NuSMV: a new symbolic model checker

A. Cimatti[1] E. Clarke[2] F. Giunchiglia[1] M. Roveri[1,3]

[1]ITC-IRST, Via Sommarive 18, 38055 Povo, Trento, Italy

{cimatti,fausto,roveri}@irst.itc.it

[2]SCS, Carnegie-Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213-3891, USA

Edmund.Clarke@cs.cmu.edu

[3]DSI, University of Milano, Via Comelico 39, 20135 Milano, Italy

Abstract

This paper describes a new symbolic model checker, called NuSMV, developed as part of a joint project between CMU and IRST. NuSMV is the result of the reengineering, reimplementation, and, to a limited extent, extension of the CMU SMV model checker. The core of this paper consists of a detailed description of the NuSMV functionalities, architecture, and implementation.

Key words Symbolic Model Checking – Temporal Logics – Automatic verification – Tools for technology transfer.

1 Introduction

This paper describes the results of a joint project between Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientifica e Tecnologica (IRST) whose goal is the development of a new symbolic model checker.[1] The new model checker, called NuSMV, is designed to be a well structured, open, flexible and documented platform for model checking. To be usable in technology transfer projects, NuSMV was designed to be very robust, easy to modify, and close to the standards required by industry.

NuSMV is the result of the reengineering and reimplementation of the CMU SMV [47, 26] symbolic model checker. With respect to CMU SMV, NuSMV has been upgraded along three dimensions.

- From the point of view of the system functionalities, NuSMV has some features (e.g., multiple interfaces, LTL specifications) that enhance the user ability to interact with the system, and provide more heuristics for, e.g., achieving efficiency or partially controlling the state explosion.
- The system architecture of NuSMV is highly modular (thus allowing for the substitution or elimination of certain modules) and open (thus allowing for the addition of new modules). A further feature is that in NuSMV the user can control, and possibly change, the order of execution of some system modules.
- The quality of the implementation is much enhanced. NuSMV is a very robust and well documented system, whose code is (relatively) easy to modify.

The paper is organized as follows: in Section 2 we briefly introduce the logical framework below symbolic model checking; Section 3 describes the interaction with the system; Section 4 explains the functionalities provided by the system; Section 5 describes the NuSMV system architecture; Section 6 describes the NuSMV implementation features. Finally, Section 7 describes the results of some tests and future development directions.

NuSMV is available at the url "http://sra.itc.it/tools/nusmv".

[1]The material presented in this paper is self-contained. This paper is readable with ordinary efforts for someone with some background in formal methods.

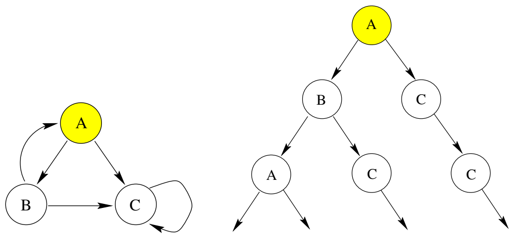

Figure 1: A State Transition Graph and its unwinding.

2 Symbolic Model Checking

The most widely used verification techniques are testing and simulation. In the case of complex, asynchronous systems, however, these techniques can cover only a limited portion of possible behaviors. A complementary verification technique is Temporal Logic Model Checking [23, 28, 51]. In this approach, the verified system is modeled as a finite state transition system, and the specifications are expressed in a propositional temporal logic. Then, by exhaustively exploring the state space of the state transition system, it is possible to check automatically if the specifications are satisfied. The termination of model checking is guaranteed by the finiteness of the model. One of the most important features of model checking is that, when a specification is found not to hold, a counterexample (i.e., a witness of the offending behavior of the system) is produced.

2.1 Temporal Logic

A finite state system can be described as a tuple:

$$\mathcal{M} =< \mathcal{S}, \mathcal{I}, \mathcal{R}, \mathcal{L} >$$

where $\mathcal{S}$ is a finite set of states, $\mathcal{I} \subseteq \mathcal{S}$ is the set of initial states, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation, specifying the possible transitions from state to state. $\mathcal{L}$ is a function that labels states with the *atomic propositions* from a given language. Such a tuple is called *state transition graph* or *Kripke structure* [44].

Temporal logics are used to predicate over the behavior defined by Kripke structures. A behavior in a Kripke structure is obtained starting from a state $s \in \mathcal{I}$, and then repeatedly appending states reachable through $\mathcal{R}$. We require that the transition relation $\mathcal{R}$ be total.[2] As a consequence all the behaviors of the system are infinite. Since a state can have more than one successor, the structure can be thought of as unwinding into an infinite tree, representing all the possible executions of the system starting from the initial states. Figure 1 shows a state transition graph and its unwinding from the state labeled with "`A`".

Two useful temporal logics are *Computation Tree Logic* (called CTL) and *Linear Temporal Logic* (called LTL). They differ in how they handle branching in the underlying computation tree. In CTL temporal operators it is possible to quantify over the paths departing from a given state. In LTL operators are intended to describe properties of all possible computation paths.

The syntax of CTL formulas is given by the following rules:

1. any atomic proposition is a CTL formula;
2. if α and β are CTL formulas, then $\alpha \bullet \beta$ and $\neg\alpha$ are CTL formulas, where $\bullet$ is any boolean connective $(\wedge, \vee, \ldots)$;
3. if α and β are CTL formulas, then $\mathbf{EX}\alpha$, $\mathbf{EG}\alpha$, $\mathbf{E}[\alpha\mathbf{U}\beta]$ are CTL formulas.

Figure 2 describes the intuitive meaning of CTL formulas. $\mathbf{EX}\alpha$ means that there exists $(\mathbf{E})$ a path starting from a state $s_0 \in \mathcal{S}$ in which in the next $(\mathbf{X})$ state α holds. $\mathbf{EG}\alpha$ means that there exists a path starting from a state s_0 in which globally $(\mathbf{G})$ α holds. $\mathbf{E}[\alpha\mathbf{U}\beta]$ there exists a path starting from a state s_0 in which α holds until $(\mathbf{U})$ β holds. The other CTL operators (e.g., $\mathbf{AF}\alpha$, meaning for all paths eventually α) can be derived from these three according to the rules listed in Table 1.

$\mathbf{AX}\alpha = \neg\mathbf{EX}(\neg\alpha)$	For all paths, in the next state α
$\mathbf{EF}\alpha = \mathbf{E}[\top\mathbf{U}\alpha]$	There exists a path in which eventually α
$\mathbf{AG}\alpha = \neg\mathbf{EF}(\neg\alpha)$	Invariantly α
$\mathbf{A}[\alpha\mathbf{U}\beta] = \neg\mathbf{E}[\neg\beta\mathbf{U}\neg\alpha \wedge \neg\beta] \wedge \neg\mathbf{EG}\neg\beta$	For all paths, α until β
$\mathbf{AF}\alpha = \mathbf{A}[\top\mathbf{U}\alpha]$	For all paths, eventually α

Table 1: Definition of CTL operators.

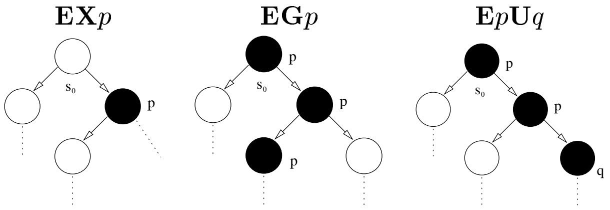

Figure 2: The meaning of temporal modalities.

The syntax of LTL formulas is the following:

1. any atomic proposition is an LTL formula;
2. if α and β are LTL formulas, then $\alpha \bullet \beta$ and $\neg\alpha$ are LTL formulas, where $\bullet$ is any boolean connective $(\wedge, \vee, \ldots)$;
3. if α and β are LTL formulas, then $\mathbf{X}\alpha$, $\mathbf{G}\alpha$, $[\alpha\mathbf{U}\beta]$ are LTL formulas.

The model checking problem can be formally formulated as follows. Given a Kripke structure $\mathcal{M}$ and a temporal logic formula φ, find the set of all states that satisfy φ, namely the set of states

$$\{s \in \mathcal{S} \mid \mathcal{M}, s \models \varphi\} \tag{1}$$

We say that the system satisfies the specification provided that *all the initial states* are in the set $\{s \in \mathcal{S} \mid \mathcal{M}, s \models \varphi\}$:

$$\mathcal{I} \subseteq \{s \in \mathcal{S} \mid \mathcal{M}, s \models \varphi\} \tag{2}$$

Each CTL formula φ is identified by the set of states where it holds, $\{s \in \mathcal{S} | \mathcal{M}, s \models \varphi\}$. CTL operators may be characterized as a least or a greatest fixpoint of an appropriate predicate transformer [12]:

- $\mathbf{EG}\alpha = \mathbf{gfp}_Z[\alpha \wedge \mathbf{EX}Z]$
- $\mathbf{E}[\alpha\mathbf{U}\beta] = \mathbf{lfp}_Z[\beta \vee (\alpha \wedge \mathbf{EX}Z)]$

This suggests an algorithm for symbolic model checking which is driven by the structure of the formula. For example, the computation of $\mathbf{E}[\alpha\mathbf{U}\beta] = \mathbf{lfp}_Z[\beta \vee (\alpha \wedge \mathbf{EX}Z)]$ corresponds to the following iterations:

$$\begin{aligned}
&s_0 = \bot \\
&s_1 = \beta \vee (\alpha \wedge \mathbf{EX}\overbrace{\bot}^{s_0}) = \beta \\
&s_2 = \beta \vee (\alpha \wedge \mathbf{EX}\overbrace{\beta}^{s_1}) \\
&s_3 = \beta \vee (\alpha \wedge \mathbf{EX}\overbrace{(\beta \vee (\alpha \wedge \mathbf{EX}\beta))}^{s_2}) \\
&\ldots
\end{aligned}$$

Iterations continue until $s_{i+1} = s_i$. The existence of least and greatest fixpoints is guaranteed by the monotonicity of the predicate transformers and by the finiteness of the domain [12].

[2] A transition relation $\mathcal{R} \subseteq S \times S$ is total if and only if for each state $s \in \mathcal{S}$ there exists a state $s' \in \mathcal{S}$ such that $(s, s') \in \mathcal{R}$.

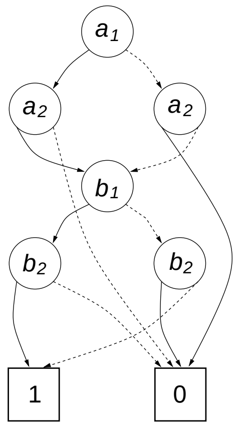

Figure 3: A BDD for the formula "(a1 iff a2) and (b1 iff b2)".

2.2 Symbolic Representation of Kripke Structures

The first model checking algorithms used an explicit representation of the Kripke structure as a labeled, directed graph [23, 28, 51].[3] A major improvement was achieved with the use of symbolic representations [29, 49, 59, 12, 47], based on the use of Ordered Binary Decision Diagrams [7, 8] (BDDs for short). BDDs are a representation for boolean formulas, which is canonical once an order on the variables has been established. Figure 3 depicts the BDD for the boolean formula $(a_1 \leftrightarrow a_2) \wedge (b_1 \leftrightarrow b_2)$, using the variable ordering a_1, a_2, b_1, b_2. Solid lines represent "then" arcs (the corresponding variable has to be considered positive), dashed lines represent "else" arcs (the corresponding variable has to be considered negative). Paths from the root to the node labeled with "1" represent the satisfying assignments of the represented boolean formula (e.g., $a_1 \leftarrow 1, a_2 \leftarrow 1, b_1 \leftarrow 0, b_2 \leftarrow 0$). Intuitively, a state of the system is symbolically represented by an assignment of boolean values to the set of state variables.[4] A boolean formula (and thus its BDD) is a compact representation of the set of the states represented by the assignments which make the formula true. Similarly, the transition relation can be expressed as a boolean formula in two sets of variables, one relative to the current state and the other relative to the next state.

This makes it possible to represent predicate transformers and fixpoints as BDDs. The basic boolean operations are handled by means of standard algorithms for computing boolean connectives with BDDs [7, 8], and fix-point algorithms can be easily implemented in terms of basic BDD operations [12, 47].

The basic computation step in the previously described predicate transformers is the computation of the states satisfying $\mathbf{EX}\alpha$. Let $\alpha(\underline{x})$ be a boolean formula on the boolean variables $\underline{x}$, where $\underline{x}$ is a vector of boolean state variables, representing the set of states satisfying α. The set of states satisfying $\mathbf{EX}\alpha$ can be computed as:

$$(\mathbf{EX}\alpha)(\underline{x}) = \exists \underline{x}'.(\alpha(\underline{x})[\underline{x}'/\underline{x}] \wedge \mathcal{R}(\underline{x}, \underline{x}'))$$

Where with $\alpha(\underline{x})[\underline{x}'/\underline{x}]$ we mean the simultaneous substitution in $\alpha(\underline{x})$ of variables $\underline{x}$ with the corresponding variables $\underline{x}'$. This operation, called *Relational Product* [47], is performed as an atomic operation on BDDs.

The set $\{s \in \mathcal{S} \mid \mathcal{M}, s \models \varphi\}$ can be computed using basic BDDs operations plus fixpoint characterization of the CTL operators. Formally, the check which determines whether the model verifies a property φ (see Equation 2) reduces to the following check between BDDs:

$$\neg\varphi \wedge \mathcal{I} = \bot$$

2.3 Fields of Application

The use of BDDs (i.e. the implicit representation of the transition system) makes it possible to verify very large systems (larger than 10^{20} states [12, 47, 11]). Symbolic model checking has been successful in various fields, allowing the discovery of design bugs that were very difficult to highlight with traditional techniques. Some of the most significant results are given below (an extensive discussion of the applications of symbolic model checking can be found in [27]):

[3] This is almost true. However, the very first implementation of the CESAR tool was a symbolic model checker. See [52] for more details.

[4] For non boolean state variables a boolean encoding can be performed.

- in [22] the authors applied this technique to verify the cache coherence protocol described in the IEEE Futurebus+ Standard 896.1.1991. They found a number of previously undetected and potential errors in the design of the protocol.
- in [47, 46] the authors verified the cache consistency protocol developed at Encore Compute Corporation for their Gigamax distributed multiprocessor.
- in [31] the cache coherence protocol of the Scalable Coherent Interface, IEEE Standard 1596-1992 was verified, and several errors were found.
- in [1] the authors verified part of a preliminary version of the system requirement specifications of TCAS II (Traffic Alert and Collision Avoidance System II). TCAS II is an aircraft collision avoidance system required on most commercial aircraft in United States.
- in [18, 39] model checking techniques are used in the verification of a railway interlocking system.

3 NuSMV: Look and Feel

3.1 The SMV specification language

The current NuSMV input language [19] is essentially the same as the CMU SMV input language [47, 26]. The NuSMV input language is designed to allow for the description of finite state systems. The only data types provided by the language are booleans, bounded integer subranges, and symbolic enumerated types. Moreover, NuSMV allows for the definitions of bounded arrays of basic data types.

The description of a complex system can be decomposed into modules, and each of them can be instantiated many times. This provides the user with a modular and hierarchical description, and supports the definition of reusable components. Each module defines a finite state machine. Modules can be composed either synchronously or asynchronously using interleaving. In synchronous composition a single step in the composition corresponds to a single step in each of the components. In asynchronous composition with interleaving a single step of the composition corresponds to a single step performed by exactly one component. The NuSMV input language allows to describe deterministic and non deterministic systems.

A NuSMV program can describe both the model and the specification. Figure 4 gives a small example of a NuSMV program. The example in Figure 4 is a model of a 3 bit binary counter circuit. It illustrates the definition of reusable modules and expressions.

The module `counter_cell` is instantiated three times, with names `bit0`, `bit1` and `bit2`. The module `counter_cell` has a formal parameter `carry_in`. In the instantiation of the module, actual signals (`1` for the instance `bit0`, `bit0.carry_out` for the instance `bit1` and `bit1.carry_out` for the instance `bit2`) are plugged in for the formal parameters, thus linking the module instance to the program (a module can be seen as a subroutine). The property that we want to check is "invariantly eventually the counter count till 8" which is expressed in CTL using the design state variables as "`AG AF bit2.carry_out`".

It is also possible to specify the transition relation and the set of initial states of a module by means of propositional formulas, using the keywords `TRANS`, and `INIT` respectively. This provides the user with a lot of freedom in designing systems. In Figure 5 there is an equivalent definition of module `counter_cell` using propositional formulas.

With respect to CMU SMV, the input language of NuSMV has been extended to allow for the specification of properties expressed in LTL and the specifications of invariants [19].

3.2 The interaction with NuSMV

NuSMV has both a batch and an interactive mode. The batch mode offers a built-in method for dealing with the system in which computations are activated according to a fixed predefined algorithm. The batch mode provides an interaction with the system that is essentially the same provided by CMU SMV.
In the interactive mode computation steps are activated by commands executed by a command line interpreter, thus allowing for the modification of the predefined model checking algorithm. The definition of the interactive shell required the decomposition of the model checking algorithm into small basic computation steps (e.g., parsing, model

```
MODULE main
  VAR
    bit0 :  counter_cell(1);
    bit1 :  counter_cell(bit0.carry_out);
    bit2 :  counter_cell(bit1.carry_out);
  SPEC
    AG AF bit2.carry_out

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

Figure 4: A simple NuSMV program.

```
MODULE counter_cell(carry_in)
  VAR
    value :  boolean;
  INIT
    value = 0
  TRANS
    next(value) <-> ((!value & carry_in) | ( value & !carry_in))
  DEFINE
    carry_out := value & carry_in;
```

Figure 5: An equivalent definition of module `counter_cell` of the example described in Figure 4.

construction, reachability analysis, model checking). In the current version of the system, each of them corresponds to a command that implements a different functionality.

The three modes of interaction with the NUSMV system, namely, the interactive shell, the batch mode and the graphical user interface are described below.

3.2.1 The NUSMV interactive shell

In this mode the system 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. These steps include the construction of the model in different ways and the model checking of specifications.

The NUSMV interactive shell allows for the full configuration of the BDD options. For instance, several automatic variable ordering methods and cache configuration mechanisms can be suitably tuned according to the application.

Moreover, the NUSMV interactive shell provides the user with a script language that makes it possible to define different model checking algorithms that can be invoked as model checking tactics. These algorithms are provided via parameterized scripts.

The interactive shell of NUSMV is activated from the system prompt as follows ("NuSMV>" is the default prompt that NUSMV prints out to indicate that it is ready to evaluate commands):

```
system_prompt> NuSMV -int
NuSMV release 1.0 (compiled 11-Nov-98 at 8:28 PM)
NuSMV >
```

In Figure 6 we can see a typical interaction with NUSMV through the interactive shell.[5] The file "*counter.smv*" contains the example depicted in Figure 4. The verification process in NUSMV can be decomposed into the following sequence of basic steps.

1. The first step towards verification is reading the model. This is performed by typing the NUSMV command `read_model`. This command takes as argument a file name containing the model specification. After the correct execution of this command an internal representation of the read file is built and stored.

2. The second step consists in transforming the hierarchical description into a flattened description. This is performed by typing the command `flatten_hierarchy`. After the correct execution of this command all the information needed to build the automaton is separated out (e.g., the variable names, their range, a symbolic representation of the initial states and of the transition relation of the flattened system).

3. The third step consists in the encoding of the scalar variables into boolean variables. The NUSMV command `build_variables` is responsible for this task. It takes an optional argument specifying a file containing the ordering in which the variables have to be built.[6] As a default, the order of variables used by NUSMV corresponds to the order in which the variables appear in a depth first traversal of the hierarchy [19].

4. The fourth step consists of the compilation of the structures built by the `flatten_hierarchy` into BDDs using the encoding performed by `build_variables`. During this step some checks are performed that avoid circular dependencies in assignments and multiple definitions [19]. The NUSMV command that implements this step is `build_model`. This command has an option specifying the partitioning method to be used, i.e. monolithic, conjunctive, or disjunctive (see Section 4). It is possible to build first the transition relation in different formats and then choose the most appropriate to the specific problem.

5. The fifth step deals with the fairness constraints [12]. This command can be executed only when all the previous commands have been executed. This is due to the fact that fairness constraints are CTL formulas whose evaluation uses a previously computed transition relation.

After these five steps, the internal representation of the automaton is available to perform model checking.

[5] The output was slightly modified to make it more readable.

[6] This file can be generated by hand by the user or automatically by activating the automatic variable ordering to reduce the size of the BDDs and saving the results with the `write_order` command.

```
    system_prompt > NuSMV -int
    NuSMV release 1.0 (compiled 11-Nov-98 at 8:28 PM)
1)  NuSMV > read_model -i counter.smv
2)  NuSMV > flatten_hierarchy
3)  NuSMV > build_variables
    Building variables.....
    bit0.value:
       BDD variable 1, next BDD variable 2
    bit1.value:
       BDD variable 3, next BDD variable 4
    bit2.value:
       BDD variable 5, next BDD variable 6
4)  NuSMV > build_model
    checking for multiple assignments...
    checking for circular assignments...
    evaluating INIT statements...
    evaluating init() assignments...
       evaluating value:
       size of value = 3 ADD nodes
       evaluating value:
       size of value = 3 ADD nodes
       evaluating value:
       size of value = 3 ADD nodes
    size of global initial set = 1 states, 5 ADD nodes
    evaluating TRANS statements...
    evaluating next() assignments...
       evaluating next(value):
       size of next(value) = 5 ADD nodes
       evaluating next(value):
          evaluating bit0.carry_out:
          size of bit0.carry_out = 3 ADD nodes
       size of next(value) = 7 ADD nodes
       evaluating next(value):
          evaluating bit1.carry_out:
          size of bit1.carry_out = 4 ADD nodes
       size of next(value) = 8 ADD nodes
    size of transition relation = 14 BDD nodes
    evaluating normal assignments...
    evaluating INVAR statements...
    size of invariant set = 8 states, 1 BDD nodes
    The model has been built from file counter.smv.
5)  NuSMV > compute_fairness
6)  NuSMV > compute_reachable
    Starting computation of reachable states....
    iteration 0:  BDD size = 4, frontier size = 4, states = 1
    new frontier computed, size = 4
    ......
    iteration 7:  BDD size = 1, frontier size = 4, states = 8
    new frontier computed, size = 1
    done.
7)  NuSMV > check_spec
    -- specification AG AF bit2.carry_out is true
7)  NuSMV > check_spec AG AX bit2.carry_out
    -- specification AG AX bit2.carry_out is false
    -- as demonstrated by the following execution sequence
    state 3.1:
    bit0.carry_out = 0
    bit0.value = 0
    bit1.carry_out = 0
    bit1.value = 0
    bit2.carry_out = 0
    bit2.value = 0

    state 3.2:
    bit0.carry_out = 1
    bit0.value = 1
    NuSMV >
```


Figure 6: A typical interaction with NuSMV via the interactive shell. NuSMV commands are in boldface.

6. It is possible to compute the set of reachable states, via the `compute_reachable` command. This allows the user to simplify the evaluation of all the specifications by restricting the search only to reachable states.

7. It is possible to start the verification of CTL specifications (via the `check_spec` command), of LTL specifications (via the `check_ltlspec` command), and of invariants (via the `check_invar` command). All these commands, if called with no arguments, apply to the specifications of the respective type, if any, listed in the source file. Optionally a CTL, LTL or propositional formula can be given as an argument. In the transcript of Figure 6 first we check the property `AG AF bit2.carry_out` (the execution of `check_spec` without arguments, thus verifying the properties listed in the input file). Then we check the property `AG AX bit2.carry_out` (invariantly in the next state `bit2.carry_out` is `1`). This property appear to be false and a counterexample exploiting why the formula is not verified is computed and printed out.

3.2.2 The NuSMV batch mode

In this modality the system behaves mostly like the original CMU SMV. It performs some of the steps described previously in a fixed sequence. The sequence can be modified via command line options that enable different computations at fixed positions.

In the batch mode the system executes the first five steps previously described. The computation of the reachable states can be enabled before the computation of fairness constraints via the "`-f`" command line options. After the first five steps, the system looks for specifications in the source file, following the verification of CTL formulas, quantitative characteristics, LTL formulas and finally the invariants, if any. If the reordering of variables has been enabled via the `-reorder` command line option at the end of the computation, the reordering of variables starts and the generated ordering is saved into a file.

3.2.3 The NuSMV graphical user interface

On top of the interactive shell a graphical user interface (GUI from now on) was developed. The GUI is a separate process and communicates with NuSMV via the interactive shell.
From the GUI it is possible to edit and to modify the file containing the model description. The editor window provides the user with basic editing functionalities, such as copy and paste of blocks of text. Figure 7 gives a snapshot of the editor window. The NuSMV GUI allows for the modification of the options in a menu driven way. A snapshot of the window for setting system options is the small window in Figure 9. Moreover, the GUI offers a formula editor that helps the user in writing new specifications (see Figure 8). Depending on the kind of formula to verify, various buttons corresponding to temporal modalities and/or boolean connectives are activated. For instance in Figure 8 a CTL specification was chosen (the radio-button relative to CTL formulas is selected), and all the boolean connectives and the CTL operator buttons are active.

4 System Functionalities

The functionalities provided by NuSMV are described below. They are grouped in standard, advanced and user functionalities.

4.1 Standard Functionalities

NuSMV offers all the standard model checking functionalities. A brief summary is reported below.

Encoding. In general it is very useful to represent systems using enumerated types. As symbolic algorithms work on boolean variables, there is the need to encode propositions about non-boolean variables in the space of boolean variables. There are different ways to perform this task. At the moment NuSMV provides the encoding used in CMU SMV, which represents a finite range variable with a sequence of boolean variables. Consider a state variable `st` that can assume the values $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$. The propositions on the possible values of `st` can be encoded with three boolean variables $\mathtt{st}_0$, $\mathtt{st}_1$ and $\mathtt{st}_2$, as shown in Table 2.

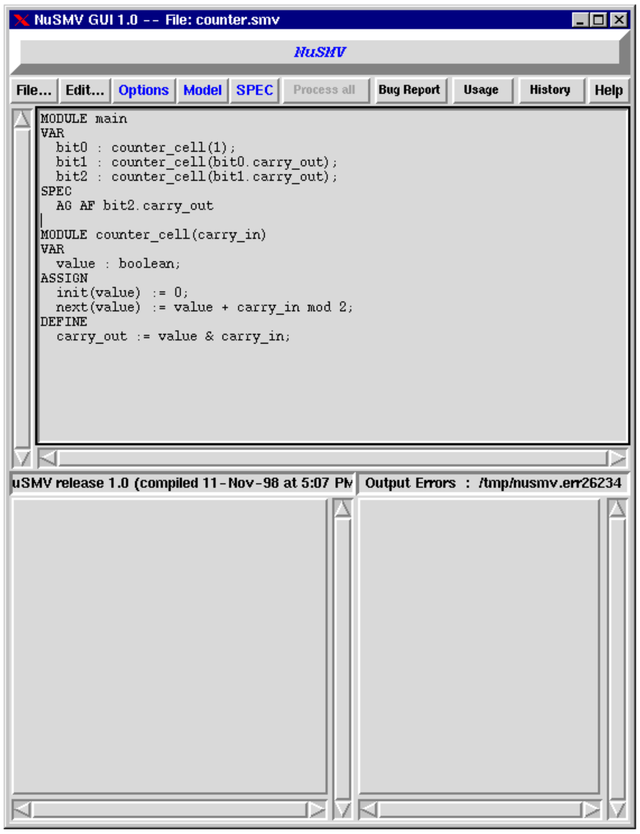

Figure 7: The NuSMV GUI editor window.

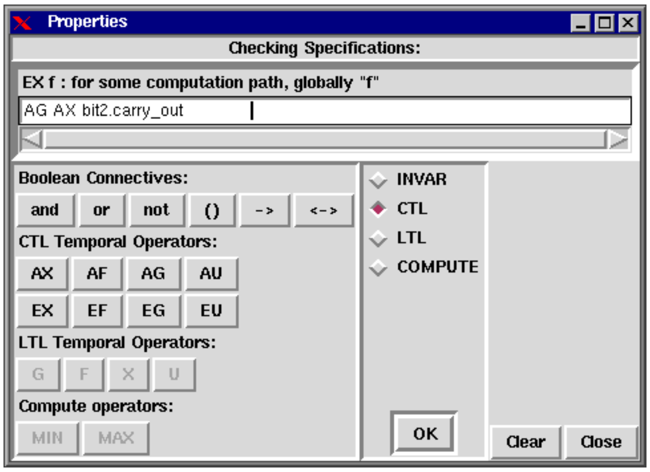

Figure 8: The NuSMV GUI formula editor window.

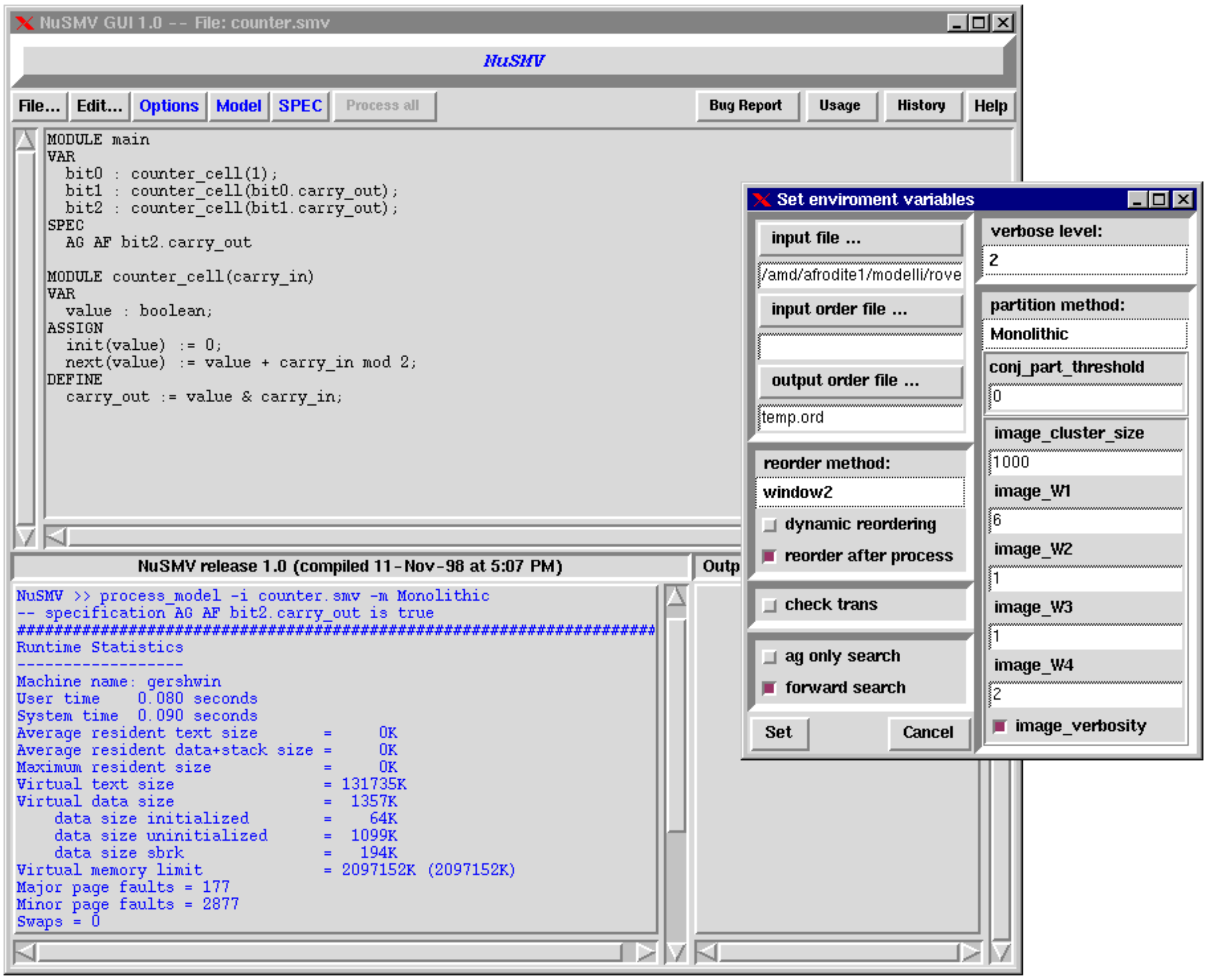

Figure 9: The NuSMV GUI and the options setting window.

Symbolic Value	Boolean Encoding
$\mathtt{st} = \mathtt{a}$	$\mathtt{st}_0 = 0 \wedge \mathtt{st}_1 = 0 \wedge \mathtt{st}_2 = 0$
$\mathtt{st} = \mathtt{b}$	$\mathtt{st}_0 = 1 \wedge \mathtt{st}_1 = 0$
$\mathtt{st} = \mathtt{c}$	$\mathtt{st}_0 = 0 \wedge \mathtt{st}_1 = 1$
$\mathtt{st} = \mathtt{d}$	$\mathtt{st}_0 = 1 \wedge \mathtt{st}_1 = 1$
$\mathtt{st} = \mathtt{e}$	$\mathtt{st}_0 = 0 \wedge \mathtt{st}_1 = 0 \wedge \mathtt{st}_2 = 1$

Table 2: The encoding of a variable “st” that can assume the values “a”, “b”, “c”, “d” and “e”.

Notice that the value assigned to the boolean variable $\mathtt{st}_2$, in the case of $\mathtt{st} \in \{\mathtt{b}, \mathtt{c}, \mathtt{d}\}$, is indifferent, and it can be either 0 or 1.[7] The boolean variables used to encode a scalar variable are grouped together, and are considered as a single block of variables in the BDD package. Other encoding policies can be more appropriate for variables of different kind. For example, word level encodings are useful for the analysis of data paths [25]. Moreover, the constraint of grouping the boolean variables together can be relaxed. Some of these policies will be codified in further releases of NUSMV.

Reachability Analysis. NUSMV, (similarly to CMU SMV), offers an enhanced algorithm for reachability analysis. The standard way to perform the computation of the reachable states $\mathbf{Reachable}(\mathcal{I})$, starting from the initial states $\mathcal{I}$, is:

$$\mathbf{Reachable}(\mathcal{I}) = R_k(\underline{x})$$

where k is the minimum integer such that $R_k(\underline{x}) = R_{k+1}(\underline{x})$, and $R_k(\underline{x})$ is recursively defined as:

$$\begin{array}{lcl} R_0(\underline{x}) & = & \mathcal{I} \\ R_{k+1}(\underline{x}) & = & (\exists \underline{x}.(R_k(\underline{x}) \wedge T(\underline{x}, \underline{x}')))[\underline{x}/\underline{x}'] \cup R_k(\underline{x}) \end{array}$$

$\underline{x}$ is the vector of state variables, $T(\cdot,\cdot)$ is the transition relation and $R_k(\underline{x})$ is the set of states reached in k or fewer steps.
The computation of the states reachable in $k+1$ steps can be performed by considering only the frontier set $F_k(\underline{x}) = R_k(\underline{x}) \wedge \overline{R_{k-1}(\underline{x})}$[8]:

$$R_{k+1}(\underline{x}) \quad = \quad (\exists \underline{x}.(F_k(\underline{x}) \wedge T(\underline{x}, \underline{x}')))[\underline{x}/\underline{x}'] \cup R_k(\underline{x})$$

We are planning to introduce new optimizations to the computation of the set of reachable states, such as the use of don't care sets. The set of reachable states can be used to simplify the following model checking computations.

Fair CTL model checking. NUSMV, similarly to CMU SMV, uses the algorithms presented in [12] as the basis for fair CTL model checking. These algorithms are essentially based on a fix-point characterization of temporal modalities and they work quite well in practice. If the set of reachable states has already been computed, the set of reachable states is used in the evaluation of CTL formulas. This allows us to restrict the search to reachable states only.

Check for Transition Relation Totality. In NUSMV it is possible to check the totality of the transition relation. The check is performed by means of basic model checking operations:

$$T(\cdot,\cdot) \text{ is total if and only if } \overline{\mathbf{EX}(\top)} = \bot$$

This functionality is very useful if the keywords TRANS or INVAR[9] are used to specify the transition relation. In these cases it is in fact possible to introduce deadlock states.

Counterexamples Generation. One of the most important functionalities of symbolic model checking is the ability to generate a counterexample for unsatisfied properties. The algorithms used in NUSMV as a basis to perform the computation of counterexamples are the same as those implemented in CMU SMV and are those presented in [24].

4.2 Advanced Functionalities

NUSMV provides advanced model checking functionalities, including specialized algorithms for the verification of invariants, and heuristics aiming at the reduction of the state explosion problem. In the following these functionalities are briefly explained.

[7] This has the effect that for examples the BDD representing the encoding of $\mathtt{st} = \mathtt{c}$ is only in terms of the boolean variables $\mathtt{st}_0$ and $\mathtt{st}_1$.
[8] If $\mathcal{A}$ is a set, then with $\overline{\mathcal{A}}$ we mean the complement set.
[9] INVAR specifies time invariants using propositional formulas.

Bounded CTL. NuSMV allows expressing specifications in real-time CTL (RTCTL) [33]. For example, it is possible to express that α always leads to β in 0 to 4 time units with the RTCTL specification[10]:

$$\mathbf{AG}(\alpha \rightarrow \mathbf{AF}^{0..4}\beta)$$

It is assumed that a time unit corresponds to a step. This functionality is the same as in CMU SMV.

Invariant checking. An important class of CTL formulas is *invariants*, i.e., formulas of the form $\mathbf{AG}\alpha$, where α is propositional formula. The semantics of this kind of formulas is that α is true in all reachable states. The computation of this kind of formulas is based on a fix-point characterization [12]:

$$\mathbf{AG}\alpha = \mathbf{gfp}_Z[\alpha \wedge \mathbf{AX}Z]$$

This approach is not very direct, and can be quite inefficient. If the set of reachable states has been previously computed, then an invariant can be simply checked by performing a test of set inclusion between the states represented by α and the set of reachable states:

$$\mathbf{Reachable}(\mathcal{I}) \subseteq \alpha$$

NuSMV offers the possibility of checking invariants using either this or the standard model checking technique. These functionalities are the same as in CMU SMV. Moreover NuSMV, w.r.t. CMU SMV, has a specialized routine for checking invariants on the fly. This is performed by verifying at each step k of the reachability analysis the following condition:

$$R_k(\underline{x}) \subseteq \alpha$$

where $R_k(\underline{x})$ is the set of states reachable in k or fewer steps. If this test fails, then the invariant is not verified, and a counterexample leading to a state not satisfying the property is provided.

Quantitative characteristics computation. In NuSMV it is possible to compute quantitative information on the Kripke structure [14, 15]. In particular, it is possible to compute the exact bound on the delay between two specified events, expressed as CTL formulas:

$$\mathbf{MIN}[\alpha, \beta] \qquad \mathbf{MAX}[\alpha, \beta]$$

The $\mathbf{MIN}[\alpha, \beta]$ computes the set of states reachable from α. When a state satisfying β is encountered, we return the number of steps taken so far. If a fixed point is reached and no state satisfying β is found, then *infinity* is returned. $\mathbf{MAX}[\alpha, \beta]$ has the dual interpretation, and returns the length of the longest path from a state in α to a state in β. If there exists an infinite path beginning in a state in α that never reaches a state in β, then *infinity* is returned. This functionality is the same as in CMU SMV.

LTL model checking. LTL is important because it allows us to express properties, such as fairness,[11] which are not expressible in CTL. Furthermore, LTL can be useful to perform selective analysis, namely to specify the paths of interest.

NuSMV supports LTL model checking implementing the algorithm presented in [32]. An LTL formula is automatically converted into a tableau [45], which is then used to extend the original model in synchronous product. The result is provided by checking the validity of an automatically generated CTL formula from the LTL specification in the extended model.

The loose integration presented in [32] allows for the use of CMU SMV as a black box by generating a new input file containing both the original description of the system and the code implementing the tableau. However, for each LTL formula to be verified the verification process has to be restarted from scratch, starting from the parsing of the newly generated input file. In NuSMV the tableau generation is tightly integrated. If more than one LTL property is specified, only the generation of the corresponding tableau is required, thus avoiding, each time, the generation of the whole model. The generated tableau is parsed in, compiled into BDD and synchronously composed with the original model.[12]

[10] In the NuSMV input language this specification is written as `SPEC AG(alpha -> ABF 0..4 beta)`

[11] A typical fairness property is "if a process is infinitely often executable then it is infinitely often executed", that corresponds to the LTL formula: "$\mathbf{GF}(P.executable) \rightarrow \mathbf{GF}(P.executed)$".

[12] Other model checkers [6, 41] are able to perform LTL model checking. In VIS [6], for example, it is possible to perform LTL model checking by using the language emptiness feature [60], but the user has to write by hand the automaton representing the acceptance condition of the LTL formula, and for each formula a different input file has to be generated and the computation must restart from scratch.

Partitioning of the model. One of the most important operations in model checking is the relational product:

$$\exists \underline{x}.(S(\underline{x}) \wedge T(\underline{x}, \underline{x}'))$$

which computes the set of states reachable from $S(\underline{x})$ through the transition relation $T(\cdot,\cdot)$. This operation can be performed as a single step by most BDD packages. Although it works quite well in practice, this operation has worst case exponential complexity [47]. The transition relation is said to be *monolithic*, as it consists of a single BDD.

In many practical cases, building the BDD for $T(\cdot,\cdot)$ may not be feasible. In many cases, however, it is possible to exploit the structure of the system and build the transition relation as a list of small BDDs, called clusters, which are implicitly disjoined (e.g., with an asynchronous model of concurrency) or conjoined (e.g., with synchronous systems) [9, 10, 11].

$$\begin{array}{ll} T(\cdot,\cdot) = \bigvee_i T_i(\cdot,\cdot) & \text{disjunctive} \\ T(\cdot,\cdot) = \bigwedge_i T_i(\cdot,\cdot) & \text{conjunctive} \end{array}$$

In both cases the monolithic relational product is reduced to a sequence of disjunctively/conjunctively composed relational products on the clusters.

With a disjunctively partitioned transition relation, the relational product can be computed as follows:

$$\begin{array}{c} \exists \underline{x}.(S(\underline{x}) \wedge T(\underline{x}, \underline{x}')) = \exists \underline{x}.(S(\underline{x}) \wedge \bigvee_i T_i(\underline{x}, \underline{x}')) = \\ \bigvee_i \exists \underline{x}.(S(\underline{x}) \wedge T_i(\underline{x}, \underline{x}')) \end{array}$$

In this way the relational product can be computed without ever building the BDD for the monolithic transition relation by distributing the existential quantification over disjunctions. The relational product is decomposed into a series of relational products involving relatively small BDDs.

For synchronous systems, NuSMV implements techniques based on early variable quantifications [59, 9, 10]. The basic idea is to find an ordering of the partitions $T_i(\cdot,\cdot)$ such that the quantification can be pushed inside the formula as much as possible, thus allowing relational products between small BDDs and existential quantification on a small number of variables, and thus reducing the complexity of the whole operation:

$$\begin{array}{c} \exists \underline{x}.(S(\underline{x}) \wedge T(\underline{x}, \underline{x}')) = \exists \underline{x}.(S(\underline{x}) \wedge \bigwedge_i T_i(\underline{x}, \underline{x}')) = \\ \exists \underline{x}_1.(\exists \underline{x}_2.(\ldots(\exists \underline{x}_{n-1}.(\exists \underline{x}_n.(S(\underline{x}) \wedge T_n(\underline{x}, \underline{x}')))\wedge \\ \wedge T_{n-1}(\underline{x}, \underline{x}')) \wedge \ldots \wedge T_1(\underline{x}, \underline{x}'))) \end{array}$$

where $\underline{x}_i$ are disjoint sets of variables, $\underline{x}_i$ being the set of variables the $T_j(\cdot,\cdot)$ with $j > i$ depends on, but $T_i(\cdot,\cdot)$ does not depend on.

NuSMV allows the user to perform model checking using the monolithic transition relation or the disjunctively or conjunctively partitioned transition relation. For conjunctive partitioning, the heuristics described in [35] and in [53] are available. They allow the user to compute the clusters for the conjunctively partitioned transition relation.

4.3 User Functionalities

The quality of interaction with the system is very important in several phases of the verification task. NuSMV provides various user functionalities. In the following these functionalities will be briefly explained.

Counterexample navigation. The ability to generate counterexamples for properties that are not verified is very useful in the design phase and is common to the majority of model checkers. In NuSMV it is also possible to navigate and inspect counterexamples, similarly to what a programmer can do in a debugger for a programming language. The user can jump from state to state, possibly between different counterexamples, and evaluate CTL expressions in a given state of a counterexample, thus generating a possible new debug trace. These functionalities are available via commands of the interactive shell.

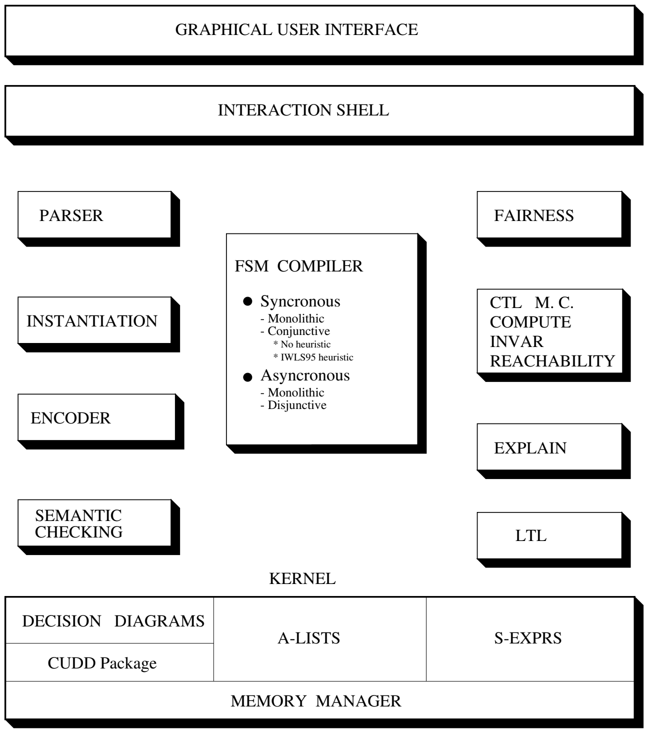

Figure 10: The NuSMV system architecture

On the fly modification of the model. The user might want to try to modify the model (e.g to try to fix a bug) or add new fairness constraints without restarting the computation from scratch. In NUSMV this can be done using shell commands that allow the user to modify interactively the model. The user can impose a new constraint on the set of initial states, the set of invariant states or on the transition relation. Let $\mathcal{I}$ be the set of initial states, Inv the set of invariants, $T(\cdot,\cdot)$ the transition relation and $\mathcal{H}$ the set of fairness constraints. Then the modifications act as follows:

$$\begin{array}{rcl} \mathcal{I} := \mathcal{I} \cap \mathcal{I}_a & \qquad & Inv := Inv \cap Inv_a \\ T(\cdot,\cdot) := T(\cdot,\cdot) \cap T_a(\cdot,\cdot) & \qquad & \mathcal{H} := \mathcal{H} \cup \{h_a\} \end{array}$$

where $\mathcal{I}_a$, Inv_a are the added set of initial states and invariant states respectively. $T_a(\cdot,\cdot)$ is the added transition relation and h_a is the new fairness constraint. Moreover, NUSMV allows the user to undo the modifications and restore the original status of the system. This is useful in case the user wants to try further modifications.

Help on line. NUSMV features an on-line help functionality. Each command provided by the interactive shell has a short help note describing briefly its options. Moreover, a complete description of each command is available via the NUSMV interactive shell in text format and, via a web browser, in HTML format.

5 System Architecture

One of the most important features of NUSMV is that it is an open system, and can be easily modified, customized or extended.[13] This is possible because the architecture of NUSMV is structured and organized in modules. Each module implements a set of functionalities and communicates with the others via a precisely defined interface. A schema of the system architecture can be found in Figure 10. The modules implementing reduction techniques were designed to work directly on the BDD representation, thus implementing their functionalities independently from the

[13] A similar philosophy of developing an open platform is advocated by the authors of the CADP tool-box [34].

input language used to describe the system. This provides a clear distinction between the system back-end and front-end. This clear distinction makes NUSMV open and usable in different fields, simply replacing or removing the modules specific of the kind of input. Section 7.2 describes an example of extension of NUSMV.

The various modules of the architecture and the provided functionalities are described below.

Parser. This module provides all the parsing routines, which process a file written in NUSMV language, check its syntactic correctness, and build a parse tree representing the internal format of the input file. The same parsing routines are used to implement commands for the interaction shell, such as the specification at command line of new properties to be verified.

Instantiation. This module processes the parse tree, and performs the instantiation of the declared modules, building a description of the finite state machine (FSM) representing the model (e.g., the transition relation, the initial states, the fairness).

Encoder. This module performs the encoding of data types and finite ranges into boolean domains. Having this module as a separate one makes it possible to have different encoding policies that can be more appropriate for different kinds of variables. Currently only the standard CMU SMV encoding is possible, but there are plans to integrate into this architecture other forms of encoding, e.g., those used in Word-Level SMV [25].

The semantic check module. This module is responsible for all the necessary semantic checks on the model, e.g., the absence of circular definitions [47].

FSM Compiler. This module provides the routines for constructing and manipulating FSMs at the BDD level. The FSMs can be represented in monolithic or partitioned form [9, 10, 11]. The heuristics used to perform the conjunctive partitioning of the transition relation and reordering of the clusters [35, 53] were developed to work at the BDD level, independently of the input language. This is a precise design choice: since NUSMV is intended to be applicable to a wide range of domains, for which the current input language might not be appropriate, it is important that the heuristics do not depend on the input language. Note that for a specialized model checker this might not be a suitable choice. For instance, since VIS [6] is highly specialized for hardware, its heuristics were developed to work directly on the BLIF format [56], which is a netlist[14]. The interface to other modules is given by the primitives for the computation of image and pre-image of set of states. These primitives are independent from the method used to represent the transition relation (monolithic, conjunctive or disjunctive).

Fairness. This module deals with the fairness constraints. It provides the model checking module with the information contained in the fairness constraints, if any.

Model Checking. This module provides all the {RT}CTL model checking functionalities, such as reachability and model checking routines, routines that perform computation of quantitative characteristics, and check invariants on the fly. All these routines are based on the image and pre-image computation primitives, and are independent of the particular method used to represent the FSM.

Explanation module. The routines for counterexamples and witness generation and inspection. Counterexamples and witnesses can be produced with different levels of verbosity, in the form of reusable data structures, and can subsequently be inspected and navigated. These routines are independent of the particular method used to represent the FSM.

[14]A netlist is a representation of a design at the structural level. A FSM can be viewed as a behavioral description of the design. A netlist is much closer to the implementation of the design than a simple FSM. A netlist is essentially the data structure used by many logic synthesis tools ,like VIS[6], to internally represent designs derived from HDL descriptions.

LTL module. The LTL module is a separate module that calls an external program that translates the LTL formula into a tableau suitable to be loaded into NuSMV. This program also generates a new CTL formula to be verified on the synchronous product of the original system and the generated tableau. This module was plugged in on top of the other modules (e.g., the Parser, the Encoder, the FSM Compiler, the Model Checking and the Explanation modules), thus reusing a lot of the functionalities already present in the architecture.

Kernel. The kernel provides the low level functionalities such as dynamic memory allocation, and manipulation of basic data structures (e.g., cons cells, hash tables). The kernel also provides all the basic BDD primitives, which are actually taken from the CUDD [57] BDD package. The CUDD package is a very complete BDD package, freely available on the web. It provides all the basic BDD operations, plus a series of advanced BDD functionalities (such as various state of the art BDD variable reordering algorithms, the possibility of grouping together variables in such a way that the reordering algorithms treats them as a single element, algorithms to perform BDD minimization, approximation and decompositions, ...). The CUDD package is integrated in NuSMV with a precisely defined interface. This makes it possible to replace the CUDD package with other state of the art BDD packages developed in the future, or with commercial BDD packages (such as the TiGeR [30] BDD package). The NuSMV kernel can be used as a black box, following coding standards that have been precisely defined.

Interactive shell. The interactive shell was designed on top of all the other modules. From the interactive shell the user has full access to all the functionalities provided by the system. The interactive shell is fully extensible, thus allowing for the development of new commands. (For more details see Section 3.2.1.)

Graphical user interface. The graphical user interface is a separate process, which communicates with NuSMV by sending textual commands to the interactive shell. It allows the user to inspect and set the value of the environment variables of the system, and provides full access to all the functionalities. Figures 7, 8 and 9 show some snapshots of the graphical interface of NuSMV. (For more details see Section 3.2.3.)

6 Implementation

NuSMV was designed to be robust and easy to maintain and to modify. The features of the NuSMV implementation are listed below. They are grouped depending the design task that they are intended to accomplish.

Robustness.

- NuSMV is written in ANSI C [55] and is POSIX [42] compliant. This makes the system portable to any compliant platform. At the moment the system has been successfully compiled on various operating systems and platforms (e.g., under Sun Solaris for SPARC and Intel X86, under SunOs 4.1.X, under various versions of the Linux operating system).
- NuSMV has been throughly debugged with `Purify`[15] to detect memory leaks and runtime memory corruption errors. The use of this tool guarantees the elimination of runtime problems in all parts of the application, and thus the delivery of a more reliable and robust application.
- The kernel of NuSMV provides low level functionalities, such as dynamic memory allocation, independently from the underlying operating system and hardware platform. Moreover, it provides routines for the manipulation of basic data structures such as cons cells, hash tables, arrays of generic types, and encapsulates the CUDD BDD package [57].

Maintainability.

- In order to implement the architecture depicted in Section 5, the source code of NuSMV has been separated into different packages. At the moment NuSMV is composed of 11 packages. Each package exports a set of routines that manipulate the data structures defined in the package and modify the options associated to the

[15]More information on this tool can be found at the url "`http://www.pureatria.com`".

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. We have packages for model checking, FSM compilation, BDD interfacing, LTL model checking and kernel functionalities. New packages can be added relatively easily, following precisely defined rules.

- The source code of NuSMV is maintained using a tool for revision controls. We use the RCS [58] tool provided by GNU. It automates the storing, retrieval, logging and merging of revisions and provides a simple and user friendly interface.
- In the coding, we have used an object-oriented programming style, following the ideas exploited by the VIS [6] system.
- The code of NuSMV is documented following the standards of the `ext` tool.[16] This tool allows for the automatic extraction of the programmer manual from the source comments in the system code. The programmer manual is available in TXT or HTML format, in a way that is browsable by an HTML viewer. This tool is also used to generate the help on line available through the interactive shell and via the graphical user interface.
- The user manual is written following the standard TEXINFO[16]. This allows us to have the user manual available in different formats (for instance POSTSCRIPT, PDF, DVI, INFO, HTML), directly from the NuSMV interactive shell, via an HTML viewer or in hardcopy.

7 Conclusions

We have presented the NuSMV symbolic model checker. The NuSMV architecture provides a precise distinction from the back-end (model checking algorithms, heuristics for optimal conjunctive partitioning, ...) and the front-end (the input language). The NuSMV back-end is general purpose, and can be used not only for verifying hardware but also for the verification in other fields in which systems can be modeled as an FSM (e.g., software,...).

The rest of this section compares the performance of NuSMV and CMU SMV, describes the development of a planner built on top of NuSMV, and some directions for future developments.

7.1 NuSMV vs. CMU SMV

The capabilities of NuSMV were tested on a series of examples taken from the literature, which were then used for a comparison with the original CMU SMV. Below is a brief description of the tests used and their sources:

- the $\{4,8,10,11\}$-bit alternating bit protocol by Armin Biere [4].
- a bounded retransmission (communication) protocol, by Klaus Havelund [40].
- the examples of the CMU SMV distribution.
- a model of the Shuttle Digital Autopilot, by Sergey Berezin.
- some models of the PCI Bus protocol, by Sergio Campos.
- a model of production cell, by Kirsten Winter [61].
- a model of a batch reactor, by S.T. Probst [50].
- some model of part of a preliminary version of the system requirements specification of TCAS II (Traffic Alert and Collision Avoidance System II), by William Chan [1].

These examples can be found in the distribution of NuSMV. Table 3 lists the results of such a comparative test. For each test, we report the number of (current and next) boolean variables necessary to represent the corresponding model ("# of BDD vars"), and the memory[17] and time required by the systems to analyze the model. We mark as time out

[16] More information about the `ext` documentation tool can be found at the url "`http://alumnus.caltech.edu/~sedwards/ext`"

[17] It is the maximum amount of memory allocated during state space generation and search, namely, the amount of memory allocated since the program starts to the end of the computation before freeing all the memory allocated.

the failure to the problems in 15000 seconds. For the examples marked with "(*)" a previously computed ordering file was provided to the system. The verification of the examples marked with a "(+)" required the modification of some of the parameters of the BDD package. The other symbols between parentheses are command line options passed to the model checkers: `-f` enables the computation of the set of reachable states (which is then used to restrict the search in model checking), while `-cp #` enables conjunctive partitioning with the threshold of each partition set to `#`. All these tests were performed on an Intel Pentium–II 300Mhz processor with 512Mb of RAM under Linux RedHat 5.0. The results listed in Table 3 show that NUSMV performs in most examples better (in 22 examples of the 39 listed w.r.t. to speed and in 8 examples w.r.t. to memory occupation) than CMU SMV, especially for larger examples. This enhancement in performance is mainly due to the use of the state of the art CUDD BDD package. In the tests none of the enhanced capabilities not present in CMU SMV (e.g., enhanced partitioning methods), were used.

7.2 NUSMV is open: the MBP example

NUSMV has been and is still used as the kernel of MBP, a planner based on model checking able to synthesize reactive controllers for achieving goals in nondeterministic domains [20, 21].[18] MBP has been obtained by substituting or eliminating some of the NUSMV modules. In particular, the NUSMV parser module has been redone (the two input languages are different); the NUSMV instantiation module has been eliminated (at the moment, in MBP there is no concept of hierarchy); the NUSMV semantic check module has been redone; the NUSMV FSM compiler has been redone to reflect the way in which the MBP input language is compiled into BDDs [17]. All the other modules have been left unchanged, and are shared between the two systems. Finally, a new module, containing all the special purpose, planning algorithms, has been added. This module provides routines based on the image and pre-image computation.

7.3 Future Directions

To make the system more usable and improve its efficiency and its expressiveness there are plans to introduce the following additional functionalities.

- A simulation functionality, which allows the user to acquire confidence with the correctness of the model before the verification of the properties.
- A sequential input language. The problem with the NUSMV language is that it is most amenable for hardware and hardware-like systems, while its ability to model software systems is left to the user. This leaves the user with the burden of a complex model generation activity. In general this can be hardly acceptable, as many specification languages (e.g., SDL) are intrinsically sequential. A possible new input language for NUSMV is VERUS [13].
- Forward CTL model checking. In symbolic model checking, CTL formulas are evaluated backward (the base operation is **EX**, see Section 2). New algorithms have been developed that allow performing evaluation of CTL formulas via a forward state traversal [43]. These algorithms allow for the verification of large systems that cannot be handled by the classical backward algorithms.
- Cone of influence reductions [3]. The idea is to simplify the model w.r.t. the cone of influence of the formula to be verified, thus reducing the search space.
- Use of "reachability don't cares" (RDC) [63] to reduce the cost of CTL model checking.
- High density reachability analysis [54] and under and over approximate reachability analysis [48]. These have shown in some cases, together with RDC, dramatic effects in the performance of CTL model checking.
- Optimizations for constraint-rich models [62], to enable the verification of systems with complex time-invariant constraints.

Some lines of research that are currently under study and whose results we plan to integrate inside NUSMV are specific reduction techniques for sequential systems, a set of abstraction techniques that implement certain heuristics developed in the theorem proving community [38] and that we believe will be very effective, the extension of CTL model

[18] See [37] for an introduction and a survey of planning as model checking.

		NuSMV		CMU SMV	
Example file name	# of BDD vars	Memory (KB)	Time (secs)	Memory (KB)	Time (secs)
abp4.smv	66	4938	9.4	960	10.6
abp4.smv (-f)	66	2818	1.5	960	3.2
abp8.smv	98	5886	81.4	1984	117.3
abp8.smv (-f)	98	15074	114.5	6528	527.7
abp10.smv	114	14275	632.9	11776	1011.4
abp10.smv (-f)	114	66767	2712.2	—	time out
abp11.smv	122	43355	3121.2	—	time out
abp11.smv (-f)	122	103251	12554.5	—	time out
brp.smv	98	23966	178.4	30784	1218.7
brp.smv (-f)	98	5346	3.5	1280	4.4
guidance.smv (*)	190	5090	23.0	1728	21.8
guidance.smv (-f) (*)	190	3926	4.8	1408	4.9
p-queue.smv (*)	86	2318	0.4	1088	0.2
p-queue.smv (-f) (*)	86	2318	0.4	1088	0.1
prod-cons.smv (*)	58	6610	80.9	2880	105.1
prod-cons.smv (-f) (*)	58	5594	11.8	1152	33.0
production-cell.smv (*)	108	5826	72.0	2368	97.0
production-cell.smv (-f) (*)	108	2583	1.8	1088	0.4
base.smv (-f) (*)	148	3223	2.2	1536	1.9
idle.smv (-f) (*)	150	7710	90.1	6144	162.0
counter.smv	6	1334	0.0	896	0.0
dme1.smv (-f) (*)	108	3618	1.3	2368	9.4
dme1-16.smv (-f) (*)	576	25310	277.7	—	time out
dme1-16.smv (-cp 2000 -f) (*)	576	15299	335.5	8768	454.4
dme2.smv (-f) (*)	112	3335	1.0	1088	0.6
dme2-16.smv (-f) (*)	586	27050	6313.1	—	time out
dme2-16.smv (-f) (+) (*)	586	49099	1821.9	68864	4985.8
gigamax.smv	88	4142	1.9	1216	2.0
mutex.smv	10	1342	0.0	896	0.0
mutex1.smv	14	1370	0.1	896	0.0
pci3p.smv (-f) (*)	92	1862	0.2	960	0.1
pci4p.smv (-f) (*)	128	5422	20.0	1152	53.0
periodic.smv (-f)	72	1786	1.0	960	0.1
ring.smv (-f)	10	1342	0.0	1152	0.0
robot.smv (-f) (*)	88	2894	8.7	1280	1.5
semaphore.smv (*)	14	1350	0.0	896	0.0
syncarb10.smv	60	3674	1.2	1216	1.0
syncarb5.smv	30	1455	0.1	896	0.0
tcas-t.smv (-cp 10000) (*)	292	98955	779.0	146816 (+)	1588.6 (+)

Table 3: The results of the comparison test.

checking to multi-agent systems and security applications [2], and the integration of model checking and theorem proving (SAT in particular) following the ideas reported in [36] and in [5].

References

[1] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. In David Garlan, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 156–166, San Francisco, CA, USA, October 1996. ACM. A full version to appear in IEEE Transactions on Software Engineering.

[2] M. Benerecetti, F. Giunchiglia, and L. Serafini. Model Checking Multiagent Systems. *Journal of Logic and Computation, Special Issue on Computational & Logical Aspects of Multi-Agent Systems*, 8(3):401–423, 1998. Also IRST-Technical Report 9708-07, IRST, Trento, Italy.

[3] S. Berezin, S. Campos, and E. Clarke. Compositional Reasoning in Model Checking. Technical Report CMU-CS-98-106, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, February 1998. To appear in proceedings of the COMPOS'97 workshop.

[4] Armin Biere. μcke – efficient μ-calculus model checking. In *International Conference on Computer-Aided Verification*, number 1254 in LNCS, pages 468–471. Springer, 1997.

[5] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99)*, March 1999. To appear.

[6] R. K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz S., Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: A system for Verification and Synthesis. In Rajeev Alur and Thomas A. Henzinger, editors, *Proc. Computer Aided Verification (CAV'96)*, number 1102 in LNCS, New Brunswick, New Jersey, USA, July/August 1996. Springer-Verlag.

[7] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[8] R. E. Bryant. Symbolic Boolean manipolation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[9] J. R. Burch, E. M. Clarke, and D. E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 403–407, Los Alamitos, CA, June 1991. IEEE Computer Society Press.

[10] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[11] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, April 1994.

[12] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[13] S. Campos, E. Clarke, and M. Minea. The Verus Tool: A quantitative approach to the formal verification of real-time systems. In Orna Grumberg, editor, *Proc. Computer Aided Verification (CAV'97)*, number 1254 in LNCS, Haifa, Israel, June 1997. Springer-Verlag.

[14] S. Campos, E. M. Clarke, W. Marrero, M. Minea, and H. Hiraishi. Computing Quantitive Characteristics of Finite-State Real-Time Systems. Technical Report CMU-CS-94-147, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1994.

[15] S. Campos and O. Grumberg. Selective quantitative analysis and interval model checking: Verifying different facets of a system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 257–268, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[16] R. J. Chassell and Richard Stallman. *Texinfo: the GNU documentation format*, October 1996.

[17] A. Cimatti, E. Giunchiglia, F. Giunchiglia, and P. Traverso. Planning via Model Checking: A Decision Procedure for $\mathcal{AR}$. In S. Steel and R. Alami, editors, *Proceeding of the Fourth European Conference on Planning*, number 1348 in Lecture Notes in Artificial Intelligence, pages 130–142, Toulouse, France, September 1997. Springer-Verlag. Also ITC-IRST Technical Report 9705-02, ITC-IRST Trento, Italy.

[18] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Model Checking Safety Critical Software with SPIN: an Application to a Railway Interlocking System. In *Proceedings of SAFECOMP'98 – Seventeenth International Conference on Computer Safety, Reliability and Security*, Heidelberg, Germany, 1998. Also IRST-Technical Report 9702-09, Trento, Italy. Presented at the Third SPIN Workshop, Twente University, Enschede, The Netherlands, April 1997.

[19] A. Cimatti and M. Roveri. NUSMV 1.0: User Manual. Technical report, ITC-IRST, Trento, Italy, December 1998.

[20] A. Cimatti, M. Roveri, and P. Traverso. Automatic OBDD-based Generation of Universal Plans in Non-Deterministic Domains. In *Proceeding of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, Wisconsin, 1998. AAAI-Press. Also IRST-Technical Report 9801-10, Trento, Italy.

[21] A. Cimatti, M. Roveri, and P. Traverso. Strong Planning in Non-Deterministic Domains via Model Checking. In *Proceeding of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, Carnegie Mellon University, Pittsburgh, USA, June 1998. AAAI-Press.

[22] E. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K.McMillan, and L. Ness. Verification of the FUTURE-BUS+ cache coherence protocol. In *11th CHDL*, 1993.

[23] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time tem poral logic. In *Logic of Programs: Workshop*. Springer Verlag, May 1981. Lecture Notes in Computer Science No. 131.

[24] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking. In *32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, June 1995.

[25] E. M. Clarke, M. Khaira, and X. Zhao. Word Level Model Checking Avoiding the Pentium FDIV Error. In *33rd Design Automation Conference (DAC'96)*, pages 645–648, New York, June 1996. Association for Computing Machinery.

[26] E. M. Clarke, K. L. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 419–422, New Brunswick, NJ, USA, July/August 1996. Springer Verlag.

[27] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[28] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[29] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, Grenoble, France, June 1989. Springer-Verlag.

[30] O. Coudert, J. C. Madre, and H. Touati. *TiGeR Version 1.0 User Guide*. Digital Paris Research Lab, December 1993.

[31] D. L. Dill, A, J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design, VLSI in Computers and Processors*, pages 522–525, Los Alamitos, Ca., USA, October 1992. IEEE Computer Society Press.

[32] O. Grumberg E. Clarke and K. Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):57–71, February 1997.

[33] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. Quantitative Temporal Reasoning. In *Proc. Computer Aided Verification*, LNCS. Springer-Verlag, 1990.

[34] J.C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. CADP: A Protocol Validation and Verification Toolbox. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th Conference on Computer-Aided Verification (CAV'96)*, number 1102 in Lecture Notes in Computer Science, pages 437–440, New Brunswick, NJ, USA, August 1996. Springer-Verlag.

[35] D. Geist and I. Beer. Efficient model checking by automated ordering of transition relation partitions. In D. L. Dill, editor, *Proc. Computer Aided Verification (CAV'94)*, number 818 in LNCS, Stanford, California, USA, June 1994. Springer-Verlag.

[36] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. of the 13th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, New Brunswick, NJ, USA, August 1996. Springer Verlag. Also DIST-Technical Report 96-0037 and IRST-Technical Report 9601-02.

[37] F. Giunchiglia and P. Traverso. Planning as Model Checking. In Susanne Biundo, editor, *Proceeding of the Fifth European Conference on Planning*, Lecture Notes in Artificial Intelligence, Durham, United Kingdom, September 1999. Springer-Verlag.

[38] F. Giunchiglia and T. Walsh. A Theory of Abstraction. *Artificial Intelligence*, 57(2-3):323–390, 1992. Also IRST-Technical Report 9001-14, IRST, Trento, Italy.

[39] V. Hartonas-Garmhausen, S. Campos, A. Cimatti, E. Clarke, and F. Giunchiglia. Verification of a Safety-Critical Railway Interlocking System with Real-time Constraints. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing (FTCS-28)*, pages 458–463, Munich, Germany, June 1998. IEEE Computer Society Press.

[40] Klaus Havelund and Natarajan Shankar. Experiments in Theorem Proving and Model Checking for Protocol Verification. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 662–681. Springer-Verlag, March 1996.

[41] G. J. Holzmann. The model checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[42] IEEE. *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

[43] H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 82–87, Washington, November10–14 1996. IEEE Computer Society Press.

[44] S. A. Kripke. Semantical considerations on modal logic. In *Proc. A colloquium on Modal and Many-Valued Logics*, Helsinki, 1962.

[45] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107. ACM, ACM, January 1985.

[46] K. L. McMillan and J. Schwalbe. Formal verification of the encore gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, pages 242–251. (sponsored by Information Processing Society, Tokyo, Japan), 1991.

[47] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publ., 1993.

[48] In-Ho Moon, Jae-Young Jang, Gary D. Hacthtel, Fabio Somenzi, Jun Yuan, and Carl Pixley. Approximate Reachability Don't Cares fo CTL Model Checking. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, San Jose, California, November8–12 1998. IEEE Computer Society Press.

[49] C. Pixley. A Computational Theory and Implementation of Sequential Hardware Equivalence. In *DIMACS Workshop on Computer Aided Verification '90*, pages 293–320, Providence, RI, 1990.

[50] S. T. Probst. *Chemical Process Safety and Operability Analysis using Symbolic Model Checking*. PhD thesis, Department of Chemical Engineering – Carnegie Mellon University, Pittsburgh, PA1523 USA, May 1996.

[51] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, Lecture Notes in Computer Science, Vol. 137, pages 337–371. SV, Berlin/New York, 1982.

[52] J.P. Queille. Le systéme CESAR: Description, spécification et analyse des applications réparties. Thesis, Uuniversité Scientifique et Médicale de Grenoble, June 1982.

[53] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD algorithms for FSM synthesis and verification. In *IEEE/ACM Proceedings International Workshop on Logic Synthesis*, Lake Tahoe (NV), May 1995.

[54] K. Ravi and F. Somenzi. High-density reachability analysis. In *International Conference on Computer Aided Design*, pages 154–158, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.

[55] Herbert Schildt, American National Standards Institute, International Organization for Standardization, International Electrotechnical Commission, and ISO/IEC JTC 1. *The annotated ANSI C standard: American National Standard for Programming Languages C: ANSI/ISO 9899-1990*. Osborne/McGraw-Hill, Berkeley, CA, USA, 1990.

[56] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[57] F. Somenzi. CUDD: CU Decision Diagram package — release 2.1.2. Department of Electrical and Computer Engineering — University of Colorado at Boulder, April 1997.

[58] Walter F. Tichy. RCS: A system for version control. *Software Practice and Experience*, 15(7):637–654, July 1985.

[59] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDDs. In Satoshi Sangiovanni-Vincentelli, Alberto; Goto, editor, *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 130–133, Santa Clara, CA, November 1990. IEEE Computer Society Press.

[60] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Symposium on Logic in Computer Science (LICS '86)*, pages 332–345, Washington, D.C., USA, June 1986. IEEE Computer Society Press.

[61] K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

[62] Bwolen Yang, Reid Simmons, Randal E. Bryant, and David R. O'Hallaron. Optimizing Symbolic Model Checking for Constraint-Rich Models. In N. Halbwachs and D. Peled, editors, *Eleventh Conference on Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 328–340, Trento – Italy, July 1999. Springer.

[63] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. In *Proceedings of the 9th International Conference on Computer Aided Verification CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 376–387, 1997.