

*FP6-IST-507219*

## **PROSYD:**

*Property-Based System Design*

Instrument: Specific Targeted Research Project

Thematic Priority: Information Society Technologies

### **Manual for port of NUSMV (Deliverable 3.3/4)**

Due date of deliverable: June 30, 2005

Actual submission date: July 19, 2005

Start date of project: January 1, 2004

Duration: Three years

Organisation name of lead contractor for this deliverable: ITC-irst

Revision 1.0

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

## **Notices**

For information, contact Marco Roveri [roveri@irst.itc.it](mailto:roveri@irst.itc.it).

This document is intended to fulfil the contractual obligations of the PROSYD project concerning deliverable 3.3/4 described in contract number 507219.

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

© Copyright PROSYD 2005. All rights reserved.

## Table of Revisions

Version	Date	Description and reason	By	Affected sections
0.1	June 15, 2005	First version	Simone Semprini	All
0.2	June 20, 2005	General review	Marco Roveri	All
0.3	June 28, 2005	New section on technical details	Simone Semprini	Section 4
0.4	July 5, 2005	General review	Alessandro Cimatti	All
0.5	July 9, 2005	General review	Marco Roveri	All
0.6	July 14, 2005	Feedback from consortium	Simone Semprini	All, new section on PSL supported subset 2.2
0.7	July 16, 2005	General review	Marco Roveri	All
0.8	July 18, 2005	Final general review	Marco Roveri	All
1.0	July 19, 2005	Final approval from project coordinator	Marco Roveri for Cindy Eisner	

## Authors

Marco Roveri  
Simone Semprini  
Alessandro Cimatti

## Executive Summary

This document contains the description of the work performed under the effort for Deliverable 3.4/4 for the porting of NuSMV to PSL.

## Purpose

The purpose of this document is to describe the work performed under the effort for Deliverable 3.4/4 aiming at the porting of NuSMV to PSL. The functionalities of the tool and the current status of PSL support are fully described in the NuSMV user manual [13] (also in Appendix A). This document is intended to accompany the user manual.

## Intended Audience

This document is intended for individuals who work with NUSMV and PSL. It is assumed that readers are familiar with the notions and terms related to PSL, and that they are familiar with the functionalities of NUSMV.

## Background

NUSMV was born as an open architecture for model checking; its goal is that of providing users with an academic tool that offers a wide range of verification techniques (e.g. BDD-based, BMC SAT-based, induction-based), and is amenable to be extended based on the users' needs. NUSMV has hundreds of active installations worldwide, some of which in industrial settings, and has been used both to face real life problems and to provide proof of concepts for new research achievements. See <http://nusmv.irst.itc.it> for more information on the NUSMV system and for a list of known projects related to NUSMV.

NUSMV defines its own specification language and was originally defined to provide users with the possibility of applying verification on specification written in the NUSMV version of CTL and LTL, then it was extended to support RCTL too. This was the status of the tool at the beginning of the PROSYD project.

In the framework of the PROSYD project, each partner is committed to updating its tools to PSL. ITC-irst has undertaken this task in Deliverable 3.3/4, and this report describes the results that have been achieved.

# Contents

Table of Revisions .....	iii
Authors .....	iii
Executive Summary .....	iii
Purpose .....	iii
Intended Audience.....	iii
Background .....	iv
Contents .....	v
Table of Figures .....	vi
Glossary .....	vii
1 Introduction .....	1
1.1 Overall results.....	2
2 Adding new functionalities .....	3
2.1 Integrating the IBM PSL parser.....	3
Background:.....	3
Starting point:.....	3
Work performed:.....	3
Results achieved: .....	5
2.2 Defining the PSL subset supported for verification .....	5
Background:.....	5
Starting point:.....	5
Work performed:.....	6
Results achieved: .....	7
2.3 Implementing a translation from a subset of PSL to CTL and LTL ..	7
Background:.....	7
Starting point:.....	8
Work performed:.....	8
Results achieved: .....	8
3 Extending existing functionalities.....	9
3.1 Definition of a new command for verification of PSL specifications	9
Background:.....	9
Starting point:.....	9
Work performed:.....	9
Results achieved: .....	10
4 Technical details.....	13
4.1 Extended NUSMV architecture .....	13
The psl package .....	13
4.2 Extended NUSMV flow .....	14
4.3 Ported some NUSMV examples to PSL.....	16
5 References.....	17
A NUSMV manual .....	19

# Table of Figures

Figure 1 - The NuSMV architecture. .... 15

# Glossary

## **BDD**

A binary decision diagram (BDD) is a data structure that is used to represent a Boolean function. The Boolean function is represented in a BDD as a rooted, directed, acyclic graph where each non-terminating vertex is labeled by a variable and has two directed edges connecting to child nodes. One edge represents a variable's assignment to zero and the other represents an assignment to one. A 1 or 0 labels all terminal vertexes [4].

## **BMC, Bounded Model Checking**

Bounded Model Checking is a verification technique based on the reduction of model checking to the satisfiability of a propositional formula that represents a bounded encoding of the model checking problem [2].

## **CTL, Computational Tree Logic**

Computational Tree Logic is a temporal logic for property specification in formal verification [7].

## **DIMACS**

A standard format for files containing the description of a satisfiability problem.

## **GDL**

General Description Language is the modeling language of the IBM verification tools, RuleBase PE and FoCS. Its primary purpose is to describe the environment for formal verification. However, it is also used in conjunction with PSL to aid specification of a design.

## **LTL, Linear Time Temporal Logic**

Linear Time Temporal Logic is a temporal logic for property specification in formal verification [15].

## **Model Checking**

Model checking is a method for formally verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not [6].

## **NUSMV**

An academic tool that implements an open architecture for model checking, that provides the users with a rich set of verification techniques, and which has hundreds of active installations worldwide. NUSMV is developed by ITC-irst [14].

## **NUSMV language**

The input language of NUSMV [13].

## **Property**

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

## **PSL**

Property Specification Language, the language for specification of designs upon which PROSYD is based [19].

### **RBC**

Reduced Boolean Circuits, are a representation of propositional logic formulae based on direct acyclic graphs that allows to reduce the size of the resulting graphs by exploiting sub-formula sharing.

### **RCTL**

Real Time Computational Tree Logic, is an extension of Computational Tree Logic that allows quantitative reasoning on time [10].

### **Regular expression**

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

### **SERE**

Sequential Extended Regular Expressions are the PSL version of regular expressions [19].

### **Verilog**

One of the two standardized hardware description languages used to specify the structure and behavior of electronic systems in textual format. Developed in the mid-1980s as a proprietary language and acquired by Cadence Design Systems, it became a de facto industry standard. In the mid-1990s Cadence placed it into the public domain and it became a de jure standard promulgated by the Institute of Electric and Electronic Engineers (IEEE). Verilog is also the name of a legacy simulation tool offered by Cadence.

# 1 Introduction

Till the last public release (version 2.2.5), NUSMV did not offer any support to PSL. The goal of this deliverable was to extend the tool to provide its users the possibility of writing their specifications in PSL, and of performing verification on the specifications belonging to a subset of PSL.

The results of this activity are described in detail in the following sections. They are categorized as follows:

- entirely new functionalities (Section 2);
- existing functionalities that have been extended to (a subset of) PSL (Section 3).

The PSL version supported by NUSMV is currently version 1.01, as described in [19] and implemented in the PSL front-end released by IBM as [18], with a NUSMV flavor.

Currently NUSMV allows the users to input specifications based on the full grammar of PSL properties, and provides verification for the specifications that belong to a subset of the language. The subset for which verification is provided covers the PSL OBE (Optional Branching Extension), the subset of PSL which is directly mappable on LTL, and a subset of SEREs (Sequential Extended Regular Expressions) related operators for which a translation into LTL has been defined in [5]; see Section 2.2 for a definition of the subset. The PSL language includes a very large set of features, and each of the PSL-supporting tools that exist today have started by supporting a limited subset, which is continually extended as the demand for PSL support grows. Giving the users the possibility to input full-fledged PSL properties reflects the commitment to gradually extend the support to verification of PSL inside NUSMV to eventually cover the whole language. Future extensions of NUSMV will be based on [5], which defines how to embed SEREs into LTL formulae by means of automata and proper LTL expressions that predicate about the runs of the automata, and on [17], which provides a translation of PSL properties into automata (thus paving the way to an automata-based verification of PSL), and on a research on the extension of BMC techniques [2] to deal with SEREs.

The choice of adopting a NUSMV flavor has been motivated by the following elements:

- the NUSMV language, on which the flavor is based, is an academic standard and having an academic tool that provides PSL in terms of a familiar look and feel, may ease adoption of PSL into academic world;
- the NUSMV flavor is very close to the PSL GDL proprietary flavor;

- there is the possibility to automatically translate specifications written in the Verilog flavor into the NUSMV flavor by using external public domain tools (e.g. the Confluence tool suite [8]).

For these reasons, the definition of a NUSMV flavor, besides having positive impact on the effort spent for this deliverable, has not limited the usability of the achieved results.

The final product of the porting activity – NUSMV extended to support PSL – is fully described in the NUSMV user manual [13] and Appendix A. This document is intended to accompany the user manual. It describes the detailed tasks comprised by the porting activity, and clarifies the differences between the pre-existing status of the NUSMV tool and its status after the completion of this deliverable.

---

## 1.1 Overall results

A new version of NUSMV which provides support for PSL has been produced and released for public download at the NUSMV web site <http://nusmv.iirst.itc.it> and on the PROSYD web-site <http://www.prosyd.org>.

The PSL subset currently supported for verification by NUSMV is defined in Section 2.2.

# 2 Adding new functionalities

In this section we describe the tasks carried out to add new functionalities to NUSMV. These tasks comprise an integration of the parser provided by IBM in [18], the definition of a PSL subset supported for verification by NUSMV, and the definition and implementation of a translation from a significant subset of PSL to CTL or LTL.

Substantial syntactic and semantics analyses of the PSL language have been carried out to give the reported design and implementation activities a principled basis.

---

## 2.1 Integrating the IBM PSL parser

### **Background:**

IBM provided a PSL parser for PSL version 1.01 as PROSYD Deliverable 1.2/3 [18]. The IBM parser has been implemented assuming C++ as the underlying development language, and it accepts only the Verilog flavor.

NUSMV has been developed assuming C as the underlying development language.

### **Starting point:**

At the starting point of the porting activity, NUSMV did not provide any support to PSL.

Plans for supporting PSL have started within the PROSYD project, and included the specification and design of an additional parsing module `PSL_parser` to provide the NUSMV standard parser the extensions needed to cover the grammar of PSL properties.

## Work performed:

### Specification and Design of PSL\_parser:

- for the sake of modularity we did not want to directly extend the NUSMV standard parser with support to the PSL properties grammar, rather we preferred to define an auxiliary ad-hoc module to carry out parsing of PSL specifications;
- the addition of ad-hoc module to parse PSL properties implied the definition of a proper interaction protocol between the two parsers that has been defined in terms of a two-actors parsing process: the control of parsing moves from the standard NUSMV parser to the PSL\_parser and vice-versa based on the type of properties that need to be parsed;
- this architecture, besides fostering a better engineering of the tool, allows to easily mix standard NUSMV specifications with PSL specifications, giving the users the chance to refer to a particular language based on their specific needs rather than forcing them to an a priori choice;
- besides designing the parser architecture, we defined the NUSMV flavor starting from the Verilog flavor guided by the GDL specifications.

### Implementation of PSL\_parser:

- the NUSMV system is implemented in C, while the IBM parser is implemented in C++, moreover, the internal structures of the IBM parser are different from the ones of NUSMV standard parser; to ease the integration of the IBM parser, we decided to tailor it to our needs, by rewriting the semantics actions to fit the NUSMV internals and programming language needs;
- this implementation activity covered also the porting of the parser to the NUSMV flavor.

### Integration of PSL\_parser:

- an initial stand-alone version of the PSL\_parser has been integrated into the NUSMV system by modifying the flow of the standard parser to make it able to recognize the presence of a PSL specification and delegate its parsing to the PSL\_parser; the result of the parsing of a PSL property is passed back by the PSL\_parser to the standard NUSMV parser by means of common data structures;
- it has been necessary to extend several packages of the NUSMV system to handle PSL properties, e.g. the hierarchy flattener, the property database etc. (See Section 4.1 for a description of the NUSMV architecture.)

### Testing of the PSL\_parser

- the PSL\_parser has been thoroughly tested against a wide set of PSL properties;
- a random PSL properties generator and a proper testing procedure has been implemented for this purpose.

### Results achieved:

- NUSMV has been extended to accept in input PSL specifications;
- the NUSMV flavor for PSL has been defined;
- a stand-alone parser for the PSL NUSMV flavor has been implemented; this parser can be used as an external module by third parties willing to integrate PSL with NUSMV flavor into their tools;
- a stand-alone random PSL properties generator has been developed to allow for the generation of generic properties or properties that belong to the supported subset;
- a test-suite of more than 1000 PSL properties has been defined and used throughout the whole activity here reported.

---

## 2.2 Defining the PSL subset supported for verification

### Background:

PSL as a whole constitutes a source of interesting problems when it comes to formal semantics and verification (both simulation and static verification). Efforts in the direction of formal verification of full PSL are being carried on by industrial companies and academic research groups, and in most cases share the common feature of focusing on a restricted set of the language to lay the bases for further extensions. The task of identifying which subset to cover as a first step is not trivial since it involves considerations that refer to both pragmatic and scientific issues.

## Starting point:

The ‘Property-by-Example guide: a handbook of PSL/Sugar examples’ [16] and the verification capabilities currently provided by NUSMV (LTL BDD and SAT based model checking and CTL BDD based model checking).

## Work performed:

We analyzed the “Property-by-Example guide: a handbook of PSL/Sugar examples” [16] driven by the verification capabilities currently provided by NUSMV to extract an initial significant subset of PSL to support for verification.

The identified and currently supported subset of PSL can be characterized by applying the following constraints to the full-fledged language as described in [19]:

1. the **within** and **whilenot** families of operators, and the **abort** operator are not supported;
2. top level expressions whose language contains the empty sequence are ruled out; in [5] a predicate has been defined to identify such expressions;
3.  $r$  in  $r[*]$  must be a boolean; stand-alone  $[*]$  is allowed as part of a concatenation, i.e. non at top-level;  $[+]$  is always allowed;
4. arguments of sequence conjunction operators, namely **&** and **&&**, must be star-free;
5. left arguments of suffix implication operators must be star-free; note that in some cases properties built using repeated SEREs as left arguments of suffix implications can be rewritten without using suffix implication, e.g.  $\{[*];p\}|\Rightarrow q$  with  $p$  propositional can be rewritten as **always** ( $p \rightarrow$  **next**  $q$ ), see [16].

Given these constraints, here we report few examples of supported and not supported properties, together with the constraint they relate to:

Supported	Not supported	Constraint
<b>always</b> $\{b[*];a\}$	<b>always</b> $\{b[*]\}$	2
<b>always</b> $\{\{a[*]   b[*]\};\{c;d\}\}$	<b>always</b> $\{a[*]   b[*]\}$	2
<b>always</b> $\{[*];b\}$	<b>always</b> $\{[*]\}$	2
<b>always</b> $\{b;[*]\}$		2
<b>never</b> $\{a;c[*];b\}$	<b>never</b> $\{a;\{c;d\}[*];b\}$	3
<b>never</b> $\{a;[*];b\}$		3
<b>never</b> $\{[+]   \{a;[*];b\}\}$		3
<b>always</b> $\{\{a;c\}   \{a;b;c\}\} \&\& \{d;f\}$	<b>always</b> $\{a;b[*];c\} \&\& \{d;f\}$	4
$\{a;b\} \Rightarrow \{c[*]\}$	$\{a[*];b\} \Rightarrow \{c[*]\}$	5

When not against the constraints above, any nesting of temporal or sequence operators, and any combination of propositional or relational operators are allowed. For example, the following formula belongs to the supported subset:

```

next {[+] |
  {(e[idx]/f) ≤ (g - h)} |
  {((i * -j) ≤ (k - l))} & {((m ↔ n) → (!o[idx]!p))}}
  (((q xor r) & (!s → !t)) until_ ((u * v) != (w * x[idx])))
until!
  {((( -y) * (-az[idx])) ≥ ((-ba) * (bb)))};
  {((bc + bd) > (be / (-bf)))}
  (next_event_a((bg - bh) < (bi mod bj))[1:3]((-bk + bl) ≥ (bm[idx] * (bn))))

```

where the argument of the outermost **next** is a suffix implication of the kind  $\{r\}(\varphi)$ . The expression  $r$  is a disjunction among two SEREs, the second of which is a conjunction of two SEREs; the formula  $\varphi$  is an **until!** expression on an **until\_** expression and a suffix implication of the kind  $\{r\}(\varphi)$ . The expression  $r$ , in this case, is a concatenation of two SEREs, while  $\varphi$  is a **next\_event\_a** expressions on propositional formulae.

## Results achieved:

With respect to [16], the properties we do not support are mainly those ruled out by constraint 5, and to a certain extent by constraint 3. We believe these restrictions do not limit the usability of NUSMV since the family of properties not supported is small; anyway, further versions of the NUSMV system will provide support for a larger subset of the language.

---

## 2.3 Implementing a translation from a subset of PSL to CTL and LTL

### Background:

The language for PSL properties is based on LTL, CTL (called OBE for Optional Branching Extension), regular expressions, a mixing of the latter with LTL. The subset of PSL which is based on LTL and the OBE can be mapped directly onto standard LTL and CTL. Moreover, some constructs that regards the boolean layer,

the temporal layer and the SERE operators can be directly translated into basic operators following the rewriting rules defined in [19].

### **Starting point:**

The NUSMV system extended with the capability of receiving in input PSL specifications.

### **Work performed:**

Proper filtering techniques have been implemented to characterize generic PSL properties with respect to the subset of the language supported for verification.

Based on the correspondences between a subset of PSL and the temporal logics LTL and CTL, we built a suite of rewriting rules to translate PSL syntactic sugar into basic PSL constructs, we designed and implemented in the NUSMV system a functionality that translates PSL properties into an internal standard NUSMV representation. Besides the already known rewriting rules, we also implemented a set of rules that allow to rewrite SERE-based specifications into LTL when proper assumptions on the use of SERE operators are satisfied (i.e. the specification belongs to the subset presented in Section 2.2), for more details see [5]. This translation moreover implements some optimizations with respect to the rewriting rules in [19] that helps on keeping under control the size of the internal representation of a PSL property, which in some cases may grow bigger than the size of the original property (this applies in particular to **next** and **next\_event** classes of operators).

A heavy testing activity has been carried out to check both the correctness of the implementation of the filtering techniques, and the correctness of the implementation of the translation process.

### **Results achieved:**

- suite of rewriting rules to translate the LTL fragment of PSL into NUSMV LTL;
- suite of rewriting rules to translate the OBE fragment of PSL into NUSMV CTL;
- translation mechanism to derive NUSMV LTL formulae from a subset of PSL SEREs;
- NUSMV has been extended to be able to deal natively with a subset of the PSL properties, by means of leveraging the existing verification techniques.

# 3 Extending existing functionalities

---

## 3.1 Definition of a new command for verification of PSL specifications

### Background:

NUSMV provides its users with a rich set of commands that implement verification techniques ranging from model checking (both for LTL and CTL) to simulation, from BDD-based algorithms to SAT-based algorithms. For a subset of the PSL language it is possible to define a mapping in LTL or CTL, see Section 2.3 and [5]. The mapping is based on a one-to-one correspondence of some PSL operators and operators of the temporal logics, and on rewriting rules that allow to translate complex PSL constructs in basic one that can be on their turn translated into LTL or CTL.

### Starting point:

The NUSMV system extended with the capability of receiving in input PSL specifications, and able to translate a subset of those specifications into an internal standard LTL-based or CTL-based representation.

### Work performed:

A new verification command has been implemented that applies to PSL properties. This command implements a two-phase process:

- based on the syntactic characteristics of the properties, i.e. the PSL subset the properties belong to, performs a post-processing of the properties translating them into standard LTL or CTL;
- leverages the verification algorithms already implemented in NUSMV to perform the proper analysis of the translated properties.

The existing verification engines have been extended in order to provide the users with feedback (e.g. counterexamples, witnesses) referring to the original PSL property and not on its internal representation. Hence the translation process is completely transparent to the user that can thus interact with the tool by using exclusively PSL, and do not need to know anything about the tool internals.

## Results achieved:

- NUSMV has been extended to be able to deal natively with a subset of the PSL properties, by means of leveraging the existing verification techniques that were defined to cope with LTL and CTL specifications. The user need not to be aware of the translation process that remains an internal phase of the processing of PSL properties.
- The NUSMV set of commands has been extended with the following command to deal with PSL specifications:

```
check_pslspec [-h] [-m | -o output-file]
               [-n number | -p "psl_expr [IN context]"]
               [-b [-i] [-g] [-l] [-k bmc_lenght]
               [-l loopback]]}
```

Depending on the characteristics of the PSL property and on the options, the command applies CTL-based model checking, or LTL-based, possibly bounded model checking.

A `psl_expr` to be checked can be specified at command line using option *p*. Alternatively, option *n* can be used for checking a particular formula in the property database. If neither *n* nor *p* are used, all the PSLSPEC formulas in the database are checked. If option *b* is used, LTL bounded model checking is applied, otherwise BDD-based model checking is applied. For LTL bounded model checking, options *k* and *l* can be used to define the maximum problem bound, and the value of the loop-back for the single generated problems respectively; their values can be stored in the environment variables `bmc_lenght` and `bmc_loopback` (see [13] for details). Single problems can be generated by using option *l*. By using option *i* the incremental version of bounded model checking is activated. Bounded model checking problems can be generated and dumped in a file by using option *g* (for generate).

In the following are the meaning and use of the command options:

*-m*: Pipes the output generated by the command in processing PSLSPECs to the program specified by the `PAGER` shell variable if defined, else through the command `more`.

*-o output-file*: Writes the output generated by the command in processing PLSPECS to the file *output-file*

*-p "psl\_expr [IN context]"*: A PSL formula to be checked. *context* is the module instance name which the variables in *psl\_expr* must be evaluated in.

*-n number*: Checks the PSL property with index *number* in the property database.

*-b*: Applies SAT-based bounded model checking. The SAT solver to be used will be chosen according to the current value of the system variable *sat\_solver* [13].

*-i*: Applies incremental SAT-bounded model checking if available, i.e. if an incremental SAT solver has been linked to NUSMV. This option can be used only in combination with the option *b*.

*-g*: Dumps DIMACS version of bounded model checking problem into a file whose name depends on the system variable *bmc\_dimacs\_filename* [13]. This feature is not allowed in combination of the option *i*.

*-l*: Generates a single bounded model checking problem with fixed bound and loop-back values, it does not iterate incrementing the value of the problem bound.

*-k bmc\_length* : *bmc\_length* is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc\_length* is considered instead [13].

*-l loopback*: The *loopback* value may be:

- a natural number in  $(0, max\_length-1)$ . A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process.
- a negative number in  $(-1, -bmc\_length)$ . In this case *loopback* is considered a value relative to *max\_length*. Any invalid combination of length and loopback will be skipped during the generation/solving process.
- the symbol X, which means “no loop-back”.
- the symbol \*, which means “all possible loop-backs from zero to *length-1*”.

If no value is given the environment variable *bmc\_loopback* is considered instead [13].



# 4 Technical details

In this section we describe the changes we made to the NUSMV system and software architecture in order to extend the system to deal with PSL, namely: the extension of the standard parser, the creation of a new package that includes all the functionalities needed to deal with PSL properties, the linking of already existing functionalities to the new ones. Last, we also describe a preliminary activity of porting to PSL of some example files currently belonging to the NUSMV distribution. These converted example files have been currently included in the NUSMV distribution.

---

## 4.1 Extended NuSMV architecture

The NUSMV system is structured in modules, each providing a set of related functionalities and communicating with others via a precisely defined interface, e.g. parsing, model checking of CTL properties, bounded model checking. Each module is implemented in one or more packages, possibly structured hierarchically. Each package exports a set of routines that manipulate the data structures defined in the package and modify the options associated to the functionalities provided by the package itself; moreover each package is associated with a set of commands that can be interpreted by the NUSMV interactive shell. Extending NUSMV to provide support for PSL has meant adding a new module, and the relative package, to the architecture.

### The `psl` package

The `psl` package includes the following files:

**`psl_input.l`** The input file for the `lex` token scanner generator.

**`psl_grammar.y`** The input file for the `bison` compiler of compilers.

**`pslNode.h`** This file contains the declarations of the `PslNode_ptr` data structure we developed to handle PSL specifications, and of the prototypes for the functions and procedures to manipulate such data structure (e.g. constructors of expressions, getters of arguments for PSL operators, predicates, converters). All the data structures and functions here declared are those exported by the `psl` package.

- pslInt.h** This file contains the private interface of the `psl` package.
- pslExpr.h** This file contains the declaration of the data structure `PslExpr`, used within the PSL parser, and functions used to represent and to manipulate the parse tree built by the PSL parser.
- pslExpr.c** This file contains the definition of the data structures and functions used by the PSL parser.
- pslNode.c** This file is the core file for the `psl` package. It contains the definition of the `PslNode_ptr`. Moreover it contains the definition for the functions and procedures to manipulate the `PslNode_ptr` data structure (e.g. constructors of expressions, getters of arguments for PSL operators, predicates).
- pslPrint.c** This file contains the definition for the functions to print out PSL specifications parsed.
- pslConv.c** This file contains the definition for the functions to convert PSL expressions first by expanding syntactic sugar constructs as described in [19]; second, by applying some of the rules described in [5], finally to convert the resulting formula into NUSMV internal structures that can be manipulated by the existing engines implemented within NUSMV.

The standard NUSMV parser has been modified to make it read its input character by character instead of buffering it; this allows to recognize the token `PSLSPEC` as soon as its last character is read. The `PSLSPEC` token identifies the beginning of a PSL specification, and its recognition triggers a shift in the control from the standard parser to the PSL parser. Since the input stream is read character by character, this shift can be performed without compromising its integrity, i.e. passing to the PSL parser a pointer to the exact beginning of the PSL specification; buffering of the input stream could mean reading more characters than actually needed, thus having the two parsers exchanging incomplete data. The PSL parser parses its input until the terminating character `;` is met, then passes the control back to the standard NUSMV parser.

---

## 4.2 Extended NuSMV flow

The diagram in Figure 1 describes the standard NUSMV flow, that is the way specifications are routed through the system internals and incrementally transformed into a representation apt to be passed as input to the verification engines.

The process starts from the modular description of a model  $M$  and of a set of properties  $P_1, \dots, P_n$ . The specification is read and parsed based on the standard NUSMV language grammar.

The parsing has been extended to deal with PSL specifications, as described in Section 2.1 and Section 4.1.

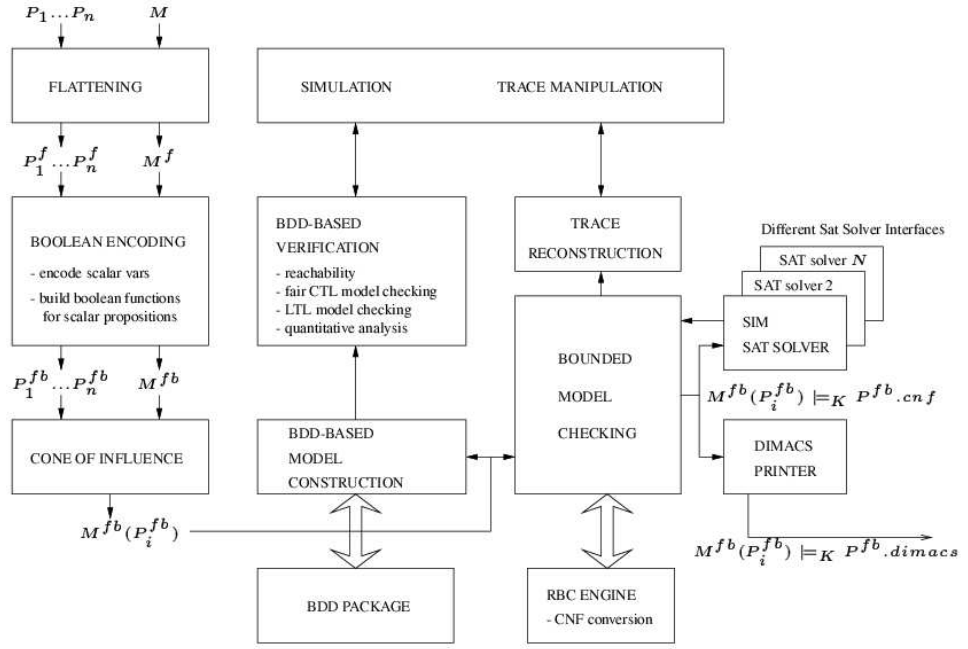


Figure 1: The NUSMV architecture.

The first step, called *flattening*, performs the instantiation of module types, thus creating modules and processes, and produces a synchronous, flat model  $M^f$ , where each variable is given an absolute name.

The flattener module has been extended in order to be able to carry out its tasks also on PSL specifications, i.e. it can now handle PSL parsing trees beside standard NUSMV parsing trees.

The standard flow has been extended adding a functional block after the flattening phase. PSL (flattened) specifications are intercepted and properly handled in order to map them onto NUSMV specifications (CTL or LTL depending on the specification); once this translation has been performed, the specifications follows the usual flow and are processed by NUSMV verification engines. The second step, called *boolean encoding*, maps a flat model into a boolean model  $M^{fb}$ , thus eliminating scalar variables. This second step takes into account the whole SMV language, including the encoding of bounded integers, and all the set-theoretic and arithmetic functions and predicates. It is possible to print out the different levels of the input file, thus using NUSMV as a flattener. The same reduction steps are applied to the properties  $P_i$ , thus obtaining the corresponding flattened boolean versions  $P_i^{fb}$ . In addition, by means of the *cone of influence* reduction [1], it is possible to restrict the analysis of each property to the relevant parts of the model  $M^{fb}(P_i^{fb})$ . The preprocessing is carried out independently from the model checking engine to be used for verification. After this, the user can choose whether to apply BDD-based or SAT-based model checking. In the case of BDD-based model checking, a BDD-based representation of the Finite State Machine (FSM) is constructed. In this step, different partitioning methods and strategies [20] can be used. Then, different forms of *BDD-based verification* can be applied: reachability analysis, fair CTL model checking, LTL model checking via reduction to CTL model checking, computation of quantitative characteristics of the model.

In the case of SAT-based model checking, NUSMV constructs an internal representation of the model based on a simplified version of *Reduced Boolean Circuit* (RBC), a representation mechanism for propositional formulae. Then, it is possible to perform SAT-based *bounded model checking* of LTL formulae [3]. Given a bound on the length of the counterexample, a LTL model checking problem is encoded into a SAT problem. If a propositional model is found, it corresponds to a counterexample of the original model checking problem. NUSMV represents each SAT problem as an RBC, that is then converted in CNF format and given in input to the internal SAT solver. Alternatively, the SAT problems can be printed out in the standard *DIMACS* format, thus allowing for the stand-alone use of other SAT solvers. In bounded model checking, NUSMV enters a loop, interleaving problem generation and solution attempt via a call to the SAT solver, and iterates until a solution is found or the specified maximum bound is reached.

NUSMV uses SIM [11] as the internal SAT solver and has a generic interface to SAT solvers to allow for the use of new state of the art SAT solvers. Such an interface has been used to integrate the CHAFF [12] and MiniSAT [9] state-of-the-art SAT solvers.

The different properties that are checked on a FSM are handled and shown to the user by a property manager, that is independent of the model checking engine used for the verification. This means that it is possible for the user to decide what solution method to adopt for each property.

The property manager has been extended to provide to the user feedback based on the PSL form of the specification rather than on the internal NUSMV representation.

Furthermore, the counterexample *traces* being generated by both model checking modules are presented and stored into a unique format.

The presentation of counterexamples has been extended to show the user the results of the processing of PSL specifications in terms of the original PSL specification and not in terms of its internal representation.

Similarly, the user can *simulate* the behavior of the specified system, by generating traces either interactively or randomly. Simulation can be carried out both via BDD-based or SAT-based techniques.

---

### 4.3 Ported some NuSMV examples to PSL

The NUSMV distribution includes several example files contributed by some of the users of the NUSMV system. Some of them, containing properties that could be expressed more concisely and intuitively in PSL have been ported to PSL. In particular for any property described in the original file the corresponding PSL formula was added. Moreover, in some cases several equivalent more intuitive PSL formulas have been added to show the expressive power of the PSL language.

# 5 References

- [1] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *Proc. COMPOS*, 1997.
- [2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W.R. Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems. 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, July 1999.
- [3] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, 1999.
- [4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
- [5] A. Cimatti, M. Roveri, and S. Semprini. Towards model checking of PSL. To be published as technical report of ITC-IRST.
- [6] E. M. Clarke, E. A. Emerson, , and A. P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, 1986.
- [7] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.
- [8] Confluence Logic Design Language. <http://www.confluence.org>.
- [9] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.
- [10] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In *CAV*, pages 136–145, 1990.
- [11] E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proc. of IJCAR 2001*, volume 2083 of *LNCS*, pages 347–363. Springer, 2001.
- [12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. of the 39th Design Automation Conference*, Las Vegas, June 2001.
- [13] NUSMV User Manual. Provided in appendix A of this deliverable.
- [14] NUSMV home page. <http://nusmv.irst.itc.it/>.
- [15] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
- [16] Property-by-example guide: a handbook of PSL/Sugar examples. PROSYD DELIVERABLE 1.1/3.
- [17] Automata Construction Algorithms Optimized for PSL. PROSYD DELIVERABLE 3.2/4.
- [18] Language Front-End for Sugar Foundation Language. PROSYD DELIVERABLE 1.2/3.

- [19] Accellera, Property Specification Language - Reference Manual - Version 1.01. [http://www.eda.org/vfv/docs/psl\\_lrm-1.01.pdf](http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf), April 2003.
- [20] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *Proc. IEEE/ACM International Workshop on Logic Synthesis*, Lake Tahoe (NV), May 1995.

# A NuSMV manual

# **NuSMV 2.3 User Manual**

**Roberto Cavada, Alessandro Cimatti,  
Emanuele Olivetti, Gavin Keighren,  
Marco Pistore, Marco Roveri,  
Simone Semprini and Andrey Tchaltsev**

IRST - Via Sommarive 18, 38055 Povo (Trento) – Italy

Email: [nusmv@irst.itc.it](mailto:nusmv@irst.itc.it)

This document is part of the distribution package of the NUSMV model checker, available at <http://nusmv.irst.itc.it>.

Parts of this documents have been taken from “The SMV System - Draft”, by K. McMillan, available at:  
<http://www.cs.cmu.edu/~modelcheck/smv/smvmanual.r2.2.ps>.

Copyright ©1998-2005 by CMU and ITC-irst.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Syntax</b>	<b>6</b>
2.1	Expressions . . . . .	6
2.1.1	Simple Expressions . . . . .	6
2.1.2	Next Expressions . . . . .	8
2.2	Definition of the FSM . . . . .	9
2.2.1	State Variables . . . . .	9
2.2.2	Input Variables . . . . .	9
2.2.3	ASSIGN declarations . . . . .	10
2.2.4	TRANS declarations . . . . .	10
2.2.5	INIT declarations . . . . .	10
2.2.6	INVAR declarations . . . . .	11
2.2.7	DEFINE declarations . . . . .	11
2.2.8	ISA declarations . . . . .	11
2.2.9	MODULE declarations . . . . .	12
2.2.10	Identifiers . . . . .	13
2.2.11	The main module . . . . .	14
2.2.12	Processes . . . . .	14
2.2.13	FAIRNESS declarations . . . . .	14
2.3	Specifications . . . . .	15
2.3.1	CTL Specifications . . . . .	15
2.3.2	LTL Specifications . . . . .	16
2.3.3	Real Time CTL Specifications and Computations . . . . .	17
2.3.4	PSL Specifications . . . . .	18
2.4	Variable Order Input . . . . .	22
2.4.1	Input File Syntax . . . . .	22
2.4.2	Scalar Variables . . . . .	23
2.4.3	Array Variables . . . . .	24
<b>3</b>	<b>Running NuSMV interactively</b>	<b>25</b>
3.1	Model Reading and Building . . . . .	26
3.2	Commands for Checking Specifications . . . . .	30
3.3	Commands for Bounded Model Checking . . . . .	35
3.4	Commands for checking PSL specifications . . . . .	45
3.5	Simulation Commands . . . . .	47
3.6	Traces . . . . .	49
3.6.1	Inspecting Traces . . . . .	49

3.6.2	Displaying Traces . . . . .	50
3.6.3	Trace Plugin Commands . . . . .	50
3.7	Trace Plugins . . . . .	51
3.7.1	Basic Trace Explainer . . . . .	52
3.7.2	States/Variables Table . . . . .	52
3.7.3	XML Format Printer . . . . .	53
3.7.4	XML Format Reader . . . . .	53
3.8	Interface to the DD Package . . . . .	54
3.9	Administration Commands . . . . .	58
3.10	Other Environment Variables . . . . .	65
<b>4</b>	<b>Running NuSMV batch</b>	<b>68</b>
<b>A</b>	<b>Compatibility with CMU SMV</b>	<b>73</b>

# Chapter 1

## Introduction

NUSMV is a symbolic model checker originated from the reengineering, reimplementation and extension of CMU SMV, the original BDD-based model checker developed at CMU [McM93]. The NUSMV project aims at the development of a state-of-the-art symbolic model checker, designed to be applicable in technology transfer projects: it is a well structured, open, flexible and documented platform for model checking, and is robust and close to industrial systems standards [CCGR00].

Version 1 of NUSMV basically implements BDD-based symbolic model checking. Version 2 of NUSMV (NUSMV2 in the following) inherits all the functionalities of the previous version, and extend them in several directions [CCG<sup>+</sup>02]. The main novelty in NUSMV2 is the integration of model checking techniques based on propositional satisfiability (SAT) [BCCZ99]. SAT-based model checking is currently enjoying a substantial success in several industrial fields, and opens up new research directions. BDD-based and SAT-based model checking are often able to solve different classes of problems, and can therefore be seen as complementary techniques.

Starting from NUSMV2, we are also adopting a new development and license model. NUSMV2 is distributed with an OpenSource license<sup>1</sup>, that allows anyone interested to freely use the tool and to participate in its development. The aim of the NUSMV OpenSource project is to provide to the model checking community a common platform for the research, the implementation, and the comparison of new symbolic model checking techniques. Since the release of NUSMV2, the NUSMV team has received code contributions for different parts of the system. Several research institutes and commercial companies have express interest in collaborating to the development of NUSMV. The main features of NUSMV are the following:

- **Functionalities.** NUSMV allows for the representation of synchronous and asynchronous finite state systems, and for the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL), using BDD-based and SAT-based model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state explosion. The interaction with the user can be carried on with a textual interface, as well as in batch mode.
- **Architecture.** A software architecture has been defined. The different components and functionalities of NUSMV have been isolated and separated in mod-

---

<sup>1</sup>(see <http://www.opensource.org>)

ules. Interfaces between modules have been provided. This reduces the effort needed to modify and extend NUSMV.

- **Quality of the implementation.** NUSMV is written in ANSIC, is POSIX compliant, and has been debugged with Purify in order to detect memory leaks. Furthermore, the system code is thoroughly commented. NUSMV uses the state of the art BDD package developed at Colorado University, and provides a general interface for linking with state-of-the-art SAT solvers. This makes NUSMV very robust, portable, efficient, and easy to understand by other people than the developers.

This document is structured as follows.

- In Chapter 2 [Syntax], page 6 we define the syntax of the input language of NUSMV.
- In Chapter 3 [Running NuSMV interactively], page 25 the commands of the interaction shell are described.
- In Chapter 4 [Running NuSMV batch], page 68 we define the batch mode of NUSMV.

NUSMV is available at <http://nusmv.irst.itc.it>.

# Chapter 2

## Syntax

We present now the complete syntax of the input language of NUSMV. In the following, an atom may be any sequence of characters starting with a character in the set  $\{A-Za-z\}$  and followed by a possibly empty sequence of characters belonging to the set  $\{A-Za-z0-9\_\$#\-\}$ . A number is any sequence of digits. A digit belongs to the set  $\{0-9\}$ .

All characters and case in a name are significant. Whitespace characters are space (`<SPACE>`), tab (`<TAB>`) and newline (`<RET>`). Any string starting with two dashes (`'--'`) and ending with a newline is a comment. Any other tokens recognized by the parser are enclosed in quotes in the syntax expressions below. Grammar productions enclosed in square brackets (`[ ]`) are optional.

### 2.1 Expressions

Expressions are constructed from variables, constants, and a collection of operators, including boolean connectives, integer arithmetic operators, case expressions and set expressions.

#### 2.1.1 Simple Expressions

Simple expressions are expressions built only from current state variables. Simple expressions can be used to specify sets of states, e.g. the initial set of states. The syntax of simple expressions is as follows:

```
simple_expr ::=
    atom                ;; a symbolic constant
  | number              ;; a numeric constant
  | "TRUE"              ;; The boolean constant 1
  | "FALSE"            ;; The boolean constant 0
  | var_id              ;; a variable identifier
  | "(" simple_expr ")"
  | "!" simple_expr    ;; logical not
  | simple_expr "&" simple_expr  ;; logical and
  | simple_expr "|" simple_expr  ;; logical or
  | simple_expr "xor" simple_expr  ;; logical exclusive or
  | simple_expr "->" simple_expr  ;; logical implication
  | simple_expr "<->" simple_expr  ;; logical equivalence
```

```

| simple_expr "=" simple_expr      ;; equality
| simple_expr "!=" simple_expr     ;; inequality
| simple_expr "<" simple_expr      ;; less than
| simple_expr ">" simple_expr      ;; greater than
| simple_expr "<=" simple_expr     ;; less than or equal
| simple_expr ">=" simple_expr     ;; greater than or equal
| simple_expr "+" simple_expr      ;; integer addition
| simple_expr "-" simple_expr      ;; integer subtraction
| simple_expr "*" simple_expr      ;; integer multiplication
| simple_expr "/" simple_expr      ;; integer division
| simple_expr "mod" simple_expr    ;; integer remainder
| set_simple_expr                  ;; a set simple_expression
| case_simple_expr                  ;; a case expression

```

A *var\_id*, (see Section 2.2.10 [Identifiers], page 13) or identifier, is a symbol or expression which identifies an object, such as a variable or a defined symbol. Since a *var\_id* can be an atom, there is a possible ambiguity if a variable or defined symbol has the same name as a symbolic constant. Such an ambiguity is flagged by the interpreter as an error.

The order of parsing precedence for operators from high to low is:

```

*, /
+, -
mod
=, !=, <, >, <=, >=
|, xor
<->
->

```

Operators of equal precedence associate to the left, except `->` that associates to the right. Parentheses may be used to group expressions.

### Case Expressions

A case expression has the following syntax:

```

case_simple_expr ::
    "case"
    simple_expr ":" simple_expr ";"
    simple_expr ":" simple_expr ";"
    ...
    simple_expr ":" simple_expr ";"
    "esac"

```

A *case\_simple\_expr* returns the value of the first expression on the right hand side of `:`, such that the corresponding condition on the left hand side evaluates to 1. Thus, if *simple\_expr* on the left side is true, then the result is the corresponding *simple\_expr* on the right side. If none of the expressions on the left hand side evaluates to 1, the result of the *case\_expression* is the numeric value 1. It is an error for any expression on the left hand side to return a value other than the truth values 0 or 1.

### Set Expressions

A set expression has the following syntax:

```

set_expr ::
    "{" set_elem "," ... "," set_elem "}" ;; set definition
  | simple_expr "in " simple_expr      ;; set inclusion test
  | simple_expr "union " simple_expr   ;; set union
set_elem :: simple_expr

```

A set can be defined by enumerating its elements inside curly braces ‘{...}’. The inclusion operator ‘in’ tests a value for membership in a set. The union operator ‘union’ takes the union of two sets. If either argument is a number or a symbolic value instead of a set, it is coerced to a singleton set.

## 2.1.2 Next Expressions

While simple expressions can represent sets of states, next expressions relate current and next state variables to express transitions in the FSM. The structure of next expressions is similar to the structure of simple expressions (See Section 2.1.1 [simple expressions], page 6). The difference is that next expression allow to refer to next state variables. The grammar is depicted below.

```

next_expr ::
    atom                ;; a symbolic constant
  | number              ;; a numeric constant
  | "TRUE"             ;; The boolean constant 1
  | "FALSE"            ;; The boolean constant 0
  | var_id             ;; a variable identifier
  | "(" next_expr ")"
  | "next" "(" simple_expr ")" ;; next value of an "expression"
  | "!" next_expr      ;; logical not
  | next_expr "&" next_expr ;; logical and
  | next_expr "|" next_expr ;; logical or
  | next_expr "xor" next_expr ;; logical exclusive or
  | next_expr "->" next_expr ;; logical implication
  | next_expr "<->" next_expr ;; logical equivalence
  | next_expr "=" next_expr ;; equality
  | next_expr "!=" next_expr ;; inequality
  | next_expr "<" next_expr ;; less than
  | next_expr ">" next_expr ;; greater than
  | next_expr "<=" next_expr ;; less than or equal
  | next_expr ">=" next_expr ;; greater than or equal
  | next_expr "+" next_expr ;; integer addition
  | next_expr "-" next_expr ;; integer subtraction
  | next_expr "*" next_expr ;; integer multiplication
  | next_expr "/" next_expr ;; integer division
  | next_expr "mod" next_expr ;; integer remainder
  | set_next_expr      ;; a set next_expression
  | case_next_expr     ;; a case expression

```

set\_next\_expr and case\_next\_expr are the same as set\_simple\_expr (see Section 2.1.1 [set expressions], page 7) and case\_simple\_expr (see Section 2.1.1 [case expressions], page 7) respectively, with the replacement of “simple ” with “next ”. The only additional production is “next ” “( simple expr )”, which allows to “shift” all the variables in simple\_expr to the next state. The next operator distributes on every operator. For instance, the formula next ( ( A & B ) |

C) is a shorthand for the formula  $(\text{next}(A) \ \& \ \text{next}(B)) \ | \ \text{next}(C)$ . It is an error if in the scope of the `next` operator occurs another `next` operator.

## 2.2 Definition of the FSM

### 2.2.1 State Variables

A state of the model is an assignment of values to a set of state variables. These variables (and also instances of modules) are declared by the notation:

```
var_declaration :: "VAR "  
                atom ":" type ";"  
                atom ":" type ";"  
                ...
```

The type associated with a variable declaration can be either a boolean, a scalar, a user defined module, or an array of any of these (including arrays of arrays).

#### Type Specifiers

A type specifier has the syntax:

```
type :: boolean  
      | "{" val "," val "," ... val "  
      | number ".." number  
      | "array " number ".." number "of " type  
      | atom [ "(" simple_expr "," simple_expr "," ... ")" ]  
      | "process " atom [ "(" simple_expr "," ... "," simple_expr ")" ]  
val  :: atom  
      | number
```

A variable of type `boolean` can take on the numerical values 0 and 1 (representing false and true, respectively). In the case of a list of values enclosed in quotes (where atoms are taken to be symbolic constants), the variable is a scalar which take any of these values. In the case of an `array` declaration, the first `simple_expr` is the lower bound on the subscript and the second `simple_expr` is the upper bound. Both of these expressions must evaluate to integer constants. Finally, an atom optionally followed by a list of expressions in parentheses indicates an instance of module atom (see Section 2.2.9 [MODULE declarations], page 12). The keyword causes the module to be instantiated as an asynchronous process (see Section 2.2.12 [processes], page 14).

### 2.2.2 Input Variables

IVAR (input variables) are used to label transitions of the Finite State Machine. The syntax for the declaration of input variables is the following:

```
ivar_declaration :: "IVAR "  
                  atom ":" type ";"  
                  atom ":" type ";"  
                  ...
```

The type associated with a variable declaration can be either a boolean, a scalar, a user defined module, or an array of any of these (including arrays of arrays) (see Section 2.2.1 [state variables], page 9).

### 2.2.3 ASSIGN declarations

An assignment has the form:

```
assign_declaration :: "ASSIGN "  
                    assign_body ";"  
                    assign_body ";"  
                    ...  
assign_body ::  
    atom           "!=" simple_expr    ;; normal assignment  
| "init" "(" atom ")" "!=" simple_expr  ;; init assignment  
| "next" "(" atom ")" "!=" next_expr    ;; next assignment
```

On the left hand side of the assignment, *atom* denotes the current value of a variable, ‘init(*atom*)’ denotes its initial value, and ‘next(*atom*)’ denotes its value in the next state. If the expression on the right hand side evaluates to an integer or symbolic constant, the assignment simply means that the left hand side is equal to the right hand side. On the other hand, if the expression evaluates to a set, then the assignment means that the left hand side is contained in that set. It is an error if the value of the expression is not contained in the range of the variable on the left hand side.

In order for a program to be implementable, there must be some order in which the assignments can be executed such that no variable is assigned after its value is referenced. This is not the case if there is a circular dependency among the assignments in any given process. Hence, such a condition is an error. It is also an error for a variable to be assigned more than once at any given time. More precisely, it is an error if any of the following occur:

- the next or current value of a variable is assigned more than once in a given process
- the initial value of a variable is assigned more than once in the program
- the current value and the initial value of a variable are both assigned in the program
- the current value and the next value of a variable are both assigned in the program

### 2.2.4 TRANS declarations

The transition relation  $R$  of the model is a set of current state/next state pairs. Whether or not a given pair is in this set is determined by a boolean valued expression  $T$ , introduced by the ‘TRANS’ keyword. The syntax of a TRANS declaration is:

```
trans_declaration :: "TRANS " trans_expr [";"]  
trans_expr        :: next_expr
```

It is an error for the expression to yield any value other than 0 or 1. If there is more than one TRANS declaration, the transition relation is the conjunction of all of TRANS declarations.

### 2.2.5 INIT declarations

The set of initial states of the model is determined by a boolean expression under the ‘INIT’ keyword. The syntax of a INIT declaration is:

```

init_declaration :: "INIT " init_expr ";"
init_expr       :: simple_expr

```

It is an error for the expression to contain the `next()` operator, or to yield any value other than 0 or 1. If there is more than one `INIT` declaration, the initial set is the conjunction of all of the `INIT` declarations.

### 2.2.6 INVAR declarations

The set of invariant states (i.e. the analogous of normal assignments, as described in Section 2.2.3 [ASSIGN declarations], page 10) can be specified using a boolean expression under the ‘`INVAR`’ keyword. The syntax of a `INVAR` declaration is:

```

invar_declaration :: "INVAR " invar_expr ";"
invar_expr       :: simple_expr

```

It is an error for the expression to contain the `next()` operator, or to yield any value other than 0 or 1. If there is more than one `INVAR` declaration, the invariant set is the conjunction of all of the `INVAR` declarations.

### 2.2.7 DEFINE declarations

In order to make descriptions more concise, a symbol can be associated with a commonly expression. The syntax for this kind of declaration is:

```

define_declaration :: "DEFINE "
                    atom " := " simple_expr ";"
                    atom " := " simple_expr ";"
                    ...
                    atom " := " simple_expr ";"

```

Whenever an identifier referring to the symbol on the left hand side of the ‘`:=`’ in a `DEFINE` occurs in an expression, it is replaced by the expression on the right hand side. The expression on the right hand side is always evaluated in its context (see Section 2.2.9 [MODULE declarations], page 12 for an explanation of contexts). Forward references to defined symbols are allowed, but circular definitions are not allowed, and result in an error.

It is not possible to assign values to defined symbols non-deterministically. Another difference between defined symbols and variables is that while variables are statically typed, definitions are not.

### 2.2.8 ISA declarations

There are cases in which some parts of a module could be shared among different modules, or could be used as a module themselves. In `NUSMV` it is possible to declare the common parts as separate modules, and then use the `ISA` declaration to import the common parts inside a module declaration. The syntax of an `ISA` declaration is as follows:

```

isa_declaration :: "ISA " atom

```

where `atom` must be the name of a declared module. The `ISA` declaration can be thought as a simple macro expansion command, because the body of the module referenced by an `ISA` command is replaced to the `ISA` declaration.

## 2.2.9 MODULE declarations

A module is an encapsulated collection of declarations. Once defined, a module can be reused as many times as necessary. Modules can also be so that each instance of a module can refer to different data values. A module can contain instances of other modules, allowing a structural hierarchy to be built. The syntax of a module declaration is as follows.

```
module :: "MODULE " atom [ "(" atom "," atom "," ... atom ")" ]
        [ var_declaration      ]
        [ ivar_declaration     ]
        [ assign_declaration   ]
        [ trans_declaration    ]
        [ init_declaration     ]
        [ invar_declaration    ]
        [ spec_declaration     ]
        [ checkinvar_declaration ]
        [ ltlspec_declaration  ]
        [ compute_declaration  ]
        [ fairness_declaration  ]
        [ define_declaration   ]
        [ isa_declaration      ]
```

The *atom* immediately following the keyword 'MODULE ' is the name associated with the module. Module names are drawn from a separate name space from other names in the program, and hence may clash with names of variables and definitions. The optional list of atoms in parentheses are the formal parameters of the module. Whenever these parameters occur in expressions within the module, they are replaced by the actual parameters which are supplied when the module is instantiated (see below).

An *instance* of a module is created using the VAR declaration (see Section 2.2.1 [state variables], page 9). This declaration supplies a name for the instance, and also a list of actual parameters, which are assigned to the formal parameters in the module definition. An actual parameter can be any legal expression. It is an error if the number of actual parameters is different from the number of formal parameters. The semantic of module instantiation is similar to call-by-reference. For example, in the following program fragment:

```
MODULE main
...
VAR
  a : boolean;
  b : foo(a);
...
MODULE foo(x)
  ASSIGN
    x := 1;
```

the variable *a* is assigned the value 1. This distinguishes the call-by-reference mechanism from a call-by-value scheme.

Now consider the following program:

```
MODULE main
...
DEFINE
```

```

    a := 0;
VAR
    b : bar(a);
...
MODULE bar(x)
    DEFINE
        a := 1;
        y := x;

```

In this program, the value of  $y$  is 0. On the other hand, using a call-by-name mechanism, the value of  $y$  would be 1, since  $a$  would be substituted as an expression for  $x$ .

Forward references to module names are allowed, but circular references are not, and result in an error.

### 2.2.10 Identifiers

An *id*, or identifier, is an expression which references an object. Objects are instances of modules, variables, and defined symbols. The syntax of an identifier is as follows.

```

id :: atom
    | "self"
    | id "." atom
    | id "[" simple_expr "]"

```

An *atom* identifies the object of that name as defined in a VAR or DEFINE declaration. If  $a$  identifies an instance of a module, then the expression ' $a.b$ ' identifies the component object named ' $b$ ' of instance ' $a$ '. This is precisely analogous to accessing a component of a structured data type. Note that an actual parameter of module ' $a$ ' can identify another module instance ' $b$ ', allowing ' $a$ ' to access components of ' $b$ ', as in the following example:

```

MODULE main
... VAR
    a : foo(b);
    b : bar(a);
...
MODULE foo(x)
    DEFINE
        c := x.p | x.q;
MODULE bar(x)
    VAR
        p : boolean;
        q : boolean;

```

Here, the value of ' $c$ ' is the logical or of ' $p$ ' and ' $q$ '.

If ' $a$ ' identifies an array, the expression ' $a[b]$ ' identifies element ' $b$ ' of array ' $a$ '. It is an error for the expression ' $b$ ' to evaluate to a number outside the subscript bounds of array ' $a$ ', or to a symbolic value.

It is possible to refer the name the current module has been instantiated to by using the `self` builtin identifier.

```

MODULE element(above, below, token)
    VAR

```

```

    Token : boolean;
ASSIGN
    init(Token) := token;
    next(Token) := token-in;
DEFINE
    above.token-in := Token;
    grant-out := below.grant-out;
MODULE cell
    VAR
        e2 : element(self, e1, 0);
        e1 : element(e1, self, 1);
    DEFINE
        e2.token-in := token-in;
        grant-out := grant-in & !e1.grant-out;
MODULE main
    VAR c1 : cell;

```

In this example the name the `cell` module has been instantiated to is passed to the submodule `element`. In the main module, declaring `c1` to be an instance of module `cell` and defining `above.token-in` in module `e2`, really amounts to defining the symbol `c1.token-in`. When you, in the `cell` module, declare `e1` to be an instance of module `element`, and you define `grant-out` in module `e1` to be `below.grant-out`, you are really defining it to be the symbol `c1.grant-out`.

### 2.2.11 The main module

The syntax of a NUSMV program is:

```

program ::
    module_1
    module_2
    ...
    module_n

```

There must be one module with the name `main` and no formal parameters. The module `main` is the one evaluated by the interpreter.

### 2.2.12 Processes

Processes are used to model interleaving concurrency. A *process* is a module which is instantiated using the keyword ‘`process`’ (see Section 2.2.1 [state variables], page 9). The program executes a step by non-deterministically choosing a process, then executing all of the assignment statements in that process in parallel. It is implicit that if a given variable is not assigned by the process, then its value remains unchanged. Each instance of a process has a special boolean variable associated with it called `running`. The value of this variable is 1 if and only if the process instance is currently selected for execution. A process may run only when its parent is running. In addition no two processes with the same parents may be running at the same time.

### 2.2.13 FAIRNESS declarations

A fairness constraint restricts the attention only to *fair execution paths*. When evaluating specifications, the model checker considers path quantifiers to apply only to fair paths.

NUSMV supports two types of fairness constraints, namely justice constraints and compassion constraints. A justice constraint consists of a formula  $f$  which is assumed to be true infinitely often in all the fair paths. In NUSMV justice constraints are identified by keywords `JUSTICE` and, for backward compatibility, `FAIRNESS`. A compassion constraint consists of a pair of formulas  $(p, q)$ ; if property  $p$  is true infinitely often in a fair path, then also formula  $q$  has to be true infinitely often in the fair path. In NUSMV compassion constraints are identified by keyword `COMPASSION`.<sup>1</sup> If compassion constraints are used then the model must not contain any input variables. Currently, NUSMV does not enforce this so it is the responsibility of the user to make sure that this is the case.

Fairness constraints are declared using the following syntax:

```

fairness_declaration ::
    "FAIRNESS " simple_expr [";"]
  | "JUSTICE " simple_expr [";"]
  | "COMPASSION " "(" simple_expr "," simple_expr ")" [";"]

```

A path is considered fair if and only if it satisfies all the constraints declared in this manner.

## 2.3 Specifications

The specifications to be checked on the FSM can be expressed in two different temporal logics: the Computation Tree Logic CTL, and the Linear Temporal Logic LTL extended with Past Operators. It is also possible to analyze quantitative characteristics of the FSM by specifying real-time CTL specifications. Specifications can be positioned within modules, in which case they are preprocessed to rename the variables according to the containing context.

CTL and LTL specifications are evaluated by NUSMV in order to determine their truth or falsity in the FSM. When a specification is discovered to be false, NUSMV constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property.

### 2.3.1 CTL Specifications

A CTL specification is given as a formula in the temporal logic CTL, introduced by the keyword ‘SPEC’. The syntax of this declaration is:

```

spec_declaration :: "SPEC " spec_expr [";"]
spec_expr         :: ctl_expr

```

The syntax of CTL formulas recognized by the NUSMV parser is as follows:

```

ctl_expr ::
    simple_expr                ;; a simple boolean expression
  | "(" ctl_expr ")"
  | "!" ctl_expr               ;; logical not
  | ctl_expr "&" ctl_expr      ;; logical and
  | ctl_expr "|" ctl_expr      ;; logical or

```

---

<sup>1</sup>In the current version of NUSMV, compassion constraints are supported only for BDD-based LTL model checking. We plan to add support for compassion constraints also for CTL specifications and in Bounded Model Checking in the next releases of NUSMV.

```

| ctl_expr "xor" ctl_expr    ;; logical exclusive or
| ctl_expr "->" ctl_expr    ;; logical implies
| ctl_expr "<->" ctl_expr    ;; logical equivalence
| "EG" ctl_expr             ;; exists globally
| "EX" ctl_expr             ;; exists next state
| "EF" ctl_expr             ;; exists finally
| "AG" ctl_expr             ;; forall globally
| "AX" ctl_expr             ;; forall next state
| "AF" ctl_expr             ;; forall finally
| "E" "[" ctl_expr "U" ctl_expr "]" ;; exists until
| "A" "[" ctl_expr "U" ctl_expr "]" ;; forall until

```

It is an error for an expressions in a CTL formula to contain a ‘next()’ operator, or to have non-boolean components, i.e. subformulas which evaluate to a value other than 0 or 1.

It is also possible to specify invariants, i.e. propositional formulas which must hold invariantly in the model. The corresponding command is ‘INVARSPEC’, with syntax:

```
checkinvar_declaration :: "INVARSPEC " simple_expr ";"
```

This statement corresponds to

```
SPEC AG simple_expr ";"
```

but can be checked by a specialized algorithm during reachability analysis.

### 2.3.2 LTL Specifications

LTL specifications are introduced by the keyword ‘LTLSPEC’. The syntax of this declaration is:

```
ltlspec_declaration :: "LTLSPEC " ltl_expr [";"]
```

where

```

ltl_expr ::
  simple_expr                ;; a simple boolean expression
| "(" ltl_expr ")"
| "!" ltl_expr              ;; logical not
| ltl_expr "&" ltl_expr      ;; logical and
| ltl_expr "|" ltl_expr     ;; logical or
| ltl_expr "xor" ltl_expr   ;; logical exclusive or
| ltl_expr "->" ltl_expr    ;; logical implies
| ltl_expr "<->" ltl_expr    ;; logical equivalence
;; FUTURE
| "X" ltl_expr              ;; next state
| "G" ltl_expr              ;; globally
| "F" ltl_expr              ;; finally
| ltl_expr "U" ltl_expr     ;; until
| ltl_expr "V" ltl_expr     ;; releases
;; PAST
| "Y" ltl_expr              ;; previous state
| "Z" ltl_expr              ;; not previous state not
| "H" ltl_expr              ;; historically
| "O" ltl_expr              ;; once
| ltl_expr "S" ltl_expr     ;; since
| ltl_expr "T" ltl_expr     ;; triggered

```

In NUSMV, LTL specifications can be analyzed both by means of BDD-based reasoning, or by means of SAT-based bounded model checking. In the first case, NUSMV proceeds along the lines described in [CGH97]. For each LTL specification, a tableau able to recognize the behaviors falsifying the property is constructed, and then synchronously composed with the model. With respect to [CGH97], the approach is fully integrated within NUSMV, and allows for full treatment of past temporal operators. In the case of BDD-based reasoning, the counterexample generated to show the falsity of a LTL specification may contain state variables which have been introduced by the tableau construction procedure. In the second case, a similar tableau construction is carried out to encode the existence of a path of limited length violating the property. NUSMV generates a propositional satisfiability problem, that is then tackled by means of an efficient SAT solver [BCCZ99]. In both cases, the tableau constructions are completely transparent to the user.

### 2.3.3 Real Time CTL Specifications and Computations

NUSMV allows for Real Time CTL specifications [EMSS91]. NUSMV assumes that each transition takes unit time for execution. RTCTL extends the syntax of CTL path expressions with the following bounded modalities:

```
rtctl_expr ::=
    ctl_expr
    | "EBF" range rtctl_expr
    | "ABF" range rtctl_expr
    | "EBG" range rtctl_expr
    | "ABG" range rtctl_expr
    | "A" "[" rtctl_expr "BU" range rtctl_expr "]"
    | "E" "[" rtctl_expr "BU" range rtctl_expr "]"
range ::= number ".." number
```

Intuitively, the semantics of the RTCTL operators is as follows:

- $EBF\ m. .n\ p$  requires that there exists a path starting from a state, such that property  $p$  holds in a future time instant  $i$ , with  $m \leq i \leq n$
- $ABF\ m. .n\ p$  requires that for all paths starting from a state, property  $p$  holds in a future time instant  $i$ , with  $m \leq i \leq n$
- $EBG\ m. .n\ p$  requires that there exists a path starting from a state, such that property  $p$  holds in all future time instants  $i$ , with  $m \leq i \leq n$
- $ABG\ m. .n\ p$  requires that for all paths starting from a state, property  $p$  holds in all future time instants  $i$ , with  $m \leq i \leq n$
- $E\ [ p\ BU\ m. .n\ q ]$  requires that there exists a path starting from a state, such that property  $q$  holds in a future time instant  $i$ , with  $m \leq i \leq n$ , and property  $p$  holds in all future time instants  $j$ , with  $m \leq j < i$
- $A\ [ p\ BU\ m. .n\ q ]$ , requires that for all paths starting from a state, property  $q$  holds in a future time instant  $i$ , with  $m \leq i \leq n$ , and property  $p$  holds in all future time instants  $j$ , with  $m \leq j < i$

Real time CTL specifications can be defined with the following syntax, which extends the syntax for CTL specifications.

```
spec_declaration :: "SPEC " rtctl_expr [";"]
```

With the ‘COMPUTE’ statement, it is also possible to compute quantitative information on the FSM. In particular, it is possible to compute the exact bound on the delay between two specified events, expressed as CTL formulas. The syntax is the following:

```
compute_declaration :: "COMPUTE " compute_expr [";"]
```

where

```
compute_expr :: "MIN" "[" rtctl_expr "," rtctl_expr "]"
              | "MAX" "[" rtctl_expr "," rtctl_expr "]"
```

MIN [*start* , *final*] computes the set of states reachable from *start*. If at any point, we encounter a state satisfying *final*, we return the number of steps taken to reach the state. If a fixed point is reached and no states intersect *final* then *infinity* is returned. MAX [*start* , *final*] returns the length of the longest path from a state in *start* to a state in *final*. If there exists an infinite path beginning in a state in *start* that never reaches a state in *final*, then *infinity* is returned.

### 2.3.4 PSL Specifications

NUSMV allows for PSL specifications as from version 1.01 of PSL Language Reference Manual [psl03]. PSL specifications are introduced by the keyword ‘PSLSPEC’. The syntax of this declaration (as from the PSL parser distributed by IBM, [PSL]) is:

```
pslspec_declaration :: "PSLSPEC " psl_expr ";"
```

where

```
psl_expr ::
  psl_primary_expr
  | psl_unary_expr
  | psl_binary_expr
  | psl_conditional_expr
  | psl_case_expr
  | psl_property
```

The six classes above define the building blocks for `psl_property` and provide means of combining instances of that class; they are defined as follows:

```
psl_primary_expr ::
  number                ;; a numeric constant
  | boolean              ;; a boolean constant
  | var_id               ;; a variable identifier
  | "{" psl_expr "," ... "," psl_expr "}"
  | "{" psl_expr "{" psl_expr "," ... "," "psl_expr" "}" "}"
  | "(" psl_expr ")"
psl_unary_expr ::
  "+" psl_primary_expr
  | "-" psl_primary_expr
  | "!" psl_primary_expr
psl_binary_expr ::
  psl_expr "+" psl_expr
  | psl_expr "union" psl_expr
```

```

| psl_expr "in" psl_expr
| psl_expr "-" psl_expr
| psl_expr "*" psl_expr
| psl_expr "/" psl_expr
| psl_expr "%" psl_expr
| psl_expr "==" psl_expr
| psl_expr "!=" psl_expr
| psl_expr "<" psl_expr
| psl_expr "<=" psl_expr
| psl_expr ">" psl_expr
| psl_expr ">=" psl_expr
| psl_expr "&" psl_expr
| psl_expr "|" psl_expr
| psl_expr "xor" psl_expr
psl_conditional_expr ::
  psl_expr "?" psl_expr ":" psl_expr
psl_case_expr ::
  "case"
    psl_expr ":" psl_expr ";"
    ...
    psl_expr ":" psl_expr ";"
  "esac"

```

Among the subclasses of `psl_expr` we depict the class `psl_bexpr` that will be used in the following to identify purely propositional, i.e. not temporal, expressions. The class of PSL properties `psl_property` is defined as follows:

```

psl_property ::
  replicator psl_expr                ;; a replicated property
| FL_property "abort" psl_bexpr
| psl_expr "<->" psl_expr
| psl_expr "->" psl_expr
| FL_property
| OBE_property
replicator ::
  "forall" var_id [index_range] "in" value_set ":"
index_range ::
  "[" range "]"
range ::
  low_bound ":" high_bound
low_bound ::
  number
| identifier
high_bound ::
  number
| identifier
| "inf"                ;; infite high bound
value_set ::
  "{" value_range "," ... "," value_range "}"
| "boolean"
value_range ::
  psl_expr
| range

```

The instances of `FL_property` are temporal properties built using LTL operators and SEREs operators, and are defined as follows:

```

FL_property ::
  ; ; PRIMITIVE LTL OPERATORS
  "X" FL_property
  | "X!" FL_property
  | "F" FL_property
  | "G" FL_property
  | "[" FL_property "U" FL_property "]"
  | "[" FL_property "W" FL_property "]"
  ; ; SIMPLE TEMPORAL OPERATORS
  "always" FL_property
  "never" FL_property
  "next" FL_property
  "next!" FL_property
  "eventually!" FL_property
  FL_property "until!" FL_property
  FL_property "until" FL_property
  FL_property "until!_" FL_property
  FL_property "until_" FL_property
  FL_property "before!" FL_property
  FL_property "before" FL_property
  FL_property "before!_" FL_property
  FL_property "before_" FL_property
  ; ; EXTENDED NEXT OPERATORS
  "X" [number] "(" FL_property ")"
  "X!" [number] "(" FL_property ")"
  "next" [number] "(" FL_property ")"
  "next!" [number] "(" FL_property ")"
  ; ;
  "next_a" [range] "(" FL_property ")"
  "next_a!" [range] "(" FL_property ")"
  "next_e" [range] "(" FL_property ")"
  "next_e!" [range] "(" FL_property ")"
  ; ;
  "next_event!" "(" psl_bexpr ")" "(" FL_property ")"
  "next_event" "(" psl_bexpr ")" "(" FL_property ")"
  "next_event!" "(" psl_bexpr ")" "[" number "]" "(" FL_property ")"
  "next_event" "(" psl_bexpr ")" "[" number "]" "(" FL_property ")"
  ; ;
  "next_event_a!" "(" psl_bexpr ")" "[" psl_expr "]" "(" FL_property ")"
  "next_event_a" "(" psl_bexpr ")" "[" psl_expr "]" "(" FL_property ")"
  "next_event_e!" "(" psl_bexpr ")" "[" psl_expr "]" "(" FL_property ")"
  "next_event_e" "(" psl_bexpr ")" "[" psl_expr "]" "(" FL_property ")"
  ; ; OPERATORS ON SEREs
  sequence "(" FL_property ")"
  sequence "|->" sequence ["!"]
  sequence "|=>" sequence ["!"]
  ; ;
  "always" sequence
  "G" sequence
  "never" sequence
  "eventually!" sequence

```

```

;;
| "within!" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
| "within" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
| "within!_" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
| "within_" "(" sequence_or_psl_bexpr "," psl_bexpr ")" sequence
;;
| "whilenot!" "(" psl_bexpr ")" sequence
| "whilenot" "(" psl_bexpr ")" sequence
| "whilenot!_" "(" psl_bexpr ")" sequence
| "whilenot_" "(" psl_bexpr ")" sequence
sequence_or_psl_bexpr ::
    sequence
  | psl_bexpr

```

Sequences, i.e. instances of class `sequence`, are defined as follows:

```

sequence ::
    "{" SERE "}"
SERE ::
    sequence
  | psl_bexpr
;; COMPOSITION OPERATORS
| SERE ";" SERE
| SERE ":" SERE
| SERE "&" SERE
| SERE "&&" SERE
| SERE "|" SERE
;; RegExp QUALIFIERS
| SERE "[" [*] [count] "]"
| "[" [*] [count] "]"
| SERE "[+]"
| "[+]"
;;
| psl_bexpr "[=" count "]"
| psl_bexpr "[->" count "]"
count ::
    number
  | range

```

Instances of `OBE_property` are CTL properties in the PSL style and are defined as follows:

```

OBE_property ::
    "AX" OBE_property
  | "AG" OBE_property
  | "AF" OBE_property
  | "A" "[" OBE_property "U" OBE_property "]"
  | "EX" OBE_property
  | "EG" OBE_property
  | "EF" OBE_property
  | "E" "[" OBE_property "U" OBE_property "]"

```

The NUSMV parser allows to input any specification based on the grammar above, but, currently verification of PSL specifications is supported only for the OBE subset, and for a subset of PSL for which it is possible to define a translation into LTL. For

the specifications that belongs to this subsets, it is possible to apply all the verification techniques that can be applied to LTL and CTL Specifications.

## 2.4 Variable Order Input

It is possible to specify the order in which variables should appear in the BDD's generated by NUSMV. The file which gives the desired order can be read in using the `-i` option in batch mode or by setting the `input_order_file` environment variable in interactive mode.

### 2.4.1 Input File Syntax

The syntax for input files describing the desired variable ordering is as follows, where the file can be considered as a list of variable names, each of which must be on a separate line:

```
vars_list :: EMPTY
           | var_list_item vars_list

var_list_item :: var_main_id
               | var_main_id.NUMBER

var_main_id :: ATOM
             | var_main_id[NUMBER]
             | var_main_id.var_id

var_id :: ATOM
        | var_id[NUMBER]
        | var_id.var_id
```

That is, a list of variable names of the following forms:

```
Complete_Var_Name          - to specify an ordinary variable
Complete_Var_Name[index]  - to specify an array variable element
Complete_Var_Name.NUMBER  - to specify a specific bit of an encoded
                           scalar variable
```

where `Complete_Var_Name` is just the name of the variable if it appears in the module `MAIN`, otherwise it has the module name(s) prepended to the start, for example:

```
mod1.mod2...modN.varname
```

where `varname` is a variable in `modN`, and `modN.varname` is a variable in `modN-1`, and so on. Note that the module name `main` is implicitly prepended to every variable name and therefore must not be included in their declarations.

Any variable which appears in the model file, but not the ordering file is placed after all the others in the ordering. Variables which appear in the ordering file but not the model file are ignored. In both cases NUSMV displays a warning message stating these actions.

Comments can be included by using the same syntax as regular NUSMV files. That is, by starting the line with `--`.

## 2.4.2 Scalar Variables

A variable which has a finite range of values that it can take is encoded as a set of boolean variables. These boolean variables represent the binary equivalents of all the possible values for the scalar variable. Thus, a scalar variable that can take values from 0 to 7 would require three boolean variables to represent it.

It is possible to not only declare the position of a scalar variable in the ordering file, but each of the boolean variables which represent it.

If only the scalar variable itself is named then all the boolean variables which are actually used to encode it are grouped together in the BDD.

Variables which are grouped together will always remain next to each other in the BDD and in the same order. When dynamic variable re-ordering is carried out, the group of variables are treated as one entity and moved as such.

If a scalar variable is omitted from the ordering file then it will be added at the end of the variable order and the specific-bit variables that represent it will be grouped together. However, if any specific-bit variables have been declared in the ordering file (see below) then these will not be grouped with the remaining ones.

It is also possible to specify that specific-bit variables are placed elsewhere in the ordering. This is achieved by first specifying the scalar variable name in the desired location, then simply specifying `Complete_Var_Name.i` at the position where you want that bit variable to appear:

```
...
Complete_Var_Name
...
Complete_Var_Name.i
...
```

The result of doing this is that the variable representing the  $i^{th}$  bit is located in a different position to the remainder of the variables representing the rest of the bits. The specific-bit variables `varname.0`, ..., `varname.i-1`, `varname.i+1`, ..., `varname.N` are grouped together as before.

If any one bit occurs before the variable it belongs to, the remaining specific-bit variables are not grouped together:

```
...
Complete_Var_Name.i
...
Complete_Var_Name
...
```

The variable representing the  $i^{th}$  bit is located at the position given in the variable ordering and the remainder are located where the scalar variable name is declared. In this case, the remaining bit variables will not be grouped together.

This is just a short-hand way of writing each individual specific-bit variable in the ordering file. The following are equivalent:

```
...
Complete_Var_Name.0      Complete_Var_Name.0
Complete_Var_Name.1      Complete_Var_Name
:
Complete_Var_Name.N-1
...
```

where the scalar variable `Complete_Var_Name` requires  $N$  boolean variables to encode all the possible values that it may take.

It is still possible to then specify other specific-bit variables at later points in the ordering file as before.

### **2.4.3 Array Variables**

When declaring array variables in the ordering file, each individual element must be specified separately. It is not permitted to specify just the name of the array.

The reason for this is that the actual definition of an array in the model file is essentially a shorthand method of defining a list of variables that all have the same type. Nothing is gained by declaring it as an array over declaring each of the elements individually, and there is no difference in terms of the internal representation of the variables.

## Chapter 3

# Running NuSMV interactively

The main interaction mode of NUSMV is through an interactive shell. In this mode NUSMV enters a read-eval-print loop. The user can activate the various NUSMV computation steps as system commands with different options. These steps can therefore be invoked separately, possibly undone or repeated under different modalities. These steps include the construction of the model under different partitioning techniques, model checking of specifications, and the configuration of the BDD package. The interactive shell of NUSMV is activated from the system prompt as follows ('NuSMV>' is the default NuSMV shell prompt):

```
system_prompt> NuSMV -int <RET>  
NuSMV>
```

A NUSMV command is a sequence of words. The first word specifies the command to be executed. The remaining words are arguments to the invoked command. Commands separated by a ';' are executed sequentially; the NUSMV shell waits for each command to terminate in turn. The behavior of commands can depend on environment variables, similar to 'csh' environment variables.

In the following we present the possible commands followed by the related environment variables, classified in different categories. Every command answers to the option -h by printing out the command usage. When output is paged for some commands (option -m), it is piped through the program specified by the UNIX PAGER shell variable, if defined, or through the UNIX command 'more'. Environment variables can be assigned a value with the 'set' command. Command sequences to NUSMV must obey the (partial) order specified in the figure depicted on page 67. For instance, it is not possible to evaluate CTL expressions before the model is built.

A number of commands and environment variables, like those dealing with file names, accept arbitrary strings. There are a few reserved characters which must be escaped if they are to be used literally in such situations. See the section describing the `history` command, on page 59, for more information.

The verbosity of NUSMV is controlled by the following environment variable.

<b>verbose_level</b>	Environment Variable
----------------------	----------------------

Controls the verbosity of the system. Possible values are integers from 0 (no messages) to 4 (full messages). The default value is 0.

### 3.1 Model Reading and Building

The following commands allow for the parsing and compilation of the model into a BDD.

<b>read_model</b> - <i>Reads a NuSMV file into NuSMV.</i>	Command
---	---------

```
read_model [-h] [-i model-file]
```

Reads a NUSMV file. If the `-i` option is not specified, it reads from the file specified in the environment variable `input_file`.

Command Options:

<code>-i model-file</code>	Sets the environment variable <code>input_file</code> to <code>model-file</code> , and reads the model from the specified file.
----------------------------	---

<b>input_file</b>	Environment Variable
-------------------	----------------------

Stores the name of the input file containing the model. It can be set by the “set” command or by the command line option “`-i`”. There is no default value.

<b>pp_list</b>	Environment Variable
----------------	----------------------

Stores the list of pre-processors to be run on the input file before it is parsed by NUSMV. The pre-processors are executed in the order specified by this variable. The argument must either be the empty string (specifying that no pre-processors are to be run on the input file), one single pre-processor name or a space separated list of pre-processor names inside double quotes. Any invalid names are ignored. The default is none.

<b>flatten_hierarchy</b> - <i>Flattens the hierarchy of modules</i>	Command
---	---------

```
flatten_hierarchy [-h]
```

This command is responsible of the instantiation of modules and processes. The instantiation is performed by substituting the actual parameters for the formal parameters, and then by prefixing the result via the instance name.

<b>show_vars</b> - <i>Shows model's symbolic variables and their values</i>	Command
---	---------

```
show_vars [-h] [-s] [-i] [-m | -o output-file]
```

Prints symbolic input and state variables of the model with their range of values (as defined in the input file).

Command Options:

- s Prints only state variables.
- i Prints only input variables.
- m Pipes the output to the program specified by the `PAGER` shell variable if defined, else through the UNIX command “more”.
- o `output-file` Writes the output generated by the command to `output-file`.

<b>encode_variables</b> - Builds the BDD variables necessary to compile the model into a BDD.	Command
---	---------

`encode_variables [-h] [-i order-file]`

Generates the boolean BDD variables and the ADD needed to encode propositionally the (symbolic) variables declared in the model. The variables are created as default in the order in which they appear in a depth first traversal of the hierarchy.

The input order file can be partial and can contain variables not declared in the model. Variables not declared in the model are simply discarded. Variables declared in the model which are not listed in the ordering input file will be created and appended at the end of the given ordering list, according to the default ordering.

Command Options:

- i `order-file` Sets the environment variable `input_order_file` to `order-file`, and reads the variable ordering to be used from file `order-file`. This can be combined with the `write_order` command. The variable ordering is written to a file, which can be inspected and reordered by the user, and then read back in.

<b>input_order_file</b>	Environment Variable
-------------------------	----------------------

Indicates the file name containing the variable ordering to be used in building the model by the ‘`encode_variables`’ command. There is no default value.

<b>write_order_dumps_bits</b>	Environment Variable
-------------------------------	----------------------

Changes the behaviour of the command `write_order`.

When this variable is set, `write_order` will dump the bits constituting the boolean encoding of each scalar variable, instead of the scalar variable itself. This helps to work at bits level in the variable ordering file. See the command `write_order` for further information. The default value is 0.

<b>write_order</b> - Writes variable order to file.	Command
---	---------

```
write_order [-h] [-b] [(-o | -f) order-file]
```

Writes the current order of BDD variables in the file specified via the `-o` option. If no option is specified the environment variable `output_order_file` will be considered. If the variable `output_order_file` is unset (or set to an empty value) then standard output will be used.

By default, the bits constituting the scalar variables encoding are not dumped. When a variable bit should be dumped, the scalar variable which the bit belongs to is dumped instead if not previously dumped. The result is a variable ordering containing only scalar and boolean model variables.

To dump single bits instead of the corresponding scalar variables, either the option `-b` can be specified, or the environment variable `write_order_dumps_bits` must be previously set.

When the boolean variable dumping is enabled, the single bits will occur within the resulting ordering file in the same position that they occur at BDD level.

Command Options:

- |                            |  |
|----------------------------|--|
| <code>-b</code>            | Dumps bits of scalar variables instead of the single scalar variables. See also the variable <code>write_order_dumps_bits</code> .       |
| <code>-o order-file</code> | Sets the environment variable <code>output_order_file</code> to <code>order-file</code> and then dumps the ordering list into that file. |
| <code>-f order-file</code> | Alias for the <code>-o</code> option. Supplied for backward compatibility.   |

<b>output_order_file</b>	Environment Variable
--------------------------	----------------------

The file where the current variable ordering has to be written. The default value is `'temp.ord'`.

<b>build_model</b> - <i>Compiles the flattened hierarchy into a BDD</i>	Command
---	---------

```
build_model [-h] [-f] [-m Method]
```

Compiles the flattened hierarchy into a BDD (initial states, invariants, and transition relation) using the method specified in the environment variable `partition_method` for building the transition relation.

Command Options:

- |                        |  |
|------------------------|--|
| <code>-m Method</code> | Sets the environment variable <code>partition_method</code> to the value <code>Method</code> , and then builds the transition relation. Available methods are <code>Monolithic</code> , <code>Threshold</code> and <code>Iwls95CP</code> . |
|------------------------|--|

- f Forces model construction. By default, only one partition method is allowed. This option allows to overcome this default, and to build the transition relation with different partitioning methods.

<b>partition_method</b>	Environment Variable
-------------------------	----------------------

The method to be used in building the transition relation, and to compute images and preimages. Possible values are:

- **Monolithic.** No partitioning at all.
- **Threshold.** Conjunctive partitioning, with a simple threshold heuristic. Assignments are collected in a single cluster until its size grows over the value specified in the variable `conj_part_threshold`. It is possible (default) to use affinity clustering to improve model checking performance. See `affinity` variable.
- **Iwls95CP.** Conjunctive partitioning, with clusters generated and ordered according to the heuristic described in [RAP<sup>+</sup>95]. Works in conjunction with the variables `image_cluster_size`, `image_W1`, `image_W2`, `image_W3`, `image_W4`. It is possible (default) to use affinity clustering to improve model checking performance. See `affinity` variable. It is also possible to avoid (default) preordering of clusters (see [RAP<sup>+</sup>95]) by setting the `iwls95preorder` variable appropriately.

<b>conj_part_threshold</b>	Environment Variable
----------------------------	----------------------

The limit of the size of clusters in conjunctive partitioning. The default value is 0 BDD nodes.

<b>affinity</b>	Environment Variable
-----------------	----------------------

Enables affinity clustering heuristic described in [MHS00], possible values are 0 or 1. The default value is 1.

<b>image_cluster_size</b>	Environment Variable
---------------------------	----------------------

One of the parameters to configure the behaviour of the *Iwls95CP* partitioning algorithm. `image_cluster_size` is used as threshold value for the clusters. The default value is 1000 BDD nodes.

<b>image_W{1,2,3,4}</b>	Environment Variable
-------------------------	----------------------

The other parameters for the *Iwls95CP* partitioning algorithm. These attribute different weights to the different factors in the algorithm. The default values are 6, 1, 1, 6 respectively. (For a detailed description, please refer to [RAP<sup>+</sup>95].)

<b>iwls95preorder</b>	Environment Variable
-----------------------	----------------------

Enables cluster preordering following heuristic described in [RAP<sup>+</sup>95], possible values are 0 or 1. The default value is 0. Preordering can be very slow.

<b>image_verbosity</b>	Environment Variable
------------------------	----------------------

Sets the verbosity for the image method *Iwls95CP*, possible values are 0 or 1. The default value is 0.

<b>print_iwls95options</b> - <i>Prints the Iwls95 Options.</i>	Command
--	---------

```
print_iwls95options [-h]
```

This command prints out the configuration parameters of the IWLS95 clustering algorithm, i.e. `image_verbosity`, `image_cluster_size` and `image_W{1,2,3,4}`.

<b>go</b> - <i>Initializes the system for the verification.</i>	Command
---	---------

```
go [-h]
```

This command initializes the system for verification. It is equivalent to the command sequence `read_model`, `flatten_hierarchy`, `encode_variables`, `build_model`, `build_flat_model`, `build_boolean_model`. If some commands have already been executed, then only the remaining ones will be invoked.

<b>process_model</b> - <i>Performs the batch steps and then returns control to the interactive shell.</i>	Command
---	---------

```
process_model [-h] [-i model-file] [-m Method]
```

Reads the model, compiles it into BDD and performs the model checking of all the specification contained in it. If the environment variable `forward_search` has been set before, then the set of reachable states is computed. If the environment variables `enable_reorder` and `reorder_method` are set, then the reordering of variables is performed accordingly. This command simulates the batch behavior of NUSMV and then returns the control to the interactive shell.

Command Options:

<code>-i model-file</code>	Sets the environment variable <code>input_file</code> to file <code>model-file</code> , and reads the model from file <code>model-file</code> .
<code>-m Method</code>	Sets the environment variable <code>partition.method</code> to <code>Method</code> and uses it as partitioning method.

## 3.2 Commands for Checking Specifications

The following commands allow for the BDD-based model checking of a NUSMV model.

**compute\_reachable** - *Computes the set of reachable states* Command

`compute_reachable [-h]`

Computes the set of reachable states. The result is then used to simplify image and preimage computations. This can result in improved performances for models with sparse state spaces. Sometimes this option may slow down the performances because the computation of reachable states may be very expensive. The environment variable `forward_search` is set during the execution of this command.

**print\_reachable\_states** - *Prints out the number of reachable states* Command

`print_reachable_states [-h] [-v]`

Prints the number of reachable states of the given model. In verbose mode, prints also the list of all reachable states. The reachable states are computed if needed.

**check\_fsm** - *Checks the transition relation for totality.* Command

`check_fsm [-h] [-m | -o output-file]`

Checks if the transition relation is total. If the transition relation is not total then a potential deadlock state is shown.

Command Options:

- `-m` Pipes the output generated by the command to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
- `-o output-file` Writes the output generated by the command to the file `output-file`.

At the beginning reachable states are computed in order to guarantee that deadlock states are actually reachable.

**check\_fsm** Environment Variable

Controls the activation of the totality check of the transition relation during the `process_model` call. Possible values are 0 or 1. Default value is 0.

**print\_fair\_states** - *Prints out the number of fair states* Command

`print_fair_states [-h] [-v]`

Prints the number of fair states of the given model. In verbose mode, prints also the list of all fair states.

---

**print\_fair\_transitions** - *Prints out the number of fair states* Command

---

```
print_fair_transitions [-h] [-v]
```

Prints the number of fair transitions of the given model. In verbose mode, prints also the list of all fair transitions. The transitions are displayed as state-input pairs.

---

**check\_spec** - *Performs fair CTL model checking.* Command

---

```
check_spec [-h] [-m | -o output-file] [-n number | -p  
"ctl-expr [IN context]"]
```

Performs fair CTL model checking.

A `ctl-expr` to be checked can be specified at command line using option `-p`. Alternatively, option `-n` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` are used, all the SPEC formulas in the database are checked.

Command Options:

- |   |   |
|---|---|
| <code>-m</code>                             | Pipes the output generated by the command in processing SPEC <i>s</i> to the program specified by the <code>PAGER</code> shell variable if defined, else through the UNIX command “more”. |
| <code>-o output-file</code>                 | Writes the output generated by the command in processing SPEC <i>s</i> to the file <code>output-file</code> .   |
| <code>-p "ctl-expr<br/>[IN context]"</code> | A CTL formula to be checked. <code>context</code> is the module instance name which the variables in <code>ctl-expr</code> must be evaluated in.  |
| <code>-n number</code>                      | Checks the CTL property with index <code>number</code> in the property database.  |

If the `ag_only_search` environment variable has been set, then a specialized algorithm to check AG formulas is used instead of the standard model checking algorithms.

---

**ag\_only\_search** Environment Variable

---

Enables the use of an ad hoc algorithm for checking AG formulas. Given a formula of the form *AG alpha*, the algorithm computes the set of states satisfying *alpha*, and checks whether it contains the set of reachable states. If this is not the case, the formula is proved to be false.

---

**forward\_search** Environment Variable

---

Enables the computation of the reachable states during the `process_model` command and when used in conjunction with the `ag_only_search` environment variable enables the use of an ad hoc algorithm to verify invariants.

<b>check_invar</b> - <i>Performs model checking of invariants</i>	Command
---	---------

```
check_invar [-h] [-m | -o output-file] [-n number | -p
"invvar-expr [IN context]"]
```

Performs invariant checking on the given model. An invariant is a set of states. Checking the invariant is the process of determining that all states reachable from the initial states lie in the invariant. Invariants to be verified can be provided as simple formulas (without any temporal operators) in the input file via the `INVARSPEC` keyword or directly at command line, using the option `-p`.

Option `-n` can be used for checking a particular invariant of the model. If neither `-n` nor `-p` are used, all the invariants are checked.

During checking of invariant all the fairness conditions associated with the model are ignored.

If an invariant does not hold, a proof of failure is demonstrated. This consists of a path starting from an initial state to a state lying outside the invariant. This path has the property that it is the shortest path leading to a state outside the invariant.

**Command Options:**

- `-m` Pipes the output generated by the program in processing `INVARSPEC` s to the program specified by the `PAGER` shell variable if defined, else through the UNIX command "more".
- `-o output-file` Writes the output generated by the command in processing `INVARSPEC` s to the file `output-file`.
- `-p "invvar-expr [IN context]"` The command line specified invariant formula to be verified. `context` is the module instance name which the variables in `invvar-expr` must be evaluated in.

<b>check_ltlspec</b> - <i>Performs LTL model checking</i>	Command
---	---------

```
check_ltlspec [-h] [-m | -o output-file] [-n number | -p
"ltl-expr [IN context]"]
```

Performs model checking of LTL formulas. LTL model checking is reduced to CTL model checking as described in the paper by [CGH97].

A `ltl-expr` to be checked can be specified at command line using option `-p`. Alternatively, option `-n` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` are used, all the `LTLSPEC` formulas in the database are checked.

**Command Options:**

-m	Pipes the output generated by the command in processing LTL SPECS to the program specified by the PAGER shell variable if defined, else through the UNIX command "more".
-o output-file	Writes the output generated by the command in processing LTL SPECS to the file output-file.
-p "ltl-expr [IN context]"	An LTL formula to be checked. context is the module instance name which the variables in ltl-expr must be evaluated in.
-n number	Checks the LTL property with index number in the property database.

<b>compute</b> - <i>Performs computation of quantitative characteristics</i>	Command
--	---------

```
compute [-h] [-m | -o output-file] [-n number | -p
"compute-expr [IN context]"]
```

This command deals with the computation of quantitative characteristics of real time systems. It is able to compute the length of the shortest (longest) path from two given set of states.

```
MAX [ alpha , beta ]
MIN [ alpha , beta ]
```

Properties of the above form can be specified in the input file via the keyword COMPUTE or directly at command line, using option -p.

Option -n can be used for computing a particular expression in the model. If neither -n nor -p are used, all the COMPUTE specifications are computed.

Command Options:

-m	Pipes the output generated by the command in processing COMPUTES to the program specified by the PAGER shell variable if defined, else through the UNIX command "more".
-o output-file	Writes the output generated by the command in processing COMPUTES to the file output-file.
-p "compute-expr [IN context]"	A COMPUTE formula to be checked. context is the module instance name which the variables in compute-expr must be evaluated in.
-n number	Computes only the property with index number.

<b>add_property</b> - <i>Adds a property to the list of properties</i>	Command
--	---------

```
add_property [-h] [(-c | -l | -i | -q) -p "formula
[IN context]"]
```

Adds a property in the list of properties. It is possible to insert LTL, CTL, INVAR and quantitative (COMPUTE) properties. Every newly inserted property is initialized to unchecked. A type option must be given to properly execute the command.

Command Options:

-c	Adds a CTL property.
-l	Adds an LTL property.
-i	Adds an INVAR property.
-q	Adds a quantitative (COMPUTE) property.
-p "formula [IN context]"	Adds the formula specified on the command-line. context is the module instance name which the variables in formula must be evaluated in.

### 3.3 Commands for Bounded Model Checking

In this section we describe in detail the commands for doing and controlling Bounded Model Checking in NUSMV. Bounded Model Checking is based on the reduction of the bounded model checking problem to a propositional satisfiability problem. After the problem is generated, NUSMV internally calls a propositional SAT solver in order to find an assignment which satisfies the problem. Currently NUSMV supplies three SAT solvers: SIM, Zchaff and MiniSat. Notice that Zchaff and MiniSat are for non-commercial purposes only. They are therefore not included in the source code distribution or in some of the binary distributions of NUSMV.

Some commands for Bounded Model Checking use incremental algorithms. These algorithms exploit the fact that satisfiability problems generated for a particular bounded model checking problem often share common subparts. So information obtained during solving of one satisfiability problem can be used in solving of another one. The incremental algorithms usually run quicker than non-incremental ones but require a SAT solver with incremental interface. At the moment, only Zchaff and MiniSat offer such an interface. If none of these solvers are linked to NUSMV, then the commands which make use of the incremental algorithms will not be available.

It is also possible to generate the satisfiability problem without calling the SAT solver. Each generated problem is dumped in DIMACS format to a file. DIMACS is the standard format used as input by most SAT solvers, so it is possible to use NUSMV with a separate external SAT solver. At the moment, the DIMACS files can be generated only by commands which do not use incremental algorithms.

<b>bmc_setup</b> - Builds the model in a Boolean Expression format.	Command
---	---------

```
bmc_setup [-h]
```

You must call this command before use any other bmc-related command. Only one call per session is required.

<b>go_bmc</b> - Initializes the system for the BMC verification.	Command
--	---------

```
go_bmc [-h]
```

This command initializes the system for verification. It is equivalent to the command sequence `read_model, flatten_hierarchy, encode_variables, build_boolean_model, bmc_setup`. If some commands have already been executed, then only the remaining ones will be invoked.

<b>check_ltlspec_bmc</b> - Checks the given LTL specification, or all LTL specifications if no formula is given. Checking parameters are the maximum length and the loopback value	Command
--	---------

```
check_ltlspec_bmc [-h | -n idx | -p "formula [IN context]"]  
[-k max_length] [-l loopback] [-o filename]
```

This command generates one or more problems, and calls SAT solver for each one. Each problem is related to a specific problem bound, which increases from zero (0) to the given maximum problem length. Here `max_length` is the bound of the problem that system is going to generate and solve. In this context the maximum problem bound is represented by the `-k` command parameter, or by its default value stored in the environment variable `bmc_length`. The single generated problem also depends on the `loopback` parameter you can explicitly specify by the `-l` option, or by its default value stored in the environment variable `bmc_loopback`.

The property to be checked may be specified using the `-n idx` or the `-p "formula"` options. If you need to generate a DIMACS dump file of all generated problems, you must use the option `-o "filename"`.

Command Options:

- |  |  |
|--|--|
| <code>-n index</code>                  | <i>index</i> is the numeric index of a valid LTL specification formula actually located in the properties database.  |
| <code>-p "formula [IN context]"</code> | Checks the <i>formula</i> specified on the command-line. <i>context</i> is the module instance name which the variables in <i>formula</i> must be evaluated in.  |
| <code>-k max_length</code>             | <i>max_length</i> is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable <i>bmc_length</i> is considered instead.  |
| <code>-l loopback</code>               | The <i>loopback</i> value may be: <ul style="list-style-type: none"><li>• a natural number in <math>(0, max\_length-1)</math>. A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process.</li><li>• a negative number in <math>(-1, -bmc\_length)</math>. In this case <i>loopback</i> is considered a value relative to <i>max_length</i>. Any invalid combination of length and loopback will be skipped during the generation/solving process.</li></ul> |

- the symbol ‘X’, which means ‘no loopback’.
  - the symbol ‘\*’, which means ‘all possible loopbacks from zero to *length-1*’.
- o *filename* *filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
- @F: model name with path part.
  - @f: model name without path part.
  - @k: current problem bound.
  - @l: current loopback value.
  - @n: index of the currently processed formula in the property database.
  - @@: the ‘@’ character.

<b>check_ltlspec_bmc_onepb</b> - Checks the given LTL specification, or all LTL specifications if no formula is given. Checking parameters are the single problem bound and the loopback value	Command
--	---------

```
check_ltlspec_bmc_onepb [-h | -n idx | -p "formula"
[IN context]] [-k length] [-l loopback] [-o filename]
```

As command `check_ltlspec_bmc` but it produces only one single problem with fixed bound and loopback values, with no iteration of the problem bound from zero to `max_length`.

Command Options:

- n *index* *index* is the numeric index of a valid LTL specification formula actually located in the properties database. The validity of *index* value is checked out by the system.
- p "formula [IN context]" Checks the *formula* specified on the command-line. *context* is the module instance name which the variables in *formula* must be evaluated in.
- k *length* *length* is the problem bound used when generating the single problem. Only natural numbers are valid values for this option. If no value is given the environment variable `bmc_length` is considered instead.
- l *loopback* The *loopback* value may be:
  - a natural number in  $(0, \text{max\_length}-1)$ . A positive sign (+) can be also used as prefix of the number. Any invalid combination of *length* and *loopback* will be skipped during the generation/solving process.
  - a negative number in  $(-1, -\text{bmc\_length})$ . In this case *loopback* is considered a value relative to *length*. Any invalid combination of *length* and *loopback* will be skipped during the generation/solving process.

- the symbol 'x', which means "no loopback".
  - the symbol '\*', which means "all possible loopback from zero to length-1".
- o *filename*
- filename* is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
- @F: model name with path part.
  - @f: model name without path part.
  - @k: current problem bound.
  - @l: current loopback value.
  - @n: index of the currently processed formula in the property database.
  - @@: the '@' character.

<b>gen_ltlspec_bmc</b> - Dumps into one or more dimacs files the given LTL specification, or all LTL specifications if no formula is given. Generation and dumping parameters are the maximum bound and the loopback value	Command
--	---------

```
gen_ltlspec_bmc [-h | -n idx | -p "formula" [IN context]]
[-k max_length] [-l loopback] [-o filename]
```

This command generates one or more problems, and dumps each problem into a dimacs file. Each problem is related to a specific problem bound, which increases from zero (0) to the given maximum problem bound. In this short description length is the bound of the problem that system is going to dump out.

In this context the maximum problem bound is represented by the *max\_length* parameter, or by its default value stored in the environment variable `bmc_length`.

Each dumped problem also depends on the loopback you can explicitly specify by the `-l` option, or by its default value stored in the environment variable `bmc_loopback`.

The property to be checked may be specified using the `-n idx` or the `-p "formula"` options.

You may specify dimacs file name by using the option `-o filename`, otherwise the default value stored in the environment variable `bmc_dimacs_filename` will be considered.

Command Options:

- n *index* *index* is the numeric index of a valid LTL specification formula actually located in the properties database. The validity of *index* value is checked out by the system.
- p "formula" [IN context]" Checks the *formula* specified on the command-line. *context* is the module instance name which the variables in *formula* must be evaluated in.

- k *max\_length*      *max\_length* is the maximum problem bound used when increasing problem bound starting from zero. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc\_length* value is considered instead.
- l *loopback*      The *loopback* value may be:
  - a natural number in (0, *max\_length-1*). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of bound and loopback will be skipped during the generation and dumping process.
  - a negative number in (-1, *-bmc\_length*). In this case *loopback* is considered a value relative to *max\_length*. Any invalid combination of bound and loopback will be skipped during the generation process.
  - the symbol 'X', which means "no loopback".
  - the symbol '\*', which means "all possible loopback from zero to *length-1*".
- o *fi lename*      *fi lename* is the name of dumped dimacs files. If this options is not specified, variable *bmc\_dimacs.fi lename* will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:
  - @F: model name with path part.
  - @f: model name without path part.
  - @k: current problem bound.
  - @l: current loopback value .
  - @n: index of the currently processed formula in the property database.
  - @@: the '@' character.

**gen\_ltlspec\_bmc\_onepb** - *Dumps into one dimacs file the problem generated for the given LTL specification, or for all LTL specifications if no formula is explicitly given. Generation and dumping parameters are the problem bound and the loopback value* Command

```
gen_ltlspec_bmc_onepb [-h | -n idx | -p "formula"
[IN context]] [-k length] [-l loopback] [-o filename]
```

As the `gen_ltlspec_bmc` command, but it generates and dumps only one problem given its bound and loopback.

**Command Options:**

- n *index*      *index* is the numeric index of a valid LTL specification formula actually located in the properties database. The validity of *index* value is checked out by the system.

-p "formula [IN context]" Checks the formula specified on the command-line. context is the module instance name which the variables in formula must be evaluated in.

-k length length is the single problem bound used to generate and dump it. Only natural numbers are valid values for this option. If no value is given the environment variable bmc\_length is considered instead.

-l loopback The loopback value may be:

- a natural number in (0, length-1). A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation and dumping process.
- negative number in (-1, -length). Any invalid combination of length and loopback will be skipped during the generation process.
- the symbol 'X', which means "no loopback".
- the symbol '\*', which means "all possible loopback from zero to length-1".

-o fi lename fi lename is the name of the dumped dimacs file. If this options is not specified, variable bmc\_dimacs\_filename will be considered. The file name string may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are:

- @F: model name with path part
- @f: model name without path part
- @k: current problem bound
- @l: current loopback value
- @n: index of the currently processed formula in the property database
- @@: the '@' character

<b>check_ltlspec_bmc_inc</b> - Checks the given LTL specification, or all LTL specifications if no formula is given, using an incremental algorithm. Checking parameters are the maximum length and the loopback value	Command
--	---------

```
check_ltlspec_bmc_inc [-h | -n idx | -p "formula [IN context]"]
[-k max_length] [-l loopback]
```

For each problem this command incrementally generates many satisfiability subproblems and calls the SAT solver on each one of them. The incremental algorithm exploits the fact that subproblems have common subparts, so information obtained during a previous call to the SAT solver can be used in the consecutive ones. Logically, this command does the same thing as `check_ltlspec_bmc` (see the description on page 36) but usually runs considerably quicker. A SAT solver with an incremental interface is required by this command, therefore if no such SAT solver is provided then this command will be unavailable.

Command Options:

- n *index* *index* is the numeric index of a valid LTL specification formula actually located in the properties database.
- p "formula [IN context]" Checks the *formula* specified on the command-line. *context* is the module instance name which the variables in *formula* must be evaluated in.
- k *max\_Length* *max\_Length* is the maximum problem bound must be reached. Only natural numbers are valid values for this option. If no value is given the environment variable *bmc\_Length* is considered instead.
- l *loopback* The *loopback* value may be:
- a natural number in  $(0, \text{max\_Length}-1)$ . A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process.
  - a negative number in  $(-1, -\text{bmc\_Length})$ . In this case *loopback* is considered a value relative to *max\_Length*. Any invalid combination of length and loopback will be skipped during the generation/solving process.
  - the symbol 'X', which means 'no loopback'.
  - the symbol '\*', which means 'all possible loopback from zero to *length-1*'.

<b>bmc_Length</b>	Environment Variable
-------------------	----------------------

Sets the generated problem bound. Possible values are any natural number, but must be compatible with the current value held by the variable *bmc\_Loopback*. The default value is 10.

<b>bmc_Loopback</b>	Environment Variable
---------------------	----------------------

Sets the generated problem loop. Possible values are:

- Any natural number, but less than the current value of the variable *bmc\_Length*. In this case the loop point is absolute.
- Any negative number, but greater than or equal to  $-\text{bmc\_Length}$ . In this case specified loop is the loop length.
- The symbol 'X', which means 'no loopback'.
- The symbol '\*', which means 'any possible loopbacks'.

The default value is \*.

<b>bmc_dimacs_filename</b>	Environment Variable
----------------------------	----------------------

This is the default file name used when generating DIMACS problem dumps. This variable may be taken into account by all commands which belong to the *gen\_ltlspec\_bmc* family. DIMACS file name can contain special symbols which will be expanded to represent the actual file name. Possible symbols are:

- **@F** The currently loaded model name with full path.
- **@f** The currently loaded model name without path part.
- **@n** The numerical index of the currently processed formula in the property database.
- **@k** The currently generated problem length.
- **@l** The currently generated problem loopback value.
- **@@** The '@' character.

The default value is "@f k@k l@l n@n".

<b>check_invar_bmc</b> - Generates and solves the given invariant, or all invariants if no formula is given	Command
---	---------

```
check_invar_bmc [-h | -n idx | -p "formula" [IN context]]
[-a alg] [-o filename]
```

In Bounded Model Checking, invariants are proved using induction. For this, satisfiability problems for the base and induction step are generated and a SAT solver is invoked on each of them. At the moment, two algorithms can be used to prove invariants. In one algorithm, which we call "classic", the base and induction steps are built on one state and one transition, respectively. Another algorithm, which we call "een-sorensson"[ES04], can build the base and induction steps on many states and transitions. As a result, the second algorithm is more powerful.

Command Options:

- |                           |  |
|---------------------------|--|
| -n <i>index</i>           | <i>index</i> is the numeric index of a valid INVAR specification formula actually located in the property database. The validity of <i>index</i> value is checked out by the system.   |
| -p "formula [IN context]" | Checks the <i>formula</i> specified on the command-line. <i>context</i> is the module instance name which the variables in <i>formula</i> must be evaluated in.  |
| -k <i>max_length</i>      | <i>max_length</i> is the maximum problem bound that can be reached. Only natural numbers are valid values for this option. Use this option only if the "een-sorensson" algorithm is selected. If no value is given the environment variable <i>bmc_length</i> is considered instead.   |
| -a <i>alg</i>             | <i>alg</i> specifies the algorithm. The value can be <code>classic</code> or <code>een-sorensson</code> . If no value is given the environment variable <i>bmc_invar_alg</i> is considered instead.  |
| -o <i>filename</i>        | <i>filename</i> is the name of the dumped dimacs file. It may contain special symbols which will be macro-expanded to form the real file name. Possible symbols are: <ul style="list-style-type: none"> <li>• <b>@F</b>: model name with path part</li> <li>• <b>@f</b>: model name without path part</li> <li>• <b>@n</b>: index of the currently processed formula in the properties database</li> </ul> |

- @@: the '@' character

<b>gen_invar_bmc</b> - Generates the given invariant, or all invariants if no formula is given	Command
--	---------

```
gen_invar_bmc [-h | -n idx | -p "formula [IN context]" ]
[-o filename]
```

At the moment, the invariants are generated using “classic” algorithm only (see the description of `check_invar_bmc` on page 42).

Command Options:

- |                           |   |
|---------------------------|---|
| -n <i>index</i>           | <i>index</i> is the numeric index of a valid INVAR specification formula actually located in the property database. The validity of <i>index</i> value is checked out by the system.  |
| -p "formula [IN context]" | Checks the <i>formula</i> specified on the command-line. <i>context</i> is the module instance name which the variables in <i>formula</i> must be evaluated in.   |
| -o <i>filename</i>        | <i>filename</i> is the name of the dumped dimacs file. If you do not use this option the dimacs file name is taken from the environment variable <code>bmc_invar_dimacs_filename</code> . File name may contain special symbols which will be macro-expanded to form the real dimacs file name. Possible symbols are: <ul style="list-style-type: none"> <li>• @F: model name with path part</li> <li>• @f: model name without path part</li> <li>• @n: index of the currently processed formula in the properties database</li> <li>• @@: the '@' character</li> </ul> |

<b>check_invar_bmc_inc</b> - Generates and solves the given invariant, or all invariants if no formula is given, using incremental algorithms	Command
---	---------

```
check_invar_bmc_inc [-h | -n idx | -p "formula" [IN context]]
[-a algorithm]
```

This command does the same thing as `check_invar_bmc` (see the description on page 42) but uses an incremental algorithm and therefore usually runs considerably quicker. The incremental algorithms exploit the fact that satisfiability problems generated for a particular invariant have common subparts, so information obtained during solving of one problem can be used in solving another one. A SAT solver with an incremental interface is required by this command. If no such SAT solver is provided then this command will be unavailable.

There are two incremental algorithms which can be used: ‘Dual’ and ‘ZigZag’. Both algorithms are equally powerful, but may show different performance depending on a SAT solver used and an invariant being proved. At the moment, the ‘Dual’ algorithm cannot be used if there are input variables in a given model. For additional information about algorithms, consider [ES04].

Command Options:

-n <i>index</i>	<i>index</i> is the numeric index of a valid INVAR specification formula actually located in the property database. The validity of <i>index</i> value is checked out by the system.
-p "formula [IN context]"	Checks the formula specified on the command-line. context is the module instance name which the variables in formula must be evaluated in.
-k <i>max_length</i>	<i>max_length</i> is the maximum problem bound that can be reached. Only natural numbers are valid values for this option. If no value is given the environment variable <i>bmc_length</i> is considered instead.
-a <i>alg</i>	<i>alg</i> specifies the algorithm to use. The value can be <i>dual</i> or <i>zigzag</i> . If no value is given the environment variable <i>bmc_inc_invar_alg</i> is considered instead.

<b>bmc_invar_alg</b>	Environment Variable
----------------------	----------------------

Sets the default algorithm used by the command `check_invar_bmc`. Possible values are `classic` and `een-sorensson`. The default value is `classic`.

<b>bmc_inc_invar_alg</b>	Environment Variable
--------------------------	----------------------

Sets the default algorithm used by the command `check_invar_bmc_inc`. Possible values are `dual` and `zigzag`. The default value is `dual`.

<b>bmc_invar_dimacs_filename</b>	Environment Variable
----------------------------------	----------------------

This is the default file name used when generating DIMACS invar dumps. This variable may be taken into account by the command `gen_invar_bmc`. DIMACS file name can contain special symbols which will be expanded to represent the actual file name. Possible symbols are:

- **@F** The currently loaded model name with full path.
- **@f** The currently loaded model name without path part.
- **@n** The numerical index of the currently processed formula in the properties database.
- **@@** The ‘@’ character.

The default value is “@f\_invar\_n@n”.

<b>sat_solver</b>	Environment Variable
-------------------	----------------------

The SAT solver's name actually to be used. Default SAT solver is SIM. Depending on the NUSMV configuration, also the Zchaff and MiniSat SAT solvers can be available or not. Notice that Zchaff and MiniSat are for non-commercial purposes only.

<b>bmc_simulate</b> - Generates a trace of the model from 0 (zero) to k	Command
---	---------

```
bmc_simulate [-h | -k ]
```

`bmc_simulate` does not require a specification to build the problem, because only the model is used to build it. The problem length is represented by the `-k` command parameter, or by its default value stored in the environment variable `bmc_length`.

Command Options:

`-k length` *length* is the length of the generated simulation.

### 3.4 Commands for checking PSL specifications

The following command allow for model checking of PSL specifications.

<b>check_pslspec</b> - Performs PSL model checking	Command
--	---------

```
check_pslspec [-h] [-m | -o output-file] [-n number | -p
"psl-expr [IN context]"] [-b [-i] [-g] [-l] [-k
bmc_lenght] [-l loopback]]
```

Depending on the characteristics of the PSL property and on the options, the commands applies CTL-based model checking, or LTL-based, possibly bounded model checking.

A `psl-expr` to be checked can be specified at command line using option `-p`. Alternatively, option `-n` can be used for checking a particular formula in the property database. If neither `-n` nor `-p` are used, all the PSLSPEC formulas in the database are checked. If option `-b` is used, LTL bounded model checking is applied, otherwise bdd-based model checking is applied. For LTL bounded model checking, options `-k` and `-l` can be used to define the maximum problem bound, and the value of the loopback for the single generated problems respectively; their values can be stored in the environment variables `bmc_lenght` and `bmc_loopback`. Single problems can be generated by using option `-l`. By using option `-i` the incremental version of bounded model checking is activated. Bounded model checking problems can be generated and dumped in a file by using option `-g`.

### Command Options:

-m	Pipes the output generated by the command in processing PSLSPECS to the program specified by the PAGER shell variable if defined, else through the UNIX command "more".
-o <i>output-file</i>	Writes the output generated by the command in processing PSLSPEC s to the file <i>output-file</i>
-p "psl-expr [IN context]"	A PSL formula to be checked. context is the module instance name which the variables in psl-expr must be evaluated in.
-n number	Checks the PSL property with index number in the property database.
-b	Applies SAT-based bounded model checking. The SAT solver to be used will be chosen according to the current value of the system variable sat_solver.
-i	Applies incremental SAT-bounded model checking if available, i.e. if an incremental SAT solver has been linked to NuSMV. This option can be used only in combination with the option -b.
-g	Dumps DIMACS version of bounded model checking problem into a file whose name depends on the system variable bmc_dimacs_filename. This feature is not allowed in combination of the option -i.
-l	Generates a single bounded model checking problem with fixed bound and loopback values, it does not iterate incrementing the value of the problem bound.
-k <i>bmc_Length</i>	<i>bmc_Length</i> is the maximum problem bound to be checked. Only natural numbers are valid values for this option. If no value is given the environment variable <i>bmc_Length</i> is considered instead.
-l <i>loopback</i>	The <i>loopback</i> value may be: <ul style="list-style-type: none"><li>• a natural number in <math>(0, max\_Length-1)</math>. A positive sign ('+') can be also used as prefix of the number. Any invalid combination of length and loopback will be skipped during the generation/solving process.</li><li>• a negative number in <math>(-1, -bmc\_Length)</math>. In this case <i>loopback</i> is considered a value relative to <i>max_Length</i>. Any invalid combination of length and loopback will be skipped during the generation/solving process.</li><li>• the symbol 'X', which means "no loopback".</li><li>• the symbol '*', which means "all possible loopbacks from zero to <i>length-1</i>" If no value is given the environment variable <i>bmc_loopback</i> is considered instead..</li></ul>

## 3.5 Simulation Commands

In this section we describe the commands that allow to simulate a NUSMV specification. See also the section Section 3.6 [Traces], page 49 that describes the commands available for manipulating traces.

<b>pick_state</b> - <i>Picks a state from the set of initial states</i>	Command
---	---------

```
pick_state [-h] [-v] [-r | -i [-a]] [-c "constraints"]
```

Chooses an element from the set of initial states, and makes it the current state (replacing the old one). The chosen state is stored as the first state of a new trace ready to be lengthened by `steps` states by the `simulate` command. The state can be chosen according to different policies which can be specified via command line options. By default the state is chosen in a deterministic way.

Command Options:

-v	Verbosely prints out chosen state (all state variables, otherwise it prints out only the label $t.1$ of the state chosen, where $t$ is the number of the new trace, that is the number of traces so far generated plus one).
-r	Randomly picks a state from the set of initial states.
-i	Enables the user to interactively pick up an initial state. The user is requested to choose a state from a list of possible items (every item in the list doesn't show state variables unchanged with respect to a previous item). If the number of possible states is too high, then the user has to specify some further constraints as "simple expression".
-a	Displays all state variables (changed and unchanged with respect to a previous item) in an interactive picking. This option works only if the <code>-i</code> options has been specified.
-c "constraints"	Uses <code>constraints</code> to restrict the set of initial states in which the state has to be picked. <code>constraints</code> must be enclosed between double quotes " ".

<b>showed_states</b>	Environment Variable
----------------------	----------------------

Controls the maximum number of states showed during an interactive simulation session. Possible values are integers from 1 to 100. The default value is 25.

<b>simulate</b> - <i>Performs a simulation from the current selected state</i>	Command
--	---------

```
simulate [-h] [-p | -v] [-r | -i [-a]] [-c "constraints"]
steps
```

Generates a sequence of at most `steps` states (representing a possible execution of the model), starting from the `current state`. The current state must be set via the `pick_state` or `goto_state` commands.

It is possible to run the simulation in three ways (according to different command line policies): deterministic (the default mode), random and interactive.

The resulting sequence is stored in a trace indexed with an integer number taking into account the total number of traces stored in the system. There is a different behavior in the way traces are built, according to how `current state` is set: `current state` is always put at the beginning of a new trace (so it will contain at most `steps + 1` states) except when it is the last state of an existent old trace. In this case the old trace is lengthened by at most `steps` states.

#### Command Options:

- |                                  |   |
|----------------------------------|---|
| <code>-p</code>                  | Prints current generated trace (only those variables whose value changed from the previous state).  |
| <code>-v</code>                  | Verbosely prints current generated trace (changed and unchanged state variables).   |
| <code>-r</code>                  | Picks a state from a set of possible future states in a random way.   |
| <code>-i</code>                  | Enables the user to interactively choose every state of the trace, step by step. If the number of possible states is too high, then the user has to specify some constraints as simple expression. These constraints are used only for a single simulation step and are <i>forgotten</i> in the following ones. They are to be intended in an opposite way with respect to those constraints eventually entered with the <code>pick_state</code> command, or during an interactive simulation session (when the number of future states to be displayed is too high), that are <i>local</i> only to a single step of the simulation and are <i>forgotten</i> in the next one. |
| <code>-a</code>                  | Displays all the state variables (changed and unchanged) during every step of an interactive session. This option works only if the <code>-i</code> option has been specified.  |
| <code>-c</code><br>"constraints" | Performs a simulation in which computation is restricted to states satisfying those <code>constraints</code> . The desired sequence of states could not exist if such constraints were too strong or it may happen that at some point of the simulation a future state satisfying those constraints doesn't exist: in that case a trace with a number of states less than <code>steps</code> trace is obtained. Note: <code>constraints</code> must be enclosed between double quotes " ".  |

steps	Maximum length of the path according to the constraints. The length of a trace could contain less than <code>steps</code> states: this is the case in which simulation stops in an intermediate step because it may not exist any future state satisfying those constraints.
-------	--

## 3.6 Traces

A trace is a sequence of states-inputs pairs corresponding to a possible execution of the model. Each pair contains the inputs that caused the transition to the new state, and the new state itself. The initial state has no such input values defined as it does not depend on the values of any of the inputs. The values of any constants declared in `DEFINE` sections are also part of a trace. If the value of a constant depends only on state variables then it will be treated as if it is a state variable too. If it depends only on input variables then it will be treated as if it is an input variable. If however, a constant depends upon both input and state variables, then it gets displayed in a separate “combinatorial” section. Since the values of any such constants depend on one or more inputs, the initial state does not contain this section either.

Traces are created by NUSMV when a formula is found to be false; they are also generated as a result of a simulation (Section 3.5 [Simulation Commands], page 47). Each trace has a number, and the states-inputs pairs are numbered within the trace. Trace  $n$  has states/inputs  $n.1, n.2, n.3, \dots$  where  $n.1$  represents the initial state.

### 3.6.1 Inspecting Traces

The trace inspection commands of NUSMV allow for navigation along the labelled states-inputs pairs of the traces produced. During the navigation, there is a *current state*, and the *current trace* is the trace the *current state* belongs to. The commands are the following:

<b>goto_state</b> - <i>Goes to a given state of a trace</i>	Command
---	---------

```
goto_state [-h] state_label
```

Makes `state_label` the *current state*. This command is used to navigate along traces produced by NUSMV. During the navigation, there is a *current state*, and the *current trace* is the trace the *current state* belongs to.

<b>print_current_state</b> - <i>Prints out the current state</i>	Command
--	---------

```
print_current_state [-h] [-v]
```

Prints the name of the *current state* if defined.

Command Options:

-v

Prints the value of all the state variables of the *current state*.

### 3.6.2 Displaying Traces

NUSMV comes with three trace plugins (see Section 3.7 [Trace Plugins], page 51) which can be used to display traces in the system. Once a trace has been generated by NUSMV it is printed to `stdout` using the trace explanation plugin which has been set as the current default. The command `show_traces` (see Section 3.5 [Simulation Commands], page 47) can then be used to print out one or more traces using a different trace plugin, as well as allowing for output to a file.

### 3.6.3 Trace Plugin Commands

The following commands relate to the plugins which are available in NUSMV.

<b>show_plugins</b> - Shows the available trace explanation plugins	Command
---	---------

```
show_plugins [-h] [-n plugin-no | -a]
```

Command Options:

-n plugin-no	Shows the plugin with the index number equal to plugin-no.
-a	Shows all the available plugins.

Shows the available plugins that can be used to display a trace which has been generated by NUSMV, or that has been loaded with the `read_trace` command. The plugin that is used to read in a trace is also shown. The current default plugin is marked with “[D]”.

All the available plugins are displayed by default if no command options are given.

<b>default_trace_plugin</b>	Environment Variable
-----------------------------	----------------------

This determines which trace plugin will be used by default when traces that are generated by NUSMV are to be shown. The values that this variable can take depend on which trace plugins are installed. Use the command `show_plugins` to see which ones are available. The default value is 0.

<b>show_traces</b> - Shows the traces generated in a NuSMV session	Command
--	---------

```
show_traces [-h] [-v] [-t] [-m | -o output-file] [-p plugin-no] [-a | trace_number]
```

Shows the traces currently stored in system memory, if any. By default it shows the last generated trace, if any.

#### Command Options:

-v	Verbosely prints traces content (all state variables, otherwise it prints out only those variables that have changed their value from previous state). This option only applies when the Basic Trace Explainer plugin is used to display the trace.
-t	Prints only the total number of currently stored traces.
-a	Prints all the currently stored traces.
-m	Pipes the output through the program specified by the PAGER shell variable if defined, else through the UNIX command ‘more’.
-o output-file	Writes the output generated by the command to output-file.
-p plugin-no trace_number	Uses the specified trace plugin to display the trace. The (ordinal) identifier number of the trace to be printed. This must be the last argument of the command. Omitting the trace number causes the most recently generated trace to be printed.

If the XML Format Output plugin is being used to save generated traces to a file with the intent of reading them back in again at a later date, then only one trace should be saved per file. This is because the trace reader does not currently support multiple traces in one file.

<b>read_trace</b> - Loads a previously saved trace	Command
--	---------

```
read_trace [-h | -i file-name]
```

#### Command Options:

-i file-name	Reads in a trace from the specified file. Note that the file must only contain one trace.
--------------	---

Loads a trace which has been previously output to a file with the XML Format Output plugin. The model from which the trace was originally generated must be loaded and built using the command “go” first.

Please note that this command is only available on systems that have the Expat XML parser library installed.

## 3.7 Trace Plugins

NUSMV comes with three plugins which can be used to display a trace that has been generated:

- Basic Trace Explainer
- States/Variables Table
- XML Format Printer

There is also a plugin which can read in any trace which has been output to a file by the XML Format Printer. Note however that this reader is only available on systems

that have the Expat XML parser library installed.

Once a trace has been generated it is output to `stdout` using the currently selected plugin. The command `show_traces` can be used to output any previously generated, or loaded, trace to a specific file.

### 3.7.1 Basic Trace Explainer

This plugin prints out each state (the current values of the variables) in the trace, one after the other. The initial state contains all the state variables and their initial values. States are numbered in the following fashion:

```
trace_number.state_number
```

There is the option of printing out the value of every variable in each state, or just those which have changed from the previous one. The one that is used can be chosen by selecting the appropriate trace plugin. The values of any constants which depend on both input and state variables are printed next. It then prints the set of inputs which cause the transition to a new state (if the model contains inputs), before actually printing the new state itself. The set of inputs and the subsequent state have the same number associated to them.

In the case of a looping trace, if the next state to be printed is the same as the last state in the trace, a line is printed stating that this is the point where the loop begins.

With the exception of the initial state, for which no input values are printed, the output syntax for each state is as follows:

```
-> Input: TRACE_NO.STATE_NO <-
    /* for each input var (being printed), i: */
    INPUT_VARI = VALUE
-> State: TRACE_NO.STATE_NO <-
    /* for each state var (being printed), j: */
    STATE_VARj = VALUE
    /* for each combinatorial constant (being printed), k: */
    CONSTANTk = VALUE
```

where `INPUT_VAR`, `STATE_VAR` and `CONSTANT` have the relevant module names prepended to them (seperated by a period) with the exception of the module "main".

The version of this plugin which only prints out those variables whose values have changed is the initial default plugin used by NUSMV.

### 3.7.2 States/Variables Table

This trace plugin prints out the trace as a table, either with the states on each row, or in each column. The entries along the state axis are:

```
S0 C1 I1 S1 ... Cn In Sn
```

where  $S_0$  is the initial state, and  $I_i$  gives the values of the input variables which caused the transition from state  $S_{i-1}$  to state  $S_i$ .  $C_i$  gives the values of any combinatorial constants, where the value depends on the values of the state variables in state  $S_{i-1}$  and the values of input variables in state  $S_i$ .

The variables in the model are placed along the other axis. Only the values of state variables are displayed in the State row/column, only the values of input variables are displayed in the Input row/column and only the values of combinatorial constants are displayed in the Constants row/column. All remaining cells have '-' displayed.

### 3.7.3 XML Format Printer

This plugin prints out the trace either to `stdout` or to a specified file using the command `show_traces`. If traces are to be output to a file with the intention of them being loaded again at a later date, then each trace must be saved in a separate file. This is because the XML Reader plugin does not currently support multiple traces per file. The format of a dumped XML trace file is as follows:

```
<?XML_VERSION_STRING?>
<counter-example type=TRACE_TYPE desc=TRACE_DESC>

  /* for each state, i: */
  <node>
    <state id=i>

      /* for each state var, j: */
      <value variable=j>VALUE</value>

    </state>
    <combinatorial id=i+1>

      /* for each combinatorial constant, k: */
      <value variable=k>VALUE</value>

    </combinatorial>
    <input id=i+1>

      /* for each input var, l: */
      <value variable=l>VALUE</value>

    </input>
  </node>

</counter-example>
```

Note that for the last state in the trace, there is no input section in the node tags. This is because the inputs section gives the new input values which cause the transition to the next state in the trace. There is also no combinatorial section as this depends on the values of the inputs and are therefore undefined when there are no inputs.

### 3.7.4 XML Format Reader

This plugin makes use of the Expat XML parser library and as such can only be used on systems where this library is available. Previously generated traces for a given model

can be loaded using this plugin provided that the original model file<sup>1</sup> has been loaded, and built using the command `go`.

When a trace is loaded, it is given the smallest available trace number to identify it. It can then be manipulated in the same way as any generated trace.

### 3.8 Interface to the DD Package

NUSMV uses the state of the art BDD package CUDD [Som98]. Control over the BDD package can very important to tune the performance of the system. In particular, the order of variables is critical to control the memory and the time required by operations over BDDs. Reordering methods can be activated to determine better variable orders, in order to reduce the size of the existing BDDs.

Reordering of the variables can be triggered in two ways: by the user, or by the BDD package. In the first way, reordering is triggered by the interactive shell command `dynamic_var_ordering` with the `-f` option.

Reordering is triggered by the BDD package when the number of nodes reaches a given threshold. The threshold is initialized and automatically adjusted after each reordering by the package. This is called dynamic reordering, and can be enabled or disabled by the user. Dynamic reordering is enabled with the shell command `dynamic_var_ordering` with the option `-e`, and disabled with the `-d` option.

<code>enable_reorder</code>	Environment Variable
-----------------------------	----------------------

Specifies whether dynamic reordering is enabled (when value is '0') or disabled (when value is '1').

<code>reorder_method</code>	Environment Variable
-----------------------------	----------------------

Specifies the ordering method to be used when dynamic variable reordering is fired. The possible values, corresponding to the reordering methods available with the CUDD package, are listed below. The default value is `sift`.

- `sift`: Moves each variable throughout the order to find an optimal position for that variable (assuming all other variables are fixed). This generally achieves greater size reductions than the window method, but is slower.
- `random`: Pairs of variables are randomly chosen, and swapped in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained. The number of pairs chosen for swapping equals the number of variables in the diagram.

---

<sup>1</sup>To be exact,  $M_1 \subseteq M_2$ , where  $M_1$  is the model from which the trace was generated, and  $M_2$  is the currently loaded, and built, model. Note however, that this may mean that the trace is not valid for the model  $M_2$ .

<code>random_pivot:</code>	Same as <code>random</code> , but the two variables are chosen so that the first is above the variable with the largest number of nodes, and the second is below that variable. In case there are several variables tied for the maximum number of nodes, the one closest to the root is used.
<code>sift_converge:</code>	The <code>sift</code> method is iterated until no further improvement is obtained.
<code>symmetry_sift:</code>	This method is an implementation of symmetric sifting. It is similar to sifting, with one addition: Variables that become adjacent during sifting are tested for symmetry. If they are symmetric, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable.
<code>symmetry_sift_converge:</code>	The <code>symmetry_sift</code> method is iterated until no further improvement is obtained.
<code>window{2,3,4}:</code>	Permutes the variables within windows of $n$ adjacent variables, where $n$ can be either 2, 3 or 4, so as to minimize the overall BDD size.
<code>window{2,3,4}_converge:</code>	The <code>window{2,3,4}</code> method is iterated until no further improvement is obtained.
<code>group_sift:</code>	This method is similar to <code>symmetry_sift</code> , but uses more general criteria to create groups.
<code>group_sift_converge:</code>	The <code>group_sift</code> method is iterated until no further improvement is obtained.
<code>annealing:</code>	This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.
<code>genetic:</code>	This method is an implementation of a genetic algorithm for variable ordering. This method is potentially very slow.
<code>exact:</code>	This method implements a dynamic programming approach to exact reordering. It only stores one BDD at a time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 boolean variables.
<code>linear:</code>	This method is a combination of sifting and linear transformations.
<code>linear_conv:</code>	The <code>linear</code> method is iterated until no further improvement is obtained.

<b>dynamic_var_ordering</b> - <i>Deals with the dynamic variable ordering.</i>	Command
--	---------

`dynamic_var_ordering [-d] [-e <method>] [-f <method>] [-h]`

Controls the application and the modalities of (dynamic) variable ordering. Dynamic ordering is a technique to reorder the BDD variables to reduce the size of the existing BDDs. When no options are specified, the current status of dynamic ordering is displayed. At most one of the options `-e`, `-f`, and `-d` should be specified. Dynamic ordering may be time consuming, but can often reduce the size of the BDDs dramatically. A good point to invoke dynamic ordering explicitly (using the `-f` option) is after the commands `build_model`, once the transition relation has been built. It is possible to save the ordering found using `write_order` in order to reuse it (using `build_model -i order-file`) in the future.

#### Command Options:

- |                                |  |
|--------------------------------|--|
| <code>-d</code>                | Disable dynamic ordering from triggering automatically.  |
| <code>-e &lt;method&gt;</code> | Enable dynamic ordering to trigger automatically whenever a certain threshold on the overall BDD size is reached. <code>&lt;method&gt;</code> must be one of the following: <ul style="list-style-type: none"> <li>• <b>sift</b>: Moves each variable throughout the order to find an optimal position for that variable (assuming all other variables are fixed). This generally achieves greater size reductions than the window method, but is slower.</li> <li>• <b>random</b>: Pairs of variables are randomly chosen, and swapped in the order. The swap is performed by a series of swaps of adjacent variables. The best order among those obtained by the series of swaps is retained. The number of pairs chosen for swapping equals the number of variables in the diagram.</li> <li>• <b>random_pivot</b>: Same as <b>random</b>, but the two variables are chosen so that the first is above the variable with the largest number of nodes, and the second is below that variable. In case there are several variables tied for the maximum number of nodes, the one closest to the root is used.</li> <li>• <b>sift_converge</b>: The <b>sift</b> method is iterated until no further improvement is obtained.</li> <li>• <b>symmetry_sift</b>: This method is an implementation of symmetric sifting. It is similar to sifting, with one addition: Variables that become adjacent during sifting are tested for symmetry. If they are symmetric, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable.</li> <li>• <b>symmetry_sift_converge</b>: The <b>symmetry_sift</b> method is iterated until no further improvement is obtained.</li> <li>• <b>window{2,3,4}</b>: Permutes the variables within windows of "n" adjacent variables, where "n" can be either 2, 3 or 4, so as to minimize the overall BDD size.</li> </ul> |

- **window{2,3,4}\_converge**: The **window{2,3,4}** method is iterated until no further improvement is obtained.
- **group\_sift**: This method is similar to **symmetry\_sift**, but uses more general criteria to create groups.
- **group\_sift\_converge**: The **group\_sift** method is iterated until no further improvement is obtained.
- **annealing**: This method is an implementation of simulated annealing for variable ordering. This method is potentially very slow.
- **genetic**: This method is an implementation of a genetic algorithm for variable ordering. This method is potentially very slow.
- **exact**: This method implements a dynamic programming approach to exact reordering. It only stores a BDD at a time. Therefore, it is relatively efficient in terms of memory. Compared to other reordering strategies, it is very slow, and is not recommended for more than 16 boolean variables.
- **linear**: This method is a combination of sifting and linear transformations.
- **linear\_converge**: The **linear** method is iterated until no further improvement is obtained.

-f <method>

Force dynamic ordering to be invoked immediately. The values for <method> are the same as in option -e.

**print\_bdd\_stats** - *Prints out the BDD statistics and parameters*

Command

```
print_bdd_stats [-h]
```

Prints the statistics for the BDD package. The amount of information depends on the BDD package configuration established at compilation time. The configuration parameters are printed out too. More information about statistics and parameters can be found in the documentation of the CUDD Decision Diagram package.

**set\_bdd\_parameters** - *Creates a table with the value of all currently active NuSMV flags and change accordingly the configurable parameters of the BDD package.*

Command

```
set_bdd_parameters [-h] [-s]
```

Applies the variables table of the NUSMV environment to the BDD package, so the user can set specific BDD parameters to the given value. This command works in conjunction with the `print_bdd_stats` and `set` commands. `print_bdd_stats` first prints a report of the parameters and statistics of the current `bdd_manager`. By using the command `set`, the user may modify the value of any of the parameters of the underlying BDD package. The way to do it is by setting a value in the variable `BDD.parameter_name` where `parameter_name` is the name of the parameter exactly as printed by the `print_bdd_stats` command.

Command Options:

<code>-s</code>	Prints the BDD parameter and statistics after the modification.
-----------------	---

### 3.9 Administration Commands

This section describes the administrative commands offered by the interactive shell of NuSMV.

<code>!</code> - <i>shell_command</i>	Command
---------------------------------------	---------

`!` executes a shell command. The `shell_command` is executed by calling `bin/sh -c shell_command`. If the command does not exist or you do not have the right to execute it, then an error message is printed.

<code>alias</code> - <i>Provides an alias for a command</i>	Command
---	---------

```
alias [-h] [<name> [<string>]]
```

The `alias` command, if given no arguments, will print the definition of all current aliases. Given a single argument, it will print the definition of that alias (if any). Given two arguments, the keyword `<name>` becomes an alias for the command string `<string>`, replacing any other alias with the same name.

Command Options:

<code>&lt;name&gt;</code>	Alias
<code>&lt;string&gt;</code>	Command string

It is possible to create aliases that take arguments by using the history substitution mechanism. To protect the history substitution character `%` from immediate expansion, it must be preceded by a `\` when entering the alias.

For example:

```

NuSMV> alias read "read_model -i %:1.smv ; set input_order_file
%:1.ord"
NuSMV> read short
will create an alias 'read', execute 'read_model -i short.smv; set input_order_file
short.ord'. And again:
NuSMV> alias echo2 "echo Hi ; echo %* !"
NuSMV> echo2 happy birthday
will print:
Hi
happy birthday !
CAVEAT: Currently there is no check to see if there is a circular dependency in
the alias definition. e.g.
NuSMV> alias foo "echo print_bdd_stats; foo"
creates an alias which refers to itself. Executing the command foo will result an
infinite loop during which the command print_bdd_stats will be executed.

```

**echo** - *Merely echoes the arguments*

Command

```
echo [-h] [-o filename [-a]] <string>
```

Echoes the specified string either to standard output, or to filename if the option `-o` is specified.

Command Options:

<code>-o filename</code>	Echoes to the specified filename instead of to standard output. If the option <code>-a</code> is not specified, the file <code>filename</code> will be overwritten if it already exists.
<code>-a</code>	Appends the output to the file specified by option <code>-o</code> , instead of overwriting it. Use only with the option <code>-o</code> .

**help** - *Provides on-line information on commands*

Command

```
help [-a] [-h] [<command>]
```

If invoked with no arguments `help` prints the list of all commands known to the command interpreter. If a command name is given, detailed information for that command will be provided.

Command Options:

<code>-a</code>	Provides a list of all internal commands, whose names begin with the underscore character ( <code>'_'</code> ) by convention.
-----------------	---

**history** - *list previous commands and their event numbers*

Command

```
history [-h] [<num>]
```

Lists previous commands and their event numbers. This is a UNIX-like history mechanism inside the NUSMV shell.

#### Command Options:

<num> Lists the last <num> events. Lists the last 30 events if <num> is not specified.

#### History Substitution:

The history substitution mechanism is a simpler version of the csh history substitution mechanism. It enables you to reuse words from previously typed commands.

The default history substitution character is the '%' ('!' is default for shell escapes, and '#' marks the beginning of a comment). This can be changed using the `set` command. In this description '%' is used as the `history_char`. The '%' can appear anywhere in a line. A line containing a history substitution is echoed to the screen after the substitution takes place. '%' can be preceded by a '\n' in order to escape the substitution, for example, to enter a '%' into an alias or to set the prompt.

Each valid line typed at the prompt is saved. If the `history` variable is set (see help page for `set`), each line is also echoed to the history file. You can use the `history` command to list the previously typed commands.

#### Substitutions:

At any point in a line these history substitutions are available.

#### Command Options:

%:0	Initial word of last command.
%:n	n-th argument of last command.
%\$	Last argument of last command.
%*	All but initial word of last command.
%%	Last command.
%stuf	Last command beginning with "stuf".
%n	Repeat the n-th command.
%-n	Repeat the n-th previous command.
^old^new	Replace "old" with "new" in previous command. Trailing spaces are significant during substitution. Initial spaces are not significant.

---

**print\_usage** - Prints processor and BDD statistics.

Command |

`print_usage [-h]`

Prints a formatted dump of processor-specific usage statistics, and BDD usage statistics. For Berkeley Unix, this includes all of the information in the `getrusage()` structure.

---

**quit** - exits NuSMV

Command |

`quit [-h] [-s]`

Stops the program. Does not save the current network before exiting.

Command Options:

`-s` Frees all the used memory before quitting. This is slower, and it is used for finding memory leaks.

---

**reset** - *Resets the whole system.* Command

```
reset [-h]
```

Resets the whole system, in order to read in another model and to perform verification on it.

---

**set** - *Sets an environment variable* Command

```
set [-h] [<name>] [<value>]
```

A variable environment is maintained by the command interpreter. The `set` command sets a variable to a particular value, and the `unset` command removes the definition of a variable. If `set` is given no arguments, it prints the current value of all variables.

Command Options:

`<name>` Variable name  
`<value>` Value to be assigned to the variable.

Interpolation of variables is allowed when using the `set` command. The variables are referred to with the prefix of `'$'`. So for example, what follows can be done to check the value of a set variable:

```
NuSMV> set foo bar
NuSMV> echo $foo
bar
```

The last line "bar" will be the output produced by NuSMV. Variables can be extended by using the character `':'` to concatenate values. For example:

```
NuSMV> set foo bar
NuSMV> set foo $foo:foobar
NuSMV> echo $foo
bar:foobar
```

The variable `foo` is extended with the value `foobar`. Whitespace characters may be present within quotes. However, variable interpolation lays the restriction that the characters `':'` and `'/'` may not be used within quotes. This is to allow for recursive interpolation. So for example, the following is allowed

```
NuSMV> set "foo bar" this
NuSMV> echo $"foo bar"
this
```

The last line will be the output produced by NuSMV.

But in the following, the value of the variable `foo/bar` will not be interpreted correctly:

```
NuSMV> set "foo/bar" this
NuSMV> echo $"foo/bar"
foo/bar
```

If a variable is not set by the `set` command, then the variable is returned unchanged. Different commands use environment information for different purposes. The command interpreter makes use of the following parameters:

**Command Options:**

<code>autoexec</code>	Defines a command string to be automatically executed after every command processed by the command interpreter. This is useful for things like timing commands, or tracing the progress of optimization.
<code>open_path</code>	“ <code>open_path</code> ” (in analogy to the shell-variable <code>PATH</code> ) is a list of colon-separated strings giving directories to be searched whenever a file is opened for read. Typically the current directory ( <code>.</code> ) is the first item in this list. The standard system library (typically <code>NuSMV_LIBRARY_PATH</code> ) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.
<code>nusmv_stderr</code>	Standard error (normally <code>( stderr)</code> ) can be re-directed to a file by setting the variable <code>nusmv_stderr</code> .
<code>nusmv_stdout</code>	Standard output (normally <code>( stdout)</code> ) can be re-directed to a file by setting the variable <code>nusmv_stdout</code> .

<b>source</b> - <i>Executes a sequence of commands from a file</i>	Command
--	---------

```
source [-h] [-p] [-s] [-x] <file> [<args>]
```

Reads and executes commands from a file.

**Command Options:**

<code>-p</code>	Prints a prompt before reading each command.
<code>-s</code>	Silently ignores an attempt to execute commands from a nonexistent file.
<code>-x</code>	Echoes each command before it is executed.
<code>&lt;file&gt;</code>	File name.

Arguments on the command line after the filename are remembered but not evaluated. Commands in the script file can then refer to these arguments using the history substitution mechanism. EXAMPLE:

Contents of `test.scr`:

```
read_model -i %:2
flatten_hierarchy
build_variables
build_model
compute_fairness
```

Typing `source test.scr short.smv` on the command line will execute the sequence

```
read_model -i short.smv
flatten_hierarchy
build_variables
build_model
compute_fairness
```

(In this case `%:0` gets `source`, `%:1` gets `test.scr`, and `%:2` gets `short.smv`.)  
If you type alias `st source test.scr` and then type `st short.smv bozo`, you will execute

```
read_model -i bozo
flatten_hierarchy
build_variables
build_model
compute_fairness
```

because `bozo` was the second argument on the last command line typed. In other words, command substitution in a script file depends on how the script file was invoked. Switches passed to a command are also counted as positional parameters. Therefore, if you type `st -x short.smv bozo`, you will execute

```
read_model -i short.smv
flatten_hierarchy
build_variables
build_model
compute_fairness
```

To pass the `-x` switch (or any other switch) to `source` when the script uses positional parameters, you may define an alias. For instance, alias `srcx source -x`.

See the variable `on_failure_script_quits` for further information.

<b>time</b> - <i>Provides a simple CPU elapsed time value</i>	Command
---	---------

```
time [-h]
```

Prints the processor time used since the last invocation of the `time` command, and the total processor time used since NUSMV was started.

---

**unalias** - *Removes the definition of an alias.* Command

`unalias [-h] <alias-names>`

Removes the definition of an alias specified via the `alias` command.

Command Options:

`<alias-names>`      Aliases to be removed

---

<b>unset</b> - <i>Unsets an environment variable</i>	Command
--	---------

---

`unset [-h] <variables>`

A variable environment is maintained by the command interpreter. The `set` command sets a variable to a particular value, and the `unset` command removes the definition of a variable.

Command Options:

`<variables>`            Variables to be unset.

---

<b>usage</b> - <i>Provides a dump of process statistics</i>	Command
---	---------

---

`usage [-h]`

Prints a formatted dump of processor-specific usage statistics. For Berkeley Unix, this includes all of the information in the `getrusage()` structure.

---

<b>which</b> - <i>Looks for a file called "file name"</i>	Command
---	---------

---

`which [-h] <file_name>`

Looks for a file in a set of directories which includes the current directory as well as those in the NUSMV path. If it finds the specified file, it reports the found file's path. The searching path is specified through the `set open_path` command in `.nusmvr.c`.

Command Options:

`<file_name>`            File to be searched

### 3.10 Other Environment Variables

The behavior of the system depends on the value of some environment variables. For instance, an environment variable specifies the partitioning method to be used in building the transition relation. The value of environment variables can be inspected and modified with the "set" command. Environment variables can be either logical or utility.

---

<b>autoexec</b>	Environment Variable
-----------------	----------------------

---

Defines a command string to be automatically executed after every command processed by the command interpreter. This may be useful for timing commands, or tracing the progress of optimization.

---

<b>on_failure_script_quits</b>	Environment Variable
--------------------------------	----------------------

---

When a non-fatal error occurs during the interactive mode, the interactive interpreter simply stops the currently executed command, prints the reason of the problem, and prompts for a new command. When set, this variables makes the command interpreter quit when an error occur, and then quit NUSMV. This behaviour might be useful when the command `source` is controlled by either a system pipe or a shell script. Under these conditions a mistake within the script interpreted by `source` or any unexpected error might hang the controlling script or pipe, as by default the interpreter would simply give up the current execution, and wait for further commands. The default value of this environment variable is 0.

<b>filec</b>	Environment Variable
--------------	----------------------

Enables file completion a la “csh”. If the system has been compiled with the “readline” library, the user is able to perform file completion by typing the <TAB> key (in a way similar to the file completion inside the “bash” shell). If the system has not been compiled with the “readline” library, a built-in method to perform file completion a la “csh” can be used. This method is enabled with the ‘set filec’ command. The “csh” file completion method can be also enabled if the “readline” library has been used. In this case the features offered by “readline” will be disabled.

<b>shell_char</b>	Environment Variable
-------------------	----------------------

`shell_char` specifies a character to be used as shell escape. The default value of this environment variable is ‘!’.

<b>history_char</b>	Environment Variable
---------------------	----------------------

`history_char` specifies a character to be used in history substitutions. The default value of this environment variable is ‘%’.

<b>open_path</b>	Environment Variable
------------------	----------------------

`open_path` (in analogy to the shell-variable `PATH`) is a list of colon-separated strings giving directories to be searched whenever a file is opened for read. Typically the current directory (.) is first in this list. The standard system library (`NUSMV_LIBRARY_PATH`) is always implicitly appended to the current path. This provides a convenient short-hand mechanism for reaching standard library files.

<b>nusmv_stderr</b>	Environment Variable
---------------------	----------------------

Standard error (normally `stderr`) can be re-directed to a file by setting the variable `nusmv_stderr`.

<b>nusmv_stdout</b>	Environment Variable
---------------------	----------------------

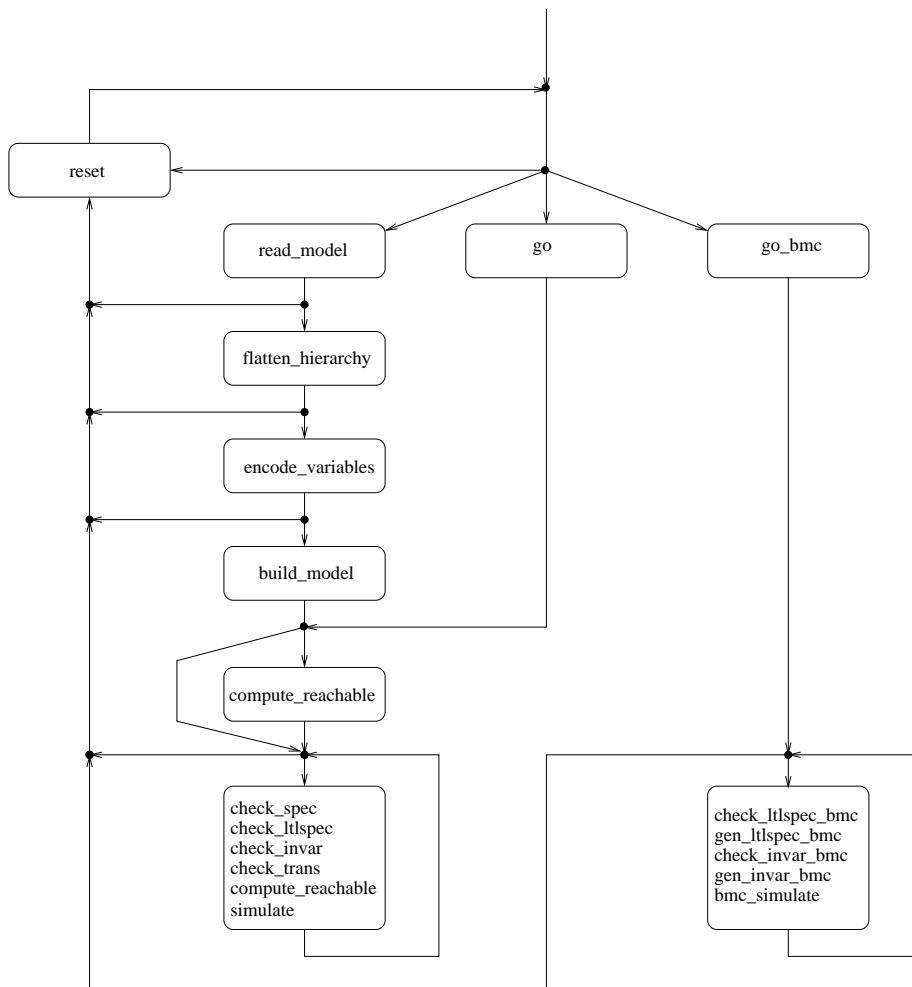


Figure 3.1: The dependency among NUSMV commands.

Standard output (normally `stdout`) can be re-directed to a file by setting the internal variable `nusmv_stdout`.

**nusmv\_stdin**

Environment Variable

Standard input (normally `stdin`) can be re-directed to a file by setting the internal variable `nusmv_stdin`.

## Chapter 4

# Running NuSMV batch

When the `-int` option is not specified, NUSMV runs as a batch program, in the style of SMV, performing (some of) the steps described in previous section in a fixed sequence.

```
system_prompt> NuSMV [command line options] input-file <RET>
```

The program described in *input-file* is processed, and the corresponding finite state machine is built. Then, if *input-file* contains formulas to verify, their truth in the specified structure is evaluated. For each formula which is not true a counterexample is printed. The batch mode can be controlled with the following command line options:

```
NuSMV [-h | -help] [-v vl] [-int] [-load script-file] [-s]
      [-cpp] [-pre pps] [-ofm fm-file] [-obm bm-file]
      [-lp] [-n idx] [-is] [-ic] [-ils] [-ips] [-ii]
      [-ctt] [-f] [-r] [-AG] [-coi]
      [-i iv-file] [-o ov-file] [-reorder] [-dynamic] [-m method]
      [[-mono]|[-thresh cp┐]|[-cp cp┐]|[-iwls95 cp┐]]
      [-noaffinity] [-iwls95preorder]
      [-bmc] [-bmc_length k]
      [input-file]
```

where the meaning of the options is described below. If *input-file* is not provided in batch mode, then the model is read from standard input.

`-help`

`-h`

Prints the command line help.

`-v verbose-level`

Enables printing of additional information on the internal operations of NUSMV. Setting *verbose-level* to 1 gives the basic information. Using this option makes you feel better, since otherwise the program prints nothing until it finishes, and there is no evidence that it is doing anything at all. Setting the *verbose-level* higher than 1 enables printing of much extra information.

-int	Starts interactive shell.
-load <i>cmd-file</i>	Starts the interactive shell and then executes NUSMV commands from file <i>cmd-file</i> . If an error occurs during a command execution, commands that follow will not be executed. See also the variable <code>on_failure_script_quits</code> .
-s	Avoids to load the NUSMV commands contained in <code>~/.nusmvr</code> or in <code>.nusmvr</code> or in <code>\$\$ {NuSMV_LIBRARY_PATH }/master.nusmvr</code> .
-cpp	Runs preprocessor on SMV files before any of those specified with the <code>-pre</code> option.
-pre <i>pps</i>	Specifies a list of pre-processors to run (in the order given) on the input file before it is parsed by NUSMV. Note that if the <code>-cpp</code> command is used, then the pre-processors specified by this command will be run after the input file has been pre-processed by that pre-processor. <i>pps</i> is either one single pre-processor name (with or without double quotes) or it is a space-separated list of pre-processor names contained within double quotes.
-ofm <i>fm-file</i>	prints flattened model to file <i>fm-file</i>
-obm <i>bm-file</i>	Prints boolean model to file <i>bm-file</i>
-lp	Lists all properties in SMV model
-n <i>idx</i>	Specifies which property of SMV model should be checked
-is	Does not check SPEC
-ic	Does not check COMPUTE
-ils	Does not check LTLSPEC
-ips	Does not check PLSPEC
-ii	Does not check INVARSPEC
-ctt	Checks whether the transition relation is total.
-f	Computes the set of reachable states before evaluating CTL expressions.
-r	Prints the number of reachable states before exiting. If the <code>-f</code> option is not used, the set of reachable states is computed.
-AG	Verifies only AG formulas using an ad hoc algorithm (see documentation for the <code>ag_only_search</code> environment variable).
-coi	Enables cone of influence reduction
-i <i>iv-file</i>	Reads the variable ordering from file <i>iv-file</i> .
-o <i>ov-file</i>	Reads the variable ordering from file <i>ov-file</i> .
-reorder	Enables variable reordering after having checked all the specification if any.
-dynamic	Enables dynamic reordering of variables

<code>-m <i>method</i></code>	Uses <i>method</i> when variable ordering is enabled. Possible values for <i>method</i> are those allowed for the <code>reorder_method</code> environment variable (see Section 3.8 [Interface to DD package], page 54).
<code>-mono</code>	Enables monolithic transition relation
<code>-thresh <i>cp_t</i></code>	conjunctive partitioning with threshold of each partition set to <i>cp_t</i> (DEFAULT, with <i>cp_t</i> =1000)
<code>-cp <i>cp_t</i></code>	DEPRECATED: use <code>thresh</code> instead.
<code>-iwls95 <i>cp_t</i></code>	Enables <i>Iwls95</i> conjunctive partitioning and sets the threshold of each partition to <i>cp_t</i>
<code>-noaffinity</code>	Disables affinity clustering
<code>-iwls95preoder</code>	Enables <i>Iwls95CP</i> preordering
<code>-bmc</code>	Enables BMC instead of BDD model checking (works only for LTL properties and PSL properties that can be translated into LTL)
<code>-bmc_length <i>k</i></code>	Sets <code>bmc_length</code> variable, used by BMC

# Bibliography

- [BCCZ99] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, In TACAS'99*, March 1999.
- [BCL<sup>+</sup>94] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.
- [CBM90] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *In J. Sifakis, editor, Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, volume 407 of LNCS, pages 365–373, Berlin, June 1990*.
- [CCG<sup>+</sup>02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of Computer Aided Verification (CAV 02)*, 2002.
- [CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. In *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4), March 2000.
- [CGH97] E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. In *Formal Methods in System Design*, 10(1):57–71, February 1997.
- [Dil88] D. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In *ACM Distinguished Dissertations. MIT Press*, 1988.
- [EMSS91] E. Allen Emerson, A. K. Mok, A. Prasad Sistla, and Jai Srinivasan. Quantitative temporal reasoning. In *Edmund M. Clarke and Robert P. Krushan, editors, Proceedings of Computer-Aided Verification (CAV'90), volume 531 of LNCS, pages 136-145, Berlin, Germany, June 1991*.
- [ES04] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. In Ofer Strichman and Armin Biere, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2004.

- [Mar85] A.J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *In H. Fuchs and W.H. Freeman, editors, Proceedings of the 1985 Chapel Hill Conference on VLSI, pages 245–260, New York, 1985.*
- [McM92] K.L. McMillan. The smv system – draft. In *Available at <http://www.cs.cmu.edu/modelcheck/smv/smvmanual.r2.2.ps>, 1992.*
- [McM93] K.L. McMillan. Symbolic model checking. In *Kluwer Academic Publ., 1993.*
- [MHS00] Moon, Hachtel, and Somenzi. Border-block triangular form and conjunction schedule in image computation. In *FMCAD, 2000.*
- [PSL] Language Front-End for Sugar Foundation Language. <http://www.haifa.il.ibm.com/projects/verification/sugar/parser.html>.
- [psl03] Accellera, Property Specification Language - Reference Manual - Version 1.01. [http://www.eda.org/vfv/docs/psl\\_lrm-1.01.pdf](http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf), April 2003.
- [RAP<sup>+</sup>95] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient bdd algorithms for fsm synthesis and verification. In *In IEEE/ACM Proceedings International Workshop on Logic Synthesis, Lake Tahoe (NV), May 1995.*
- [sfVS96] "VIS: A system for Verification and The VIS Group Synthesis". Proceedings of the 8th international conference on computer aided verification, p428-432. In *Springer Lecture Notes in Computer Science, 1102, Edited by R. Alur and T. Henzinger, New Brunswick, NJ, 1996.*
- [Som98] F. Somenzi. Cudd: Cu decision diagram package — release 2.2.0. In *Department of Electrical and Computer Engineering — University of Colorado at Boulder, May 1998.*

## Appendix A

# Compatibility with CMU SMV

The NUSMV language is mostly source compatible with the original version of SMV distributed at Carnegie Mellon University from which we started. In this appendix we describe the most common problems that can be encountered when trying to use old CMU SMV programs with NUSMV.

The main problem is variable names in old programs that conflicts with new reserved words. The list of the new reserved words of NUSMV w.r.t. CMU SMV is the following:

F, G, X, U, V, W, H, O, Y, Z, S, T, B	These names are reserved for the LTL temporal operators.
LTLSPEC	It is used to introduce LTL specifications.
INVARSPEC	It is used to introduce invariant specifications.
IVAR	It is used to introduce input variables.
JUSTICE	It is used to introduce ‘justice’ fairness constraints.
COMPASSION	It is used to introduce ‘compassion’ fairness constraints.

The IMPLEMENTS, INPUT, OUTPUT statements are not supported by NUSMV. They are parsed from the input file, but are internally ignored.

NUSMV differs from CMU SMV also in the controls that are performed on the input formulas. Several formulas that are valid for CMU SMV, but that have no clear semantics, are not accepted by NUSMV. In particular:

- It is no longer possible to write formulas containing nested ‘next’.

```
TRANS
  next(alpha & next(beta | next(gamma))) -> delta
```

- It is no longer possible to write formulas containing ‘next’ in the right hand side of ‘normal’ and ‘init’ assignments (they are allowed in the right hand side of ‘next’ assignments), and with the statements ‘INVAR’ and ‘INIT’.

```
INVAR
  next(alpha) & beta
INIT
  next(beta) -> alpha
```

```

ASSIGN
  delta := alpha & next(gamma);      -- normal assignments
  init(gamma) := alpha & next(delta); -- init assignments

```

- It is no longer possible to write ‘SPEC’, ‘FAIRNESS’ statements containing ‘next’.

```

FAIRNESS
  next(running)
SPEC
  next(x) & y

```

- The check for circular dependencies among variables has been done more restrictive. We say that variable  $x$  depends on variable  $y$  if  $x := f(y)$ . We say that there is a circular dependency in the definition of  $x$  if:

- $x$  depends on itself ( e.g.  $x := f(x,y)$  );
- $x$  depends on  $y$  and  $y$  depends on  $x$  (e.g.  $x := f(y)$  and  $y := f(x)$  or  $x := f(z)$ ,  $z := f(y)$  and  $y := f(x)$  ).

In the case of circular dependencies among variables there is no fixed order in which we can compute the involved variables. Avoiding circular dependencies among variables guarantee that there exists an order in which the variables can be computed. In NUSMV circular dependencies are not allowed.

In CMU SMV the test for circular dependencies is able to detect circular dependencies only in ‘normal’ assignments, and not in ‘next’ assignments. The circular dependencies check of NUSMV has been extended to detect circularities also in ‘next’ assignments. For instance the following fragment of code is accepted by CMU SMV but discarded by NUSMV.

```

MODULE main
VAR
  y : boolean;
  x : boolean;
ASSIGN
  next(x) := x & next(y);
  next(y) := y & next(x);

```

Another difference between NUSMV and CMU SMV is in the variable order file. The variable ordering file accepted by NUSMV can be partial and can contain variables not declared in the model. Variables listed in the ordering file but not declared in the model are simply discarded. The variables declared in the model but not listed in the variable file provided in input are created at the end of the given ordering following the default ordering. All the ordering files generated by CMU SMV are accepted in input from NUSMV but the ordering files generated by NUSMV may be not accepted by CMU SMV. Notice that there is no guarantee that a good ordering for CMU SMV is also a good ordering for NUSMV. In the ordering files for NUSMV, identifier `_process_selector_` can be used to control the position of the variable that encodes process selection. In CMU SMV it is not possible to control the position of this variable in the ordering; it is hard-coded at the top of the ordering.

# Command Index

!, *see bang* 58  
    , 58  
add\_property, 34  
alias, 58  
bmc\_setup, 35  
bmc\_simulate, 45  
build\_model, 28  
check\_fsm, 31  
check\_invar\_bmc\_inc, 43  
check\_invar\_bmc, 42  
check\_invar, 33  
check\_ltlspec\_bmc\_inc, 40  
check\_ltlspec\_bmc\_onepb, 37  
check\_ltlspec\_bmc, 36  
check\_ltlspec, 33  
check\_pslspec, 45  
check\_spec, 32  
compute\_reachable, 31  
compute, 34  
dynamic\_var\_ordering, 55  
echo, 59  
encode\_variables, 27  
flatten\_hierarchy, 26  
gen\_invar\_bmc, 43  
gen\_ltlspec\_bmc\_onepb, 39  
gen\_ltlspec\_bmc, 38  
go\_bmc, 35  
goto\_state, 49  
go, 30  
help, 59  
history, 59  
pick\_state, 47  
print\_bdd\_stats, 57  
print\_current\_state, 49  
print\_fair\_states, 31  
print\_fair\_transitions, 32  
print\_iwls95options, 30  
print\_reachable\_states, 31  
print\_usage, 60  
process\_model, 30  
quit, 60  
read\_model, 26  
read\_trace, 51  
reset, 61  
set\_bdd\_parameters, 57  
set, 61  
show\_plugins, 50  
show\_traces, 50  
show\_vars, 26  
simulate, 47  
source, 62  
time, 63  
unalias, 64  
unset, 65  
usage, 65  
which, 65  
write\_order, 27

# Variable Index

NuSMV\_LIBRARY\_PATH, 66, 69  
affinity, 29  
ag\_only\_search, 32  
autoexec, 65  
bmc\_dimacs\_filename, 41  
bmc\_inc\_invar\_alg, 44  
bmc\_invar\_alg, 44  
bmc\_invar\_dimacs\_filename, 44  
bmc\_length, 41  
bmc\_loopback, 41  
check\_fsm, 31  
conj\_part\_threshold, 29  
default\_trace\_plugin, 50  
enable\_reorder, 54  
filec, 66  
forward\_search, 33  
history\_char, 66  
image.W{1, 2, 3, 4}, 29  
image\_cluster\_size, 29  
image\_verbosity, 30  
input\_file, 26  
input\_order\_file, 27  
iwls95preorder, 30  
nusmv\_stderr, 66  
nusmv\_stdin, 67  
nusmv\_stdout, 67  
on\_failure\_script\_quits, 66  
open\_path, 66  
output\_order\_file, 28  
partition\_method, 29  
pp\_list, 26  
reorder\_method, 54  
sat\_solver, 45  
shell\_char, 66  
showed\_states, 47  
verbose\_level, 25  
write\_order\_dumps\_bits, 27

# Index

## Symbols

.nusmvr, 69  
-AG, 69  
-bmc, 70  
-bmc *k*, 70  
-coi, 69  
-cpp, 69  
-cp *cp\_t*, 70  
-ctt, 69  
-dynamic, 69  
-f, 69  
-help, 68  
-h, 68  
-ic, 69  
-ii, 69  
-ils, 69  
-int, 69  
-ips, 69  
-is, 69  
-iwls95preorder, 70  
-iwls95 *cp\_t*, 70  
-i *iv\_file*, 69  
-load *cmd\_file*, 69  
-lp, 69  
-mono, 70  
-m *method*, 70  
-noaffinity, 70  
-n *idx*, 69  
-obm *bm\_file*, 69  
-ofm *fm\_file*, 69  
-o *ov\_file*, 69  
-pre *pps*, 69  
-reorder, 69  
-r, 69  
-thresh *cp\_t*, 70  
-v *verbose-level*, 68  
FAIRNESS declarations, 14  
IVAR declaration, 9  
VAR declaration, 9  
running, 14

temp.ord, 28  
/.nusmvr, 69

## A

administration commands, 58  
Array Variables, 24

## B

Basic Trace Explainer, 52  
batch, running NUSMV, 68

## C

case expressions, 7  
Commands for Bounded Model Checking, 35  
Commands for checking PSL specifications, 45  
comments in NUSMV language, 6  
compassion constraints, 14  
CTL Specifications, 15

## D

DD package interface, 54  
DEFINE declarations, 11  
Displaying Traces, 50

## E

expressions, 6

## F

fair execution paths, 14  
fairness constraints, 14  
fairness constraints declaration, 14  
fair paths, 14

## I

Identifiers, 13  
infinity, 18  
INIT declaration, 10  
Input File Syntax, 22  
input variables syntax, 9

Inspecting Traces, 49  
interactive, running NUSMV, 25  
interactive shell, 25  
interface to DD Package, 54  
INVAR declaration, 11  
ISA declarations, 11

## **J**

justice constraints, 14

## **L**

LTL Specifications, 16

## **M**

main module, 14  
master.nusmvr, 69  
model compiling, 26  
model parsing, 26  
model reading, 26  
MODULE declarations, 12

## **N**

Next Expressions, 8  
next expressions, 8

## **O**

options, 68

## **P**

process, 14  
processes, 14  
process keyword, 14  
PSL Specifications, 18

## **R**

Real Time CTL Specifications and Computations, 17

## **S**

Scalar Variables, 23  
self, 13  
set expressions, 7  
Shell configuration Variables, 65  
simple expressions, 6  
Simulation Commands, 47  
States/Variables Table, 52  
state variables syntax, 9

## **T**

Trace Plugin Commands, 50  
Trace Plugins, 51

Traces, 49  
TRANS declarations, 10  
type declaration, 9  
type specifiers, 9

## **V**

var\_id, 7

## **X**

XML Format Printer, 53  
XML Format Reader, 53