/v24/nusmv.pdf, Apr. 2007.

[CGH94] E.M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV'94*, volume 818, pages 415–427, Standford, California, USA, 1994. Springer-Verlag.

[KPV01] O. Kupferman, N. Piterman, and M. Y. Vardi. Extended temporal logic revisited. In *Proc. 12th International Conference on Concurrency Theory*, volume 2154 of *Lecture Notes in Computer Science*, pages 519–535, Denmark, 2001. Springer.

[LWCM08] W. Liu, J. Wang, H. Chen, and X. Ma. Symbolic model checking apsl. In J. Davies and X. Li, editors, *Proc. 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 39–46. IEEE Soc, June 2008.

[McM93] K. L. McMillan. *Symbolic Model Checking, An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Kluwer Academic Publishers, 1993.

[NV07] S. Nain and M.Y. Vardi. Branching vs. linear time: Semantical perspective. In *Proc. 5th Automated Technology on Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2007.

[Var98] M. Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In *Logic in Computer Science*, pages 394–405. IEEE Society, 1998.

[Var01] M. Y. Vardi. Branching vs. linear time: Final showdown. In *TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2001.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

Index