Innovations in Systems and Software Engineering manuscript No.
(will be inserted by the editor)

A Model Advisor for NuSMV Specifications

Paolo Arcaini · Angelo Gargantini · Elvinia Riccobene

Received: date / Accepted: date

Abstract Among possible model validation techniques able to identify defects early in the system development, model review aims also at determining if a model is of sufficient quality, since quality is measured as the absence of certain faults. In this paper, we tackle the problem of automatic reviewing NuSMV formal specifications by developing a *model advisor* which helps to assure given model qualities for NuSMV programs. Vulnerabilities and defects a developer can introduce during the modeling activity using NuSMV are expressed as the violation of formal meta-properties. These meta-properties are then mapped to temporal logic formulas, and the NuSMV model-checker itself is used as engine of our model advisor to notify meta-properties violations, so revealing the absence of some quality attributes of the specification. As a proof of concept, we also report the result of applying this review process to several NuSMV specifications.

Keywords Model Advisor · Model Review · NuSMV

1 Introduction

System validation is an essential activity of any development process since it permits detecting faults and assuring that given requirements are guaranteed by the system under development. Nowadays, the use of formal methods, based on rigorous mathematical foundations, is becoming of extremely importance for system design and development, since abstract formal models allow detecting faults in the specification as early as possible with limited effort. Model validation should precede the application of more expensive and accurate verification methods, that should be applied only when a designer has enough confidence that the specification really reflects the user perceptions. Otherwise (right) properties could be proved true for a wrong specification.

Among validation techniques, *model review*, also known as "model walk-through" or "model inspection", allows to critically examine modeling efforts to determine if a model not only fulfills the intended requirements, but also are of sufficient quality to be easy to develop, maintain, and enhance. This process should, therefore, assure a certain degree of quality. The assurance of quality, namely ensuring readability and avoiding error-prone constructs, is one of the most essential aspects in the development of safety-critical reactive systems, since the failure of such systems – often attributable to modeling and, therefore, coding flaws – can cause loss of property or even human life [19]. When model reviews are performed properly, they can have a big payoff because they allow to identify defects early in the system development, reducing the cost of fixing them. Usually model review, which comes from the code-review idea, is performed by a group of external qualified people, often both technical staff and project stakeholders, who meet together to evaluate models and documents.

A weak aspect of the review process is that it is usually done by hand. This requires a great effort that might be tremendously reduced if performed in an automatic way by systematically checking specifications for known vulnerabilities or defects. The question is *what* to check on and *how* to automatically check the model. In other words, it is necessary to identify classes of faults and defects to check, and to establish a process by which to detect such deficiencies in the model. If these faults are expressed in terms of formal statements, these can be assumed as a sort of "mea-

P. Arcaini, E. Riccobene
Dept. of Information Technology, Università degli Studi di Milano, via Bramente 65, Crema, Italy
E-mail: {paolo.arcaini,elvinia.riccobene}@unimi.it

A. Gargantini
Dipartimento di Ingegneria dell'informazione e metodi matematici, Università degli Studi di Bergamo, viale Marconi 5, Dalmine, Italy
E-mail: angelo.gargantini@unibg.it

sure" of the *model quality assurance*. A tool is also necessary to make the process automatic. It would work as *model advisor* to check a model for conditions and configuration settings that can result in inaccurate or inefficient behavior of the system that the model represents.

In this paper, we tackle the problem of automatic reviewing NuSMV [10, 1] formal specifications. We develop a model advisor for NuSMV programs which helps the developers to assure given model qualities. We first detect a family of vulnerabilities and defects a developer can introduce during the modeling activity using NuSMV and we express such faults as the violation of formal properties. These properties refer to model *attributes* and characteristics that should hold in any NuSMV model, independently from the particular model to analyze. For this reason they are called *meta-properties*. They should be true in order a NuSMV model to have the required quality attributes. Therefore, they can be assumed as measures of model quality assurance. Depending on the meta-property, its violation indicates the presence of actual faults, or only of potential faults.

These meta-properties are defined in terms of logical operators *Always*, *Sometime*, and *InitiallyA*/*InitiallyS* to capture properties that must be true in every state, eventually true in at least one state, or referring to initial states (always or sometimes true) of the NuSMV model under analysis. Then, we map these logical operators to temporal logic formulas and we exploit the NuSMV model checking facilities to check the meta-property violations.

A similar model reviewing technique was presented in [6] for the Abstract State Machine (ASM) formal specifications [9]. We here refer to NuSMV specifications since NuSMV is widely used as formal specification method, it is endowed with a simulator and a model checker that makes possible to handle and to automatize our approach, and there exists a wide repository of specification case-studies on which to test our model advisor. In any case, the results obtained for the ASM and NuSMV models can be adapted to other state-transition based formal approaches.

The NuSMV formal method and the structure of a formal specification is briefly presented in Section 2. Section 3 defines a function, later used in the meta-properties definition, that statically computes the assignment condition under which a model variable is updated upon state changing. Meta-properties able to guarantee certain quality attributes of a specification are introduced in Section 4. In Section 5, we describe how it is possible to automatize our model review process by exploiting the use of the NuSMV itself as a model checker to check the possible violations of meta-properties. The general architecture of our model advisor is described in Section 6. As a proof of concept, in Section 7 we report the results of applying our model advisor to a certain number of benchmark models of various degree of complexity: some taken from the NuSMV source distribution, others found on the Internet, other obtained from ASM models of real case studies and we discuss the fault detection capability of our analysis. In Section 8, we present other works related to the model review process. Section 9 concludes the paper and indicates some future directions of this work.

2 NuSMV

NuSMV [1] is known as a model checker derived from the CMU SMV [16]. It 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 Binary Decision Diagrams (BDD)-based and SAT-based model checking techniques. Heuristics are available for achieving efficiency and partially controlling the state explosion.

A NuSMV specification describes the behavior of a Finite State Machine (FSM) in terms of a "possible next state" relation between *states* that are determined by the values of variables. Transactions between states are determined by the updates of the variables.

According to the model operational description, a NuSMV specification is made of three principal sections:

– **VAR** that contains variables declaration. A variable type can be *boolean*, *Integer* defined over intervals or sets, *enumeration* of symbolic constants.

– **ASSIGN** that contains the initialization (by the instruction *init*) and the update mechanism (by the instruction *next*) of variables. A variable can be not initialized and in this case, at the beginning NuSMV creates as many states as the number of values of the variable type; in each state the variable assumes a different value.

– **CTLSPEC** (resp. **LTLSPEC**) that contains the CTL (resp. LTL) properties to be verified.

A **DEFINE** statement can also be used as a macro to syntactically replace an *identifier* by the *expression* it is associated with. The associated expression is always evaluated in the context of the statement where the identifier is declared.

A *state* of the model is an assignment of values to variables. According to the NuSMV language definition, there exist the following four ways to explicitly assign values to a variable:

```
−− simple assignment
ASSIGN identifier := simple_expression
−− init value
ASSIGN init( identifier ) := simple_expression
−− next value
ASSIGN next( identifier ) := next_expression
−− macro definition
DEFINE identifier := simple_expression
```

where `identifier` is a variable identifier; `simple_expressions` are built only from the values of variables in the current state and they cannot have a `next` operation inside; `next_expression` relates current and next state variables to express transitions in the FSM (see the NuSMV User Manual [1] for more details on the assignment syntax and restriction rules for assignments). In both `simple-` and `next-` expressions, a variable can be determined in a straight way, or in a conditional way[1]. Conditional expressions can be:

1. An `if-then-else` expresion

cond1 ? exp1 : exp2

which evaluates to `exp1` if the condition `cond1` evaluates to true, and to `exp2` otherwise.

2. A condition `case` expression:

case
left_expression_1 : right_expression_1 ;
...
left_expression_N : right_expression_N ;
esac

which returns the value of the first `right_expression_i` such that the corresponding `left_expression_i` condition evaluates to 1 (TRUE), and the previous i-1 left expressions evaluate to 0 (FALSE). The type of expressions on the left hand side must be boolean. An error occurs if all expressions on the left hand side evaluate to 0. To avoid this kind of errors, NuSMV performs a static analysis and if it believes that in some states no left expression may be true, it forces the user to add a *default case* with `left_expression` equal to 1. This kind of analysis is conservative: sometimes the user must add a default case even if it is not necessary, as in the following example in which the default case would be useless but NuSMV requires it.

```
MODULE main
VAR     fooInt : 1..3;
ASSIGN init( fooInt ) := 1;
       next( fooInt ) := case
                             fooInt = 1: 3;
                             fooInt = 3: 1;
                             1: 2;--useless, but still required
                         esac;
```

In NuSMV it is possible to model *non deterministic behaviours* by (a) do not assigning any value to a variable that, in this case, can assume any value of its finite domain; (b) assigning to a variable a value randomly chosen from a set. It is also possible to specify *invariant conditions* by the command `INVAR`.

NuSMV offers another more declarative way of defining initial states and transition relations. Initial states can be defined by the keyword `INIT` followed by characteristic properties that must be satisfied by the variables values in the initial states. Transition relations can be expressed by constraints, through the keyword `TRANS`, on a set of *current state/next state* pairs. Since the following equivalences hold, it is not restrictive to consider only the operational description for states and transitions.

ASSIGN a:=exp; is equivalent to **INVAR** a **in** exp;
ASSIGN init(a):=exp; is equivalent to **INIT** a **in** exp;
ASSIGN next(a):=exp; is equivalent to **TRANS next**(a) **in** exp;

3 Assignment Condition

As stated in Section 2, there exist different ways to assign values to NuSMV variables. Formally, an assignment is a pair $\langle$*identifier, expr*$\rangle$ where *identifier* is a variable identifier and *expr* is a `simple` or `next`-expression which provides the variable value.

We here present a method to compute, for each assignment $\langle$*identifier, expr*$\rangle$ defined in the specification under review, the list of conditions under which the assignment is actually performed, and the corresponding expression without conditions in it, which will determine the value assigned to the variable *identifier* when computed. For this purpose, we introduce a function *assignment condition*

$$AC : Assignment \rightarrow (Condition \times ValueExpr)^*$$

It is defined as $AC(\langle identifier, expr\rangle) = CV(expr)$ in terms of the function $CV : Expression \rightarrow (Condition \times ValueExpr)^*$ which extracts the conditions from a generic expressions by returning the list of pairs $\langle cond, valExpr\rangle$, where *cond* is the condition under which the expression takes the value given by the expression (without conditions) *valExpr*. *CV* is defined recursively as follows depending if *expr* is defined in terms of a conditional operator or not.

Expression without conditions. In this case *expr* is a constant, an identifier, a logical/algebraic expression, etc. The function yields $CV(expr) = \langle true, expr\rangle$.

Expression with conditions. In this case *expr* is expressed in terms of a `case` or an `if-then-else` operator. Before defining the function *CV* in these cases, let us introduce two auxiliary functions.

Let $\oplus_{i=1}^{n}(L_i)$ be the concatenation function among lists $L_i, i = 1,\ldots,n$.

Let $L \doteq [\langle c_i, e_i\rangle]_{i=1}^{n}$ be a list of pairs $\langle c_i, e_i\rangle$, $i = 1,\ldots,n$, with c_i a boolean condition and e_i an expression. We define a function $\bigwedge_a(L) \doteq [\langle a \wedge c_i, e_i\rangle]_{i=1}^{n}$ returning the list of pairs obtained from the elements of L by making the conjunction between the boolean condition a with the condition c_i.

[1] Among the different ways to build NuSMV expressions, we restrict our attention to the conditional ones since, for our purposes, we are interested into computing the conditions under which an assignment is actually performed.

– If *expr* is an `if-then-else` expression, *CV* holds:

$$CV(c?e1:e2) = \oplus(\wedge_c(CV(e1)), \wedge_{\neg c}(CV(e2)))$$

– If *expr* is a `case` expression, the *CV* function yields:

$$CV(expr) = \oplus_{i=1}^{N}(\bigwedge_{left\_expression\_i} (CV(right\_expression\_i)))$$

The following is an example of the computation of the function *AC* on a next_expression.

```
ASSIGN next(x) := case
                    a1: case
                            b1: 2;
                            b2: 3;
                        esac;
                    a2: c ? 5 : 6;
                    1: 7;
                  esac;
```

$AC(\langle x, next\_expr\rangle) = CV(next\_expr) =$
$[(a1 \wedge b1,2), (a1 \wedge b2,3), (a2 \wedge c,5), (a2 \wedge \neg c,6), (true,7)]$

4 Meta-properties

In this section we introduce some properties that should be proved in order to assure that a NuSMV specification has some quality attributes. These properties refer to attributes that are defined independently from the particular NuSMV specification to be analyzed and they should be true in order to guarantee certain degree of quality for the NuSMV model. For this reason we call them *meta-properties*.

The violation of a meta-property always means that a quality attribute is not met and may indicate a potential/actual fault in the model. A further discussion on this is given in Section 7. We have identified the following categories of model quality attributes.

– *Consistency* requires that there are no model statements (variable assignments, propriety specifications, behaviors, etc.) contrasting each others. For instance, MP9 requires that all the specified properties are true. Consistency of assignments to variables, one of the main goals of other model review techniques [6,12], is guaranteed in NuSMV by the semantics of the language. However, one could ask mutually exclusion of assignment conditions (MP3).

– *Completeness* requires that every system behavior is explicitly modeled. This encourages the explicit assignment of variables (MP7) and that at least one assignment condition, apart the default condition, is true (MP4).

– *Minimality* guarantees that the specification does not contain elements – i.e. variables, assignments, domain elements, etc. – defined or declared in the model but never used. Minimality of the assignments requires that every assignment can be performed (MP1, MP2) and it is really useful (MP5). Every value in the domains should be necessary (MP6) and every variable used (MP7 and MP8). Minimality of properties requires that property specifications are not vacuously satisfied (MP10). These defects are also known as *over specification*.

4.1 Meta-property definition

To formally specify the above attributes in terms of meta-properties we have identified properties that must be true in every state and properties that must be eventually true in at least one state of the NuSMV specification under analysis. Some properties refer only to the initial state (all, or some). Given a FSM M expressed in NuSMV and a predicate ϕ over a state of M, we define the operators *Always* and *Sometime* ranging on the whole computational state, and the operators initially always (*InitiallyA*) and initially sometimes (*InitiallyS*) ranging on the set of initial states. They are defined as follows:

$$\begin{aligned}
M &\models Always(\phi) &&= \forall s_0 \in S_0\ \forall s \in \mathscr{R}(s_0):\ \phi(s)\\
M &\models Sometime(\phi) &&= \exists s_0 \in S_0\ \exists s \in \mathscr{R}(s_0):\ \phi(s)\\
M &\models InitiallyA(\phi) &&= \forall s_0 \in S_0:\ \phi(s_0)\\
M &\models InitiallyS(\phi) &&= \exists s_0 \in S_0:\ \phi(s_0)
\end{aligned}$$

where S_0 is the set of initial states of M, and $\mathscr{R}(s_0)$ is the set of all the states reachable from s_0. In the following we present the meta-properties we have introduced, currently support, and use for automatic review of NuSMV models.

Most of the meta-properties are expressed in terms of the assignment condition function. For notational convenience, given an assignment $\alpha = \langle id, expr\rangle$, we denote by $AC_{\alpha,i}$ the condition $cond_i$ of the i-th element $\langle cond_i, val_i\rangle$ of the list $AC(\alpha)$. Moreover, we need to distinguish between assignments α regarding initial values, called α_{init} assignments, and not initial ones. For the sake of brevity, all the following meta-properties containing $AC_{\alpha/\alpha_{init},i}$ are universally quantified over assignment α/α_{init} and condition index i.

MP1 Every assignment condition can be true

We would like that every condition under which a variable is assigned a value can be eventually true, i.e. the model does not contain conditions which are always false. We have to distinguish between initial and not initial assignments.

This meta-property requires that every condition can be true in at least one initial state, and every condition is eventually true:

$$InitiallyS(AC_{\alpha_{init},i}) \quad \text{and} \quad Sometime(AC_{\alpha,i})$$

In the example 1, the condition $foo = BB$ is never satisfied.

```
MODULE main
VAR     foo: {AA, BB, CC};
ASSIGN  init(foo) := AA;
        next(foo) := case
                        foo = AA: CC;
                        foo = BB: AA;--never satisfied
                        foo = CC: AA;
                     esac;
```

Example 1 Violation of meta-property MP1

MP2 Every assignment is eventually applied

Even if a condition φ can be true by MP1, we like φ to be actually eventually evaluated and not to be masked by other conditions preceding it. In such case, we may suspect that the conditions in a `case` expression are involuntarily listed in a wrong order. It is guaranteed by proving:

$$InitiallyS(AC_{\alpha,i} \bigwedge_{j=1}^{i-1} \neg AC_{\alpha,j}) \text{ and}$$
$$Sometime(AC_{\alpha,i} \bigwedge_{j=1}^{i-1} \neg AC_{\alpha,j})$$

```
MODULE main
VAR     foo: 1..3;
ASSIGN  init(foo) := 1;
        next(foo) := case
                        foo = 1: 2;
                        foo > 1: {1, 3};
                        foo = 3: 1;-- never applied
                     esac;
```

Example 2 Violation of meta-property MP2

In the example 2, the condition $foo = 3$ is eventually satisfied but the corresponding assignment is never applied because the condition is masked by the previous condition $foo > 1$.

MP3 The assignment conditions are mutually exclusive

This meta-property requires that every condition explicitly and precisely models the conditions under which the assignment is applied. This guarantees that if the condition is true, then it is applied and it is not masked by another condition which precedes it.

$$\forall j\ j < i \quad InitiallyA(\neg(AC_{\alpha_{init}\,i} \wedge AC_{\alpha_{init}\,j}))$$
$$\forall j\ j < i \quad Always(\neg(AC_{\alpha,i} \wedge AC_{\alpha,j}))$$

```
MODULE main
VAR     hour: 0..23;
        hour12: 1..12;
        amPm: {AM, PM};
ASSIGN  init(hour) := 0;
        next(hour) := (hour + 1) mod 24;
        hour12 := case --conditions mutually exclusive
                        hour in {0, 12}: 12;
                        !(hour in {0, 12}): hour mod 12;
                  esac;
      amPm := case --conditions not mutually exclusive
                        hour < 12: AM;
                        hour >= 11: PM;
                  esac;
```

Example 3 Violation of meta-property MP3

In the example 3, even if the model is correct, that is the value of *amPm* and *hour*12 are correctly related to the value of *hour*, the two conditions of the assignment of the variable *amPm* are not mutually exclusive. To remove the violation we could, for example, change the second condition with the condition $hour > 11$.

MP4 For every assignment terminated by a default condition true, at least an assignment condition is true

We have already discussed in Section 2 that the conditions in case expression must be *complete*. However, sometimes NuSMV forces the user to add a last *true* condition even if he/she has already explicitly listed all the conditions in the case expression. The following meta-property requires that all the conditions before the last default condition are already *complete*. If $AC_{\alpha,n} = true$, then

$$InitiallyA(AC_{\alpha_{init},1} \vee \cdots \vee AC_{\alpha_{init},n-1})$$
$$Always(AC_{\alpha,1} \vee \cdots \vee AC_{\alpha,n-1})$$

This applies only when $AC_{\alpha,n} = true$, because otherwise NuSMV already guarantees completeness. If this meta-property is verified, the $AC_{\alpha,n} = true$ is useless and the conditions already cover every case.

```
MODULE main
VAR     foo: 2..4;
ASSIGN  init(foo) := 2;
        next(foo) := case
                        foo = 2: 4;
                        foo = 4: 3;
                        1: 2; --default condition useful
                     esac;
```

Example 4 Violation of meta-property MP4

In the example 4 the meta-property is violated, i.e. $Always(\text{foo} = 2 \vee \text{foo} = 4)$ is false, because, in the next expression of variable foo, the default condition is useful because the previous conditions do not cover all the cases. The meta-property would not be violated if the next expression would be rewritten, for example, in the following way:

```
next(foo) := case
                foo = 2: 4;
                foo = 4: 3;
                foo = 3: 2;
             esac;
```

MP5 No assignment is always trivial

We say that a *next* assignment $\langle var, expr \rangle$ is trivial if *var* is already equal to *expr*, even before the update is applied. This property requires that each assignment which is eventually performed, will not be always trivial, except it is explicitly formalized by the assignment $\langle var, var \rangle$ which assigns to var' its current value. The property

$$Sometime(AC_{\alpha,i} \bigwedge_{j=1}^{i-1} \neg AC_{\alpha,j}) \rightarrow Sometime(AC_{\alpha,i} \bigwedge_{j=1}^{i-1} \neg AC_{\alpha,j} \wedge var' \neq var)$$

states that, if eventually updated (see MP2), the variable *var* will be updated to a new value at least in one state. The more simple property $Sometime(AC_i \wedge var' \neq var)$ would be false if the assignment is never applied.

We borrowed the concept of trivial update from the Abstract State Machines [9].

```
MODULE main
VAR     shuffle : boolean;
        foo: {AA, BB};
ASSIGN next(foo) := case
                        ! shuffle  & foo = AA: AA;--trivial
                        ! shuffle  & foo = BB: BB;--trivial
                         shuffle : {AA, BB};
                    esac;
```

Example 5 Violation of meta-property MP5

In the example 5, in the next expression of variable foo the first two assignments are always trivial. The next expression could be rewritten in a more simple equivalent way:

```
next(foo) := case
                shuffle : {AA, BB};
               ! shuffle : foo;
             esac;
```

MP6 Every variable can take any value in its domain

This meta-property requires that every variable, except the module instantiations, takes all the values of its domain. For each variable *var*, whose domain values are $e_1, \ldots, e_n$, the property

$$Sometime(var = e_1) \wedge \ldots \wedge Sometime(var = e_n) \quad (1)$$

states that variable *var* takes all the values of its domain. Since each $Sometime(var = e_i)$ is checked individually, we can know all the values never taken; these values, if they are really unnecessary, can be removed from the variable domain.

```
MODULE main
VAR     foo: {1, 2, 3}; --never takes value 2
ASSIGN init (foo) := 1;
       next(foo) := (foo * 3) mod 4;
```

Example 6 Violation of meta-property MP6

In the example 6, variable foo never takes value 2.

MP7 Every variable not explicitly assigned is used

In NuSMV there is no definition of monitored variables, i.e. variables that are updated by the environment. However, the variables that are not explicitly defined by a `DEFINE` statement or by a next assignment, can be considered as monitored, since at every step they can take any value in their domain. Monitored variables should be used in other parts of the specification. We say that a variable is *used* if it occurs in an assignment (in another `ASSIGN` or `DEFINE` expression) or in a property (a `SPEC` clause).

The verification of this meta-property can be performed statically by analysing the specification without the use of the proving capabilities of the model checker.

```
MODULE main
VAR     foo: boolean;
        fooMU: boolean;--monitored variable used
        fooMEA: boolean;--variable explicitly assigned
        fooMNU: 1..3;--monitored variable not used
ASSIGN foo := !fooMU;
        fooMEA := {0, 1};
```

Example 7 Violation of meta-property MP7

In the example 7 $fooMU$ and $fooMNU$ are both monitored variables. $fooMU$ satisfies the meta-property because is used in the assignment of variable foo; $fooMNU$, instead, violates the meta-property because is never read. The variable $fooMEA$ satisfies the meta-property because it is explicitly assigned.

MP8 Every independent variable is used

In NuSMV a variable x can be assigned in the next state to a value which depends only on x. In this case we say that the variable is *independent*, since it does not depend on other variables. Independent variables are generally used to model monitored variables which however have some constraints for their behaviour. These variables should be used in other parts of the model.

```
MODULE main
VAR     foo: boolean;
        fooIU: boolean;--independent variable used
        fooINU: 0..4; --independent variable not used
ASSIGN foo := !fooIU;
        init (fooIU) := 1;
        next(fooIU) := !fooIU;
        init (fooINU) := 0;
        next(fooINU) := (fooINU + 1) mod 5;
```

Example 8 Violation of meta-property MP8

In the example 8 $fooIU$ and $fooINU$ are both independent variables. $fooIU$ satisfies the meta-property because is used in the assignment of variable foo; $fooINU$, instead, violates the meta-property because is never read.

MP9 Every property is proved true

This meta-property simply requires that every property is proved.

MP10 No property is vacuously satisfied

A well known problem in formal verification is vacuous satisfaction: A property is vacuously satisfied if that property is satisfied and proved true regardless of whether the model really fulfills what the specifier originally had in mind or not. For example, the LTL property $G(x \rightarrow X(y))$ is vacuously satisfied by any model where x is never true. Vacuity is an indication of a problem in either the model or the property. Several techniques to detect vacuity have been proposed (e.g., [7,15]) and also tools that perform vacuity detection have been developed (e.g.[11]). The general strategy to detect vacuity employed in [7,15] is to replace parts of a property and see if this has any effects on the result of the verification. In order to detect vacuity it is sufficient to replace a sub-formula ϕ of property φ with **True** or **False** [15], depending on the polarity of ϕ in φ. The polarity of a sub-formula ϕ is positive, if it is nested in an even number of negations in φ, otherwise it is negative, and $pol(\phi)$ is a function such that $pol(\phi) =$ **False** if ϕ has positive polarity in φ and $pol(\phi) =$ **True** otherwise[2].

The replacement of sub-formula ϕ with ψ in formula φ is denoted as $\varphi[\phi \leftarrow \psi]$.

Definition 1 A property φ is completely/partially vacuous if for every/some of its atomic proposition ϕ, $\mathrm{VC}_\phi = \varphi[\phi \leftarrow pol(\phi)]$ is proved true by the model checking.

When the formula VC_ϕ is true, our tool reports the list of atomic propositions ϕ that make the property φ vacuously true.

```
MODULE main
VAR     request : boolean;
        state : {ready,busy};
ASSIGN  request := 0;
        init ( state ) := ready;
        next( state ) := case
                            state = ready & request : busy;
                            1: {ready,busy};
                         esac;
CTLSPEC AG(request -> AF state = busy)
```

Example 9 Violation of meta-property MP10

In the example 9, the CTL property is vacuously true for subformula $state = busy$. Indeed the CTL formula is $true$ regardless of the fact that $state$ is equal to *busy* or not, since *request* is always $false$.

[2] As in [15] we assume that all the occurrences of the subformula ϕ in φ are of a *pure polarity* (that is, they are either all under an even number of negations (positive polarity), or all are under an odd number of negations (negative polarity))

5 Meta-Property Verification by Model Checking

To verify (or falsify) the meta-properties introduced in the previous section, we translate each meta-property MP_k in a CTL property $MP_{k_{CTL}}$. We then build a new NuSMV model M_{MP} obtained by adding to the original model M the set $MP_{CTL} = \cup_{k=1}^{n} MP_{k_{CTL}}$ that contains the translations of all the meta-properties. The verification of the meta-properties is carried out through the model checking of M_{MP}.

The mapping from a meta-property to a CTL formula is not straightforward, because of *a*) the way CTL properties are verified in NuSMV, *b*) the fact that next_expressions, which can be used in some meta-properties, can not be contained in CTL formulas.

a) A CTL property ϕ is true if and only if ϕ is true in every initial state of the machine, i.e., given a model M and a property ϕ,

$$M \models \phi \quad \text{iff} \quad \forall s_0 \in S_0 \ \ (M, s_0) \models \phi$$

where S_0 is the set of initial states of M.

The operator $Always(\phi)$ is translated to $\mathtt{AG}(\phi)$. Indeed, $M_{MP} \models \mathtt{AG}(\phi)$ means that, along all paths starting from each initial state, ϕ is true in every state (globally), which corresponds to the definition of *Always*. Similarly, $InitiallyA(\phi)$ is translated as ϕ, since $M_{MP} \models \phi$ means that in each initial state ϕ is true, which corresponds to the definition of *InitiallyA*. However, the translation of $Sometime(\phi)$ is not $\mathtt{EF}(\phi)$, since $M_{MP} \models \mathtt{EF}(\phi)$ means that there exists at least one path starting from *each* initial state containing a state in which ϕ is true, while *Sometime* requires only that there exists *at least* an initial state from which ϕ will eventually hold. This means that there are cases in which $\mathtt{EF}(\phi)$ is false, since not from every initial state ϕ will eventually be true, while $Sometime(\phi)$ is true. To prove $Sometime(\phi)$ we use the following equivalence:

$$M_{MP} \models Sometime(\phi) \quad \Leftrightarrow \quad M_{MP} \not\models \mathtt{AG}(\neg\phi)$$

that means that $Sometime(\phi)$ is true if and only if $\mathtt{AG}(\neg\phi)$ is false. We run the model checker with the property $P = \mathtt{AG}(\neg\phi)$ and if a counter example of P is found, then $Sometime(\phi)$ holds, while if P is proved true, then $Sometime(\phi)$ is false. Similarly, to prove *InitiallyS* we use the equivalence:

$$M_{MP} \models InitiallyS(\phi) \quad \Leftrightarrow \quad M_{MP} \not\models \neg\phi$$

b) It is possible that a meta-property contains `next_expressions` [3]. In NuSMV such expressions can not be contained in CTL formulas, but they can occur in *invariant specifications*. Invariant specifications are propositional formulas which must hold invariantly in the model, and are expressed

[3] The meta-properties that are defined through the *Always* or the *Sometime* operators (MP1, MP2, MP3, MP4 and MP5 when are applied to the next assignments) can contain the **next** operator.

as "**INVARSPEC** next_expr;". They are equivalent to "**CTLSPEC AG** simple_expr;" and can be checked by a specialized algorithm during reachability analysis.

In conclusion, all the CTL formulas obtained by the translation described previously, that have the form $MP_{k_{CTL}} = \texttt{AG}(\varphi)$, with φ containing next_expressions, are checked as invariant specifications. All the other CTL formulas, instead, are checked as CTL specifications.

6 NuSMV Model Advisor

We have implemented a prototype tool, available at [3], written in Java to automatixe the model review procces. The tool is built on top of the NuSVM model checker and required to develop a new parser to represent the structure of a NuSMV specification in terms of Java navigable objects that could be visited to compute the assignment condition functions and to access other internal syntactical specification elements. To the purpose of developing this new parser, the Xtext [4] framework was used. It allows the development of language infrastructures including compilers and interpreters as well as full blown Eclipse-based IDE integration. The user must only provide an EBNF grammar of his language. Starting from this grammar, the XTEXT generator creates a parser, a language meta-model (implemented in EMF) as well as a full-featured Eclipse-based editor.

For our purposes, we have written the EBNF grammar of NuSMV and through Xtext we have obtained a NuSMV parser. Parsing a model, an EMF model of the NuSMV specification is built which allows accessing the structure of the NuSMV model (otherwise accessible by constructing its abstract syntax tree).

The model advisor works in the following way:

1. the model M one likes to review is parsed by the NuSMV parser provided by the model checker; if M is not parsed correctly the tool does not execute any verification and quits, otherwise it continues as follows;
2. the model M is parsed with our NuSMV parser and an EMF model of M is internally represented;
3. the CTL properties MP_{CTL} needed for the verification of the Meta-Properties are built as described in Section 5, as well as the NuSMV model $M_{MP} = M + MP_{CTL}$ obtained from the original specification M with the CTL meta-properties;
4. the tool runs the specification M_{MP} with the model checker NuSMV and reads the output of the execution;
5. it interprets the MP_{CTL} verification results and builds the meta-properties results that are finally printed (on the screen or on a file).

7 Experimental results and discussion

We have applied our model review process to three different sets of NuSMV specifications. The first set, `NuSMVsrc`, contains the NuSMV examples available in the NuSMV source distribution: some of these examples are also available in the example page on the NuSMV site [1]. The `Internet` set contains various models that we have found on the Internet: research works, students projects, etc ... The last set, `AsmetaSMV`, contains the models obtained with the tool AsmetaSMV [5], a tool that translates ASM models in NuSMV models. This last set of examples was chosen to assess the quality of NUSMV models obtained from models developed using other formal (high level) notations. Indeed, NuSMV is often used as a target language for model checking specification originally developed using other formal methods. By translating these other models to NuSMV, the NuSMV code might not be efficient and redundancies might be introduced.

The results of our experiments are reported in Table 1. It shows the name of the set, the number of models in it, the number of models which we were able to analyze[4] and, for each meta-property, the number of violations we detected.

The most violated meta-property is MP6, that is that a variable does not take all the values of its domain[5]. Simply removing the unused values of the variables type (if they are really not necessary) can dramatically improve the model performances.

The second most violated property is MP3, that requests that two conditions are always mutually exclusive. When a couple of conditions $(cond_1, cond_2)$ violates this property, the first condition $cond_1$ masks the second one $cond_2$: sometimes the developer is conscious of this behavior, but sometimes he is not.

The third most violated property is MP4, that requests that the default condition, if specified, is never taken; this meta-property is very strong: developers, indeed, often use the default condition to catch some situations not captured by the previous conditions. We must remember that our meta-properties do not signal errors, but violations of some modeling guidelines that the developer would like to follow.

Finally we would like to underline that the violations of meta-properties MP2 and MP1 (the third and the fourth most violated meta-properties) signal erroneous models where, respectively, some conditions are never applied and some conditions are never satisfied. We can notice that MP1 and MP2, although similar, are not the same meta-property: more precisely all MP1 are also MP2, but not viceversa. There are

[4] Some models could not be analyzed because *a*) they were wrong, that is they did not parse with the NuSMV parser (33 models) *b*) the verification of their meta-properties could have been longer than one hour, the execution time limit we have set (20 models).

[5] The high number of violations is also due to the fact that each value not taken is a violation (the number of variables that do not take all their values is shown in round brackets).

Spec Set	# spec.	# rev.	# not rev.	MP1	MP2	MP3	MP4	MP5	MP6	MP7	MP8	MP9	MP10
`NuSMVsrc`	63	47	3 - 13	178	230	882	683	44	120 (47)	7	3	42	44
`Internet`	187	151	30 - 6	209	261	392	351	104	2201 (105)	12	8	147	184
`AsmetaSMV`	34	33	0 - 1	94	121	20	151	34	215 (150)	0	0	1	22
total	284	231	33 - 20	481	612	1294	1185	182	2536 (302)	19	11	190	250

Table 1 Experimental results and violations found

assignments that are never executed, whose conditions are eventually satisfied: those assignments are never executed because their conditions are masked by some previous conditions.

Violations in the AsmetaSMV set deserve a particular remark. As expected the *AsmetaSMV* set contains several violations concerning the minimality of the model since these models are obtained using NuSMV as target language to model checking ASM models. Therefore, the high number of MP6 violations was expected. However, the violations of property MP3 (20 violations in 5 models) is, at a first sight, surprising. The tool AsmetaSMV, indeed, should always produce conditions mutually exclusive. We have discovered that this violations are produced by ASM models containing inconsistent updates, namely parallel updates, in the same state, of the same location (variable in NuSMV) to two different values. This proves that the analysis done at the level of NuSMV can give insights about the high level starting models. In the future, we plan to integrate our NuSMV model advisor with the AsmetaSMV tool, in order to obtain minimal models from the translation of the ASM models and check for model consistency.

Fault detection capability An important question about the technique we propose is what kind of faults it can reveal. Although a violation of a meta-property does not necessary mean that the specification is faulty, it is important to link the automatic analysis we perform to possible faults in the specifications for two reason: (1) to be sure that the meta-properties actually measure the quality of the specification also in terms of its correctness (which can be ultimately considered as absence of faults) and (2) to provide useful feedback to the user to suggest which kinds of faults can occur in the specification given a violation of a specific meta-property. We have identified the following defects:

– *Over specification or missing use of variables* is detected by meta-properties like MP7 and MP8 or by MP6 which checks that variable domains are used. These meta-properties aim at detecting faults either of over specifications, i.e. useless details are added to the model, or of omission, i.e. variables that should occur in conditions or expression but are simply forgotten.

– *Faults in assignments* can be detected by MP5.

– *Missing or misplaced conditions* can be detected by MP1 and MP2. Indeed if conditions are placed in a wrong order, then an assignment can be masked and this is signaled by our meta-properties. MP3 and MP4 try to prevent this kind of faults by making the conditional assignment independent on the order of the conditions.

– *Wrong or inaccurate properties* are detected by MP9 and MP10.

8 Related work

Typical automatic reviews of formal specifications include simple syntax checking and type checking. This kind of analysis is performed by simple algorithms which are able to immediately detect faults like wrong use of types, misspelled variables, and so on. Some complex type systems may require proving of real theorems, like the non-emptiness of PVS types [17].

As already anticipated in the introduction, a similar model reviewing technique was presented in [6] for the Abstract State Machine (ASM) formal specifications [9]. The review we propose in this paper is also similar to the kind of reviews proposed by Parnas and his colleagues. In a report about the certification of a nuclear plant, he observed that "reviewers spent too much of their time and energy checking for simple, application-independent properties" (like our meta-properties) which distracted them from the more difficult, safety-relevant issues." [18]. Tools that automatically perform such checks can save reviewers considerable time and effort, liberating them to do more creative work.

Our approach has been greatly influenced by the work done by the group lead by Heitmeyer with the Software Cost Reduction (SRC) method. SCR features a tabular notation which can be checked for *completeness* and *consistency* [12]: completeness guarantees that each function is totally defined by its table and consistency guarantees that every value of controlled and internal variables is uniquely defined at every step. In [13] it is described a method, similar to ours, to automatically verify the consistency of a software requirements specification (SRS) written in an SCR-style; properties that describe the consistency of the model are defined *structural properties*. The SRS document is translated into a PVS model where, for each structural property, a PVS theorem is declared. The verification of structural properties is carried out through the proof of PVS theorems and, for one property, through the model checking of a CTL property.

Other approaches try to apply similar analysis to non tabular notations. In [19], the authors present a set of robustness rules (like UML well-formedness rules) that seek to avoid common types of errors by ruling out certain modeling constructs for UML state machines or Statecharts. Structural checks over Statecharts models can be formulated by OCL constraints which, if complex, must be proved by theorem proving. Their work and ours extend the use of meta-properties not only to guarantee correctness but also to assure high quality standards in case the models are to be used for safety critical applications.

Other approaches emphasize the analysis of the *properties* specified for a model. For instance, the problem of the vacuity and coverage of formal requirements is studied in [14]. In that paper, the authors note the importance of suspecting the system or the specification of containing an error also in the case model checking succeeds and they proposed two sanity checks: vacuity (equal to our MP10) and coverage. In coverage, the goal is to check if components of the system are superfluous and it shares the same intent of our meta-properties MP1 and MP2.

In [8], the authors propose a methodology with the goal of performing the quality assurance of formal specifications. Their methodology is supported by a tool (RAT - requirement analysis tool) and based on two techniques: property simulation and property assurance. Property assurance has the goal of assessing the completeness and the consistency of formal specification but requires the introduction of assertions (which must be satisfied) and possibilities (which describe allowed corner-case behavior). Using assertions, a designer can check whether the requirements are strict enough to exclude any undesired behavior. With possibilities, one can check that they are not overly strict, and desirable behavior is allowed.

9 Conclusions and Future work

We have presented a method to perform automatic model review of NuSMV specifications and a model advisor as tool support for this activity. This process has the aim to guarantee certain quality attributes of models. Given quality attributes are captured by meta-properties expressed in terms of CTL formulas. The NUSMV model checker itself is used to detect possible violation of these meta-properties and, therefore, the presence of possible defects in the model. These meta-properties can be assumed as measures of model quality assurance.

In the future, we plan to study new meta-properties and integrate our model advisor to the Asmeta toolset [2] (a set of tool for ASM specifications) in order to achieve minimality of the NuSMV models obtained by means of AmetaSMV model checker from ASM specifications.

Acknowledgements We would like to thank Siamak Haschemi for the initial version of the XTEXT for NuSMV grammar.

References

1. The NuSMV website. `http://nusmv.itc.it/`.
2. The ASMETA website. `http://asmeta.sourceforge.net/`, 2010.
3. The NuSMV model advisor site. `http://code.google.com/p/nusmvmodeladvisor/`, 2010.
4. The Xtext website. `http://www.eclipse.org/Xtext/`, 2010.
5. P. Arcaini, A. Gargantini, and E. Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second Inter. Conference, ABZ 2010*, volume 5977 of *Lecture Notes in Computer Science*, pages 61–74. Springer, 2010.
6. P. Arcaini, A. Gargantini, and E. Riccobene. Automatic review of abstract state machines by meta property verification. In C. Muñoz, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215*, pages 4–13, Langley Research Center, Hampton VA 23681-2199, USA, April 2010. NASA.
7. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Proc. 9th International Computer Aided Verification Conference*, number 1254 in Lecture Notes in Computer Science, pages 279–290, 1997.
8. R. Bloem, R. Cavada, I. Pill, M. Roveri, and A. Tchaltsev. Rat: A tool for the formal analysis of requirements. In W. Damm and H. Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 263–267. Springer, 2007.
9. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.
10. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*. Springer, July 2002.
11. M. Gheorghiu and A. Gurfinkel. Vaquot: A tool for vacuity detection. In *Posters & Research Tools Track, FM 2006*, 2006.
12. C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
13. T. Kim and S. D. Cha. Automated structural analysis of SCR-style software requirements specifications using PVS. *Softw. Test, Verif. Reliab*, 11(3):143–163, 2001.
14. O. Kupferman. Sanity checks in formal verification. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 2006.
15. O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, 2003.
16. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.
17. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE-11)*, pages 748–752, London, UK, 1992. Springer-Verlag.
18. D. L. Parnas. Some theorems we should prove. In *HUG '93: 6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 155–162, London, UK, 1994. Springer-Verlag.
19. S. Prochnow, G. Schaefer, K. Bell, and R. von Hanxleden. Analyzing robustness of UML state machines. In *Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES 06)*, 2006.