NuSeen: a tool framework for the NuSMV model checker

Paolo Arcaini
Charles University
Faculty of Mathematics and Physics, Czech Republic
Email: arcaini@d3s.mff.cuni.cz

Angelo Gargantini
DIGIP
University of Bergamo, Italy
Email: angelo.gargantini@unibg.it

Elvinia Riccobene
Department of Computer Science
University of Milan, Italy
Email: elvinia.riccobene@unimi.it

Abstract—**NuSMV is a well-known tool for system verification that permits to verify both CTL and LTL properties. Although the tool is very powerful, it offers a minimal support for the editing and validation (e.g., by simulation) of models and of requirements specified as temporal properties. In this paper, we propose NuSeen, a framework that assists a designer during the modeling and V&V activities when using NuSMV. In addition to an editor furnished with syntax highlighting, autocompletion, and outline, NuSeen also provides some tools for visualizing the variable dependencies, and graphically visualizing the counterexamples. It helps the designer in validating the model by checking certain qualities like minimality and completeness. Moreover, the framework also provides facilities for model-based testing by means of a test suite generator that is able to generate tests achieving value and decision coverage for NuSMV models.**

I. Introduction

NuSMV [10], [19] is a symbolic model checker derived from SMV [18]. The NuSMV project aims at the development of a state-of-the-art symbolic model checker, designed to be applicable in technology transfer projects: it is a well structured, open, flexible and documented platform for model checking, and is robust and close to industrial systems standards [9]. NuSMV has a rich and powerful language to describe complex systems: the specification of the system behaviour is given in terms of finite state machines, and its expected requirements are specified as temporal formulas. Property verification is supported by a BDD-based symbolic model checker, and a bounded model checker.

NuSMV is used more and more, not only as simple model checker, but also as a tool supporting the validation and verification (V&V) activities of complex systems, from requirements capture to their validation and verification. For this reason, besides the essential model checker, designers should benefit from other accompanying tools like editors, visualizers, static and dynamic analyzers, and so on. When performing system design and verification, particular attention should be put on the correctness of the model itself and on the properties we want to prove: If the model and/or the properties are not correct, any verification result is useless. Indeed, there is the risk of proving either wrong properties for correct specifications or correct properties for wrong specifications, where correctness means that it captures the intended behaviour. In our experience in teaching NuSMV to master students, we observed that this risk is real. In order to assist the designers in modeling with NuSMV, we have developed in the last few years the NuSeen[1] framework that provides tools for editing, visualizing, and validating NuSMV models.

The framework provides basic functionalities for *model editing* as syntax highlighting, autocompletion, and outline. Execution of models is integrated in the framework, allowing to set all the execution options that are allowed by NuSMV, but in a more friendly way. Moreover, NuSeen provides a *graphical tabular representation* of the counterexamples returned by the model checker, that eases the process of counterexample inspection.

The framework further provides two additional tools for model validation. The *model advisor* [2] permits to check some particular *quality* attributes that any NuSMV model should have as: all the variables are read, all the updates are executed, the properties are not vacuously satisfied, and so on. Moreover, another tool permits to graphically visualize (by means of a dependency graph) the dependencies among the variables of the model; this allows the designer to debug its model. The *dependency visualizer* also permits to visualize the strongly connected components of variables, i.e., variables that all depend on each other: such visualization is particular useful when we want to divide the model in different modules.

Model checkers as NuSMV are not only used for verification, but they are also frequently used for test case generation [14]. NuSeen provides a *test case generator* for NuSMV models [4]; it permits to generate tests achieving value and decision coverage.

Sect. II briefly introduces the NuSMV notation, and Sect. III presents all the tools of the NuSeen framework. Sect. IV describes how the framework has been implemented, Sect. V discusses some lessons we have learned in developing and using NuSeen, and Sect. VI concludes the paper.

II. NuSMV notation

NuSMV [10], [19] is a well-known tool that performs symbolic model checking. It permits to represent synchronous and asynchronous finite state systems, and supports the verification of specifications expressed in *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL).

[1]NuSeen is freely available and open sourced at http://nuseen.sourceforge.net/

A NuSMV model describes the behaviour of a Finite State Machine (FSM) in terms of a "possible next state" relation between states. A model must have a **VAR** section containing variable declarations; possible variable types are boolean, integer defined over intervals or sets, or an enumeration of symbolic constants. The model state is given by the assignment of values to variables. State transitions are determined by variables updates declared in the **ASSIGN** section; the variable value can be determined in the initial state by the instruction **init**, and in the next state by the instruction **next**. Alternatively (instead of the **init/next** instructions), an expression can be used to define the value of a variable in each state. A **DEFINE** statement can also be used as a macro to syntactically replace an *alias* with the *expression* it is associated with. The syntax of these four commands is as follows:

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

where *identifier* is a variable identifier, *simple_expression*s can only refer to variables in the current state, while *next_expression* can refer to variables both in the current and in the next state. Accessing the value of a variable in the next state can be done with the **next** operator[2].

Expressions can contain conditions. Conditional expressions are:

- **if-then-else** expression *cond? exp1: exp2* which evaluates to *exp1* if the condition *cond* evaluates to true, and to *exp2* otherwise.
- **case** expression:

  ```
  case
    leftExpression_1 : rightExpression_1;
    ...
    leftExpression_n : rightExpression_n;
  esac
  ```

  where left expressions must be boolean, and the right expressions must all have the same type. The case expression returns the value of the first *rightExpression*$_i$ whose *leftExpression*$_i$ condition evaluates to true, and the previous i-1 left expressions evaluate to false. If all the left expressions evaluate to false, an error occurs. To avoid these kinds 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 *leftExpression* equal to TRUE.

An alternative way of defining initial states and the transition relation is by means of **INIT** and **TRANS** constraints. An **INIT** constraint is a boolean simple expression (i.e., not containing the **next** operator) that must be satisfied by the variables values in the initial states; a **TRANS** constraint, instead is a boolean next expression (i.e., it can contain the **next** operator) describing the relation between variables in the current and in the next state. It is also possible to define invariant constraints in the **INVAR** section.

[2]See the NuSMV User Manual [7] for more details on the assignment syntax and restriction rules for assignments

```
MODULE main
VAR
  request: boolean;
  state: {ready, busy};
ASSIGN
  init(state) := ready;
  next(state) :=
    case
      state = ready & request: busy;
      state = busy: {ready, busy};
      TRUE: ready;
    esac;

CTLSPEC AG(request −> AF(state = busy))
```

Code 1: NuSMV model of a simple producer

CTLSPEC and **LTLSPEC** allow to specify CTL and LTL temporal specifications to be verified. Properties can be named (by keyword *NAME*) in order to ask the model checker to check only a particular property.

Code 1 shows a simple NuSMV model of a producer (adapted from an example of the NuSMV tutorial [19]). Variable *state* models the state of the producer that can be *busy* in producing goods or *ready* to accept new requests. Boolean variable *request* models the fact that a request has been submitted to the producer. The *state* of the producer is initially *ready*. When it receives a request while *ready*, it becomes *busy*; when it is *busy*, it can nondeterministically decide to remain *busy* or become *ready* again; otherwise, when it is *ready* and there is no *request*, it remains *ready*. A temporal property checks that, whenever there is a *request*, the *state* of the producer will be eventually *busy* in handling it.

III. NuSeen: a NuSMV Eclipse-based ENvironment

NuSeen is a series of tools, integrated within the eclipse IDE, that aim at helping NuSMV users during their V&V activities. It focuses in easing the use of the NuSMV by means of graphical elements like buttons, menu, text highlighting, and so on. It provides also some auxiliary tools that show model information by means of graphical elements (like dependency graphs, tables, etc.). Moreover, it helps the designer in the reuse of formal models for other purposes, like documentation and testing of the real system. It features:

- A *language* defined by a grammar (concrete syntax) and provided with a metamodel (abstract syntax), with a rich (automatically generated) Java APIs useful to access NuSMV models.
- An *editor* that can be used to write NuSMV models and provides a useful feedback, like syntax highlighting, autocompletion, and outline.
- A way to *execute* the NuSMV model checker inside eclipse (in a batch mode or interactively).

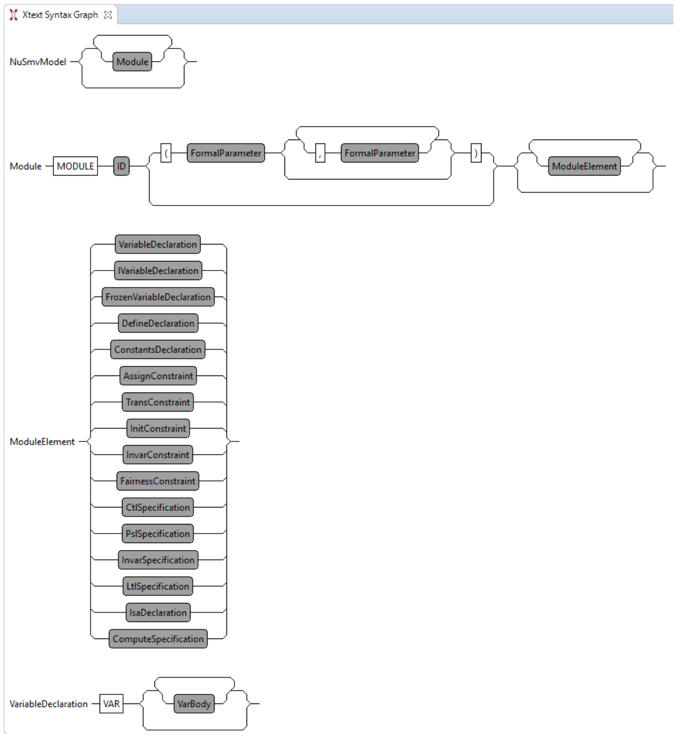
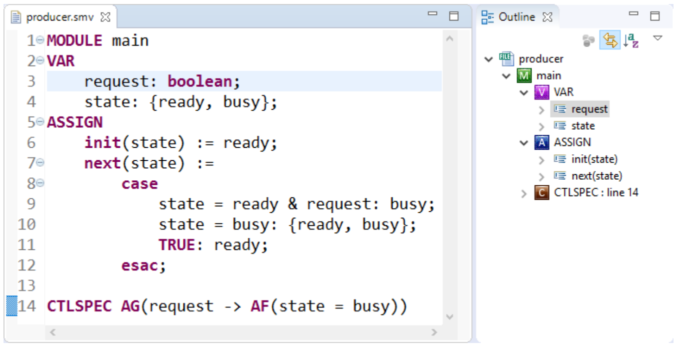


Figure 1: Syntax graph of the NuSeen grammar



```
MODULE main
VAR
    request: boolean;
    state: {ready, busy};
ASSIGN
    init(state) := ready;
    next(state) :=
        case
            state = ready & request: busy;
            state = busy: {ready, busy};
            TRUE: ready;
        esac;

CTLSPEC AG(request -> AF(state = busy))
```

Figure 2: Editor and outline

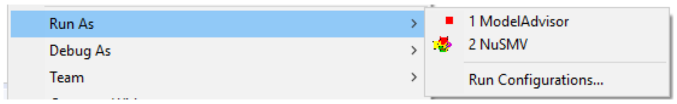


Figure 3: Executing NuSMV models

- A counterexample visualizer that shows traces in a more friendly manner than the simple text (or XML) produced by NuSMV.
- An integrated version of a model advisor, which can be executed in eclipse, to check model quality properties.
- A dependency analyzer that, by means of simple graphs, helps in understanding how the model is structured and how it can be modularized.
- A test case generation that permits to exploit NuSMV for model-based testing.

A. *Language and editor*

NuSeen introduces the definition of the language and of the editor for NuSMV by Xtext [11], [22]. Xtext is a framework for the development of programming languages and domain specific languages. It covers all aspects of a complete language infrastructure, from parsers, linker, compiler or interpreter, to fully-blown top-notch eclipse IDE integration. It comes with good default solutions for all these aspects and, at the same time, every single aspect can be tailored to the user's needs. We have defined a grammar for the NuSMV notation in the Xtext format. Fig. 1 shows an excerpt of the grammar syntax graph. From the grammar, Xtext automatically produces:

1) a *meta-model* for the language in the form of an EMF (eclipse metamodeling framework) model in the ecore format. It represents the *abstract* syntax of our language in terms of classes and relations. From the metamodel, we automatically obtained the Java APIs that are able to manage (query, create, and modify) NuSMV models and parts of them. This allows the integration of NuSMV with other Java-based tools.
2) an *editor* with syntax coloring based on the lexical structure. Content assist that proposes valid code completions at any place in the document, helping users with the syntactical details of the language. Other features are validation and quick fixes, linking of errors, the outline view, etc.

The editor can be easily extended in order to support extra semantic validation rules, particular rules for indentation and outlining, and other ad hoc editing rules.

NuSeen provides a special type of projects (i.e., `NuSMV Project`) where to add the NuSMV models. Moreover, it provides a wizard that creates a simple specification.

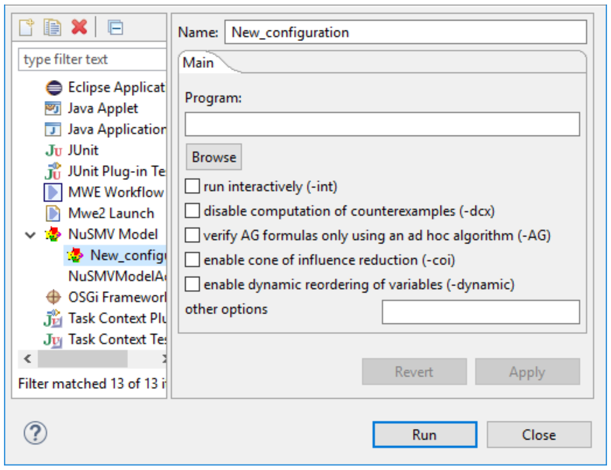
Fig. 2 shows the model editor and the outline view for the model presented in Code 1.

B. *Running NuSMV models*

NuSMV models can be run in batch mode or in interactive mode. In batch mode, all the temporal specifications are checked and the result is shown in the eclipse console. In interactive mode, instead, the user is provided with a shell from which (s)he can perform different activities: simulate the model, verify particular properties, set options, etc.

In NuSeen, the model is executed through the `Run As` extension of eclipse, as shown in Fig. 3. If the user does not specify any configuration, the model is executed in batch mode. Otherwise, the user can set some execution options as shown in Fig. 4. Some of the available options are:

- *run interactively*: the NuSMV interactive shell is open in the eclipse console. The user can perform the same actions (s)he can do in the classical interactive shell of NuSMV.
- *disable computation of counterexamples*: the computation of counterexamples is disabled and only whether the property is true or false is reported to the user.



type filter text

- Eclipse Applicat
- Java Applet
- Java Applicatior
- JUnit
- JUnit Plug-in Te
- MWE Workflow
- Mwe2 Launch
- NuSMV Model
  - New_config
- NuSMVModelA
- OSGi Framewor
- Task Context Plu
- Task Context Te

Filter matched 13 of 13 i

Name: New_configuration

Main

Program:

Browse

- [ ] run interactively (-int)
- [ ] disable computation of counterexamples (-dcx)
- [ ] verify AG formulas only using an ad hoc algorithm (-AG)
- [ ] enable cone of influence reduction (-coi)
- [ ] enable dynamic reordering of variables (-dynamic)

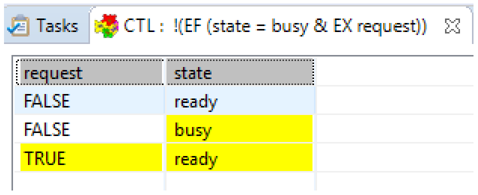
other options

Revert Apply

Run Close

Figure 4: Configuration of NuSMV execution

Figure 5: NuSeen toolbar



Tasks | CTL : !(EF (state = busy & EX request))

request	state
FALSE	ready
FALSE	busy
TRUE	ready

Figure 6: Counterexample visualization

- *enable cone of influence reduction*: NuSMV applies the cone of influence reduction technique [6] (i.e., variables that are not involved in the verified properties are removed from the model) that can speed up the verification process.
- Other NuSMV options as *dynamic* and *AG* can be selected through checkboxes. Other less common options, can be specified by the user in the final textbox.

C. Counterexample visualizer

One of the strengths of the model checker is its capability of generating a counterexample whenever a property is proved false. This makes model checking an efficient *bug hunting* technique and not only a correctness verifier. A counterexample is a valid system trace for which the temporal property is violated. By looking at the counterexample, the designer can understand better if either the bug is in the system behaviour or in the temporal property. Moreover, designers often introduce a *false* property p in order to check that the actual behaviour of the system violates p and to observe the trace that falsifies p. However, understanding and navigating a trace can be quite difficult. For this reason, NuSeen provides a counterexample visualizer that can be invoked by a button in the NuSeen toolbar (first button in Fig. 5). The tool shows the trace in a tabular format as shown in Fig. 6. In the example, given the model from Code 1, the user is interested in finding a path in which the producer is $busy$ and, in the next state, it receives a request; in order to obtain such a path, (s)he specifies the CTL property !**EF**(state = busy & **EX**(request)) stating that there is no future state in which $state$ is $busy$ with a possible next state in which $request$ is true. The model checker finds that the property is false and the returned counterexample is shown as in Fig. 6. The counterexample is presented in a compact way as a table where the columns are the variables and the rows are the states; changes from the previous state in variable values are displayed in yellow. Note that such representation is much more readable than the standard counterexample representation of NuSMV in which only the variables that changed their value are shown: in that visualization, the user has to scroll the counterexample to retrieve the information about the whole state.

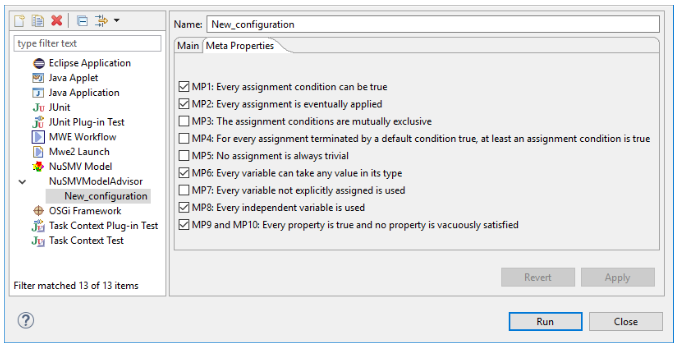
D. Model advisor

Before verifying any property, the user should validate the model itself. In [2], we devised some *quality attributes* that any NuSMV model should have. These quality attributes are expressed by the following predicates, called *meta-properties* (MP):

MP1: Every assignment condition can be true.
MP2: Every assignment is eventually applied.
MP3: The assignment conditions are mutually exclusive.
MP4: For every assignment terminated by a default condition TRUE, at least one assignment condition is true.
MP5: No assignment is always trivial.
MP6: Every variable can take any value in its domain.
MP7: Every variable not explicitly assigned is used.
MP8: Every independent variable is used.
MP9: Every property is proved true.
MP10: No property is vacuously satisfied.

The satisfaction of these MPs guarantees the required quality attributes; therefore, MPs can be assumed as a measure of the model quality. We can divide the MPs in three main categories.

- *Consistency* requires that there are no conflicting model statements (variable assignments, propriety specifications, behaviours, etc.). MP3 and MP9 belong to this category.
- *Completeness* requires that every system behaviour is explicitly modeled. The category contains MP7 requiring the explicit assignment of variables, and MP4 requiring that at least one assignment condition (apart from the default condition) is true.
- *Minimality* requires that the model does not contain elements (variables, assignments, domain elements, etc.) defined or declared in the model but never used, i.e., it is not *over-specified*. The category contains MP1 and MP2 requiring that every assignment can be performed, and MP5 requiring that each assignment is really useful. Moreover, it contains MP6 requiring that every value


Name: New_configuration

Main | Meta Properties

- [✓] MP1: Every assignment condition can be true
- [✓] MP2: Every assignment is eventually applied
- [ ] MP3: The assignment conditions are mutually exclusive
- [ ] MP4: For every assignment terminated by a default condition true, at least an assignment condition is true
- [ ] MP5: No assignment is always trivial
- [✓] MP6: Every variable can take any value in its type
- [ ] MP7: Every variable not explicitly assigned is used
- [✓] MP8: Every independent variable is used
- [✓] MP9 and MP10: Every property is true and no property is vacuously satisfied

Figure 7: Model advisor – Selection of meta-properties

```
MODULE main
VAR
  request: boolean;
  state: {ready, busy};
ASSIGN
  init(state) := ready;
  request := FALSE; -- fault
  next(state) :=
    case
      state = ready & request: busy;
      state = busy: {ready, busy};
      TRUE: ready;
    esac;

CTLSPEC AG(request -> AF(state = busy))
```

Code 2: Faulty version of the NuSMV model of the producer

in the domains is necessary, and MP7[3] and MP8 requiring that every variable is used. Finally, also MP10, checking that property specifications are not vacuously satisfied [17], belongs to this category.

The model advisor analyzes a NuSMV specification using NuSMV itself. Each meta-property is encoded as a suitable CTL property whose satisfaction means that the corresponding meta-property holds. The NuSMV models are read and queried by the model advisor by using the API generated from the metamodel introduced by the language component. The selection of the meta-properties to check is shown in Fig. 7.

Code 2 reports a faulty version of the model shown in Code 1. The designer wrongly specified that request is always false (probably (s)he wanted to initialize it to false). Checking the model with the model advisor, we obtain the output shown in Fig. 8. Meta-property MP2 is violated because the update of variable *state* to *busy* is never executed; indeed the update can be performed only when there is a request. Moreover, since *state* never becomes *busy*, also the second update is never executed. Also meta-property MP10 is violated: the property is vacuously true because the antecedent of the implication is always false (i.e., there is never a request).

[3]Note that MP7 belongs to two categories, since it captures two different quality characteristics.

```
- MP2: Assignment in next assignments never applied -

Module main
in next(state) the following updates are never executed
cond = "state = ready & request"  value = "busy"
cond = "state = busy"  value = "{ready, busy}"

-----------------------------------------------------
...
------ MP10: No CTL property is vacuously true ------

Module main
Property "AG (request -> AF state = busy)" is true.
is vacuously true for phi = "state = busy"
is partially vacuous
```

Figure 8: Output of the model advisor

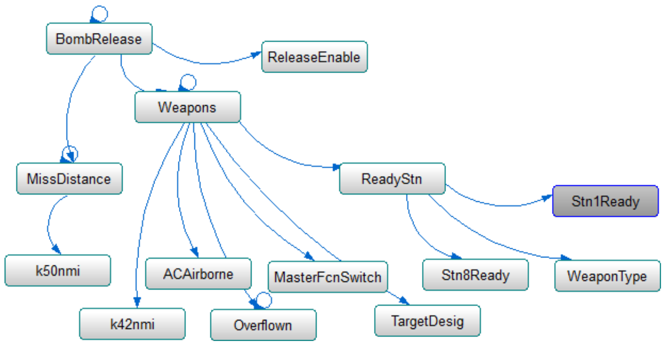


Figure 9: Variable Dependency Graph

E. Dependencies visualization

Variables of a NuSMV specification can be analyzed in order to discover their dependencies. We here first introduce the concepts of variables dependency and dependency graph (originally defined in [4]).

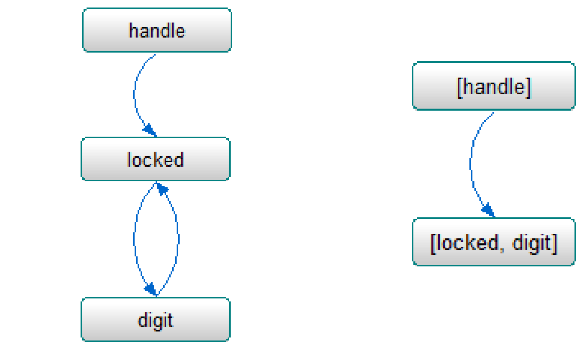
Definition 1 (Dependency): Given two variables v, w of a NuSMV model, we say that v *directly depends on* w if w (primed or not primed) occurs in the definition (**init**, **next**, or simple assignment) of v, or they both appear in the same **TRANS**, **INIT**, or **INVAR** constraint.

A variable is considered *primed* if it is accessed through the **next** operator.

Definition 2 (Dependency graph): We call *dependency graph* of a NuSMV model M the directed graph $DG = \langle V, E \rangle$, where V is the set of variables of M and $(v, w) \in E$ iff v directly depends on w.

The dependency graph can be used to better understand mutual dependencies among behaviours. NuSeen is able to build and show the dependency graph of a NuSMV model[4] as shown in Fig. 9. In the figure, the designer understands that *BombRelease* depends on *MissDistance*, *Weapon*, and *ReleaseEnable*. The inputs of the system are easy to identify, since they are the nodes in the graph that have no exiting edges (i.e, they do not depend on any other variable).

[4]It is a simplified specification of the bomb release requirements of an attack aircraft [13].



(a) Variable dependency graph (b) SCV dependency graph

Figure 10: Dependency graphs

Moreover, NuSeen is able to identify clusters of variables that depend one on the other and this can help the designer to better understand the subsystem of a model M and eventually to decompose M in different modules.

Definition 3 (Strongly Connected Variables set): Given a dependency graph $DG = \langle V, E \rangle$ of a NuSMV model M, each strongly connected component of DG identifies a *strongly connected variables set* (SCV).

Any two variables in an SCV depend one on the other. Intuitively, they constitute a group of interdependent quantities. Furthermore, some variables in an SCV may also directly depend on some variables of other SCVs. In this way, one can build the SCV dependency graph. Each SCV can be seen as a submodule and this decomposition can help the designer to decompose the system (in order to facilitate system analysis and test generation as, for example, in [4]).

NuSeen can build and show the SCV dependency graph of a NuSMV model, as shown in Fig. 10 for a small example taken from [4]. The variable dependency graph (Fig. 10a) shows that *locked* and *digit* depend one on the other. Thus, they constitute an SCV. The resulting SCV graph is shown in Fig. 10b.

The visualizer can be invoked from the NuSeen toolbar shown in Fig. 5: the **V** button shows the variable dependency graph, and the **M** button shows the SCV dependency graph.

NuSeen provides also some automatic layouts for graphs, like vertical and horizontal trees, radiants, grid, spring tree, in order to help the user in organizing the graphs visualization. For example, Fig. 11 shows a spring layout for an SCV dependency graph.

F. Test case generator

In model-based testing (MBT) [16], [21], the model describing the expected behaviour of the system is used for testing purposes. A classical MBT technique uses model checkers to produce abstract tests [12], [14], [15]; the technique exploits the capability of model checkers to produce counterexamples.

Definition 4 (Test): A *test* is a finite system execution.

Tests are usually generated to observe some particular system behaviours, called *testing goals*, formally represented by test predicates.

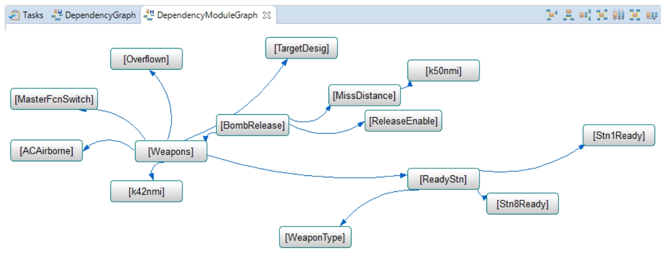


Figure 11: Dependency Graph in Spring Layout

Definition 5 (Test predicate): A *test predicate* is a temporal formula, and assesses whether the corresponding testing goal is reached.

A common pattern used to build test predicates is the LTL property $\mathbf{F}(\varphi)$, that requires that φ holds in some future state; φ is usually a predicate over the model state.

Test predicates are generated by coverage criteria.

Definition 6 (Coverage criterion): A *coverage criterion* C is a function that derives a set of test predicates from a formal model. A test suite TS satisfies a coverage criterion C if each test predicate generated with C is satisfied in at least one state of a test in TS.

Different coverage criteria for transition systems (as those described by NuSMV) have been proposed [1]. Those currently supported by NuSeen are:

- *value coverage*: each value of each variable is covered;
- *decision coverage*: each decision of each **init**, **next**, and simple assignment is covered once to true and once to false.

As future work, we plan to support other criteria as *condition coverage* and Modified Condition/Decision Coverage (*MCDC*) [8].

Example 1: The value coverage criterion applied to the NuSMV model shown in Code 1 produces the following test predicates: **F**(request), **F**(!request), **F**(state = ready), and **F**(state = busy).

The test generation with model checkers works as follows. For each test predicate tp, the *trap property* $\neg tp$ is verified. There are three possible outcomes of the verification:

- the trap property is false. This means that the test predicate tp is *feasible*; the returned counterexample is the test covering tp.
- the trap property is true. In this case, the test predicate tp is *unfeasible* and there is no test that can cover it.
- the model checker terminates without providing any result, usually because of the *state explosion problem* [6]. In this case, the user does not know whether the trap property can be covered or not. In [4], we have proposed a technique that mitigates the state explosion problem in test generation, using a decomposition technique based on the strongly connected variables sets presented in Def. 3.

The test generator for value and decision coverage can be invoked from the NuSeen toolbar shown in Fig. 5 (last two buttons). Given a model and a coverage criterion, the generator

```
- State 1 -
request = FALSE
state = ready
- State 2 -
request = TRUE
state = ready
- State 3 -
request = FALSE
state = busy
```

Figure 12: Test for test predicate **F**(state = busy)

builds all the test predicates and generates a test for each feasible test predicate. Each test is saved in a different text file (having the name of the test predicate) in a folder having the name of the model. The list of unfeasible test predicates is reported in a separate file. As an example, Fig. 12 shows the test produced for covering the test predicate **F**(state = busy) over the model shown in Code 1. We can see that the original test goal (i.e., observing *state* with the value *busy*) is achieved in the last state of the test.

Note that the test reported in Fig. 12 covers all the test predicates generated for value coverage (i.e., those reported in Ex. 1). As future work, we plan to integrate monitoring techniques in the tool, in order to avoid generating tests for test predicates that are already covered by some previously generated test.

IV. NuSeen implementation

A preliminary description of the framework architecture was presented in [5]. NuSeen is divided in 6 main components, as shown in Fig. 13, each divided in one or more plugin eclipse projects:

1) The component containing the definition of the language and the editor (dsl by Xtext), which is divided into two plugins: one contains the definition of the grammar and the other the classes for the editor.
2) The component containing the runner, which simply introduces the launching framework for the execution of NuSMV.
3) The dependency visualizer that depends on the language module (since it analyzes NuSMV models) and on two other external libraries, namely ZEST for graph visualization and JGraphT for graph algorithms (like the Tarjan algorithm used to discover SCVs).
4) The model advisor which contains the model advisor itself, its UI part in order to integrate it within eclipse and NuSeen, and a third project which allows the user to call the model advisor from outside eclipse. This latter plugin wraps everything needed by the model advisor in an executable jar file.
5) The test generator module which depends on both the NuSMV language and the NuSMV runner (since it uses NuSMV itself for test generation).
6) The counterexample visualizer that takes a counterexample as produced by NuSMV and builds the graphical table.

We have taken full advantage of the eclipse Plug-in Development Environment (PDE) by defining buttons, actions, and views as *extensions* of proper eclipse extension points. For instance, toolbar buttons use the extension point *org.eclipse.ui.menus*. To run NuSMV and the model advisor, we rely on the eclipse launching framework. This has facilitated the integration of NuSeen within eclipse and gives the user a consistent UI.

NuSeen can be installed as eclipse plugin through the update site[5] or through the eclipse marketplace.

V. Lesson learned

The starting motivation of developing NuSeen has been having a NuSMV grammar that we could exploit for our work in validating NuSMV models by model review [2]; to this purpose, Xtext gave us the perfect tool to specify the grammar quickly, and we also obtained for free different other artifacts like the feature-rich editor. Our experience in using Xtext is very positive both as a mean for defining a Domain Specific Language together with its parser and for generating the EMF metamodels and the Java APIs useful to query and modify NuSMV models.

We have been using NuSeen in our courses on formal methods and on model-based testing where we teach NuSMV (for the last four years), and we observed that the students benefit from its use: for example, the simple syntactic check done in the editor allows saving some time in debugging models. We also use the model advisor during our lectures; we have observed that students tend to misunderstand (or forget) the semantics of the case expression (i.e., the first condition that evaluates to true is selected): using the model advisor, they can easily check (by meta-properties MP1, MP2, and MP3) if there are some guards that cannot be executed because *masked* by some previous guard.

With the same aim of assisting designers (and our students) in using NuSMV, NuSeen has been further improved by the counterexample visualizer and the dependency visualizer. We observed that the counterexample visualizer is particular useful, since students have sometimes problems in identifying the whole state by inspecting the classical textual NuSMV counterexample (particularly in the long ones). As a general lesson learned, we confirm that visualization can help designers in better understanding formal specifications.

The dependency analyzer, moreover, is a key component for model decomposition techniques. It has permitted to achieve a significant improvement in test case generation for decomposable systems [3], [4].

VI. Conclusion and future work

This paper presents a tool framework that assists a designer when using the NuSMV model checker. It provides functionalities for model editing (as syntax highlighting, autocompletion,

[5]http://svn.code.sf.net/p/nuseen/code/trunk/updatesite/

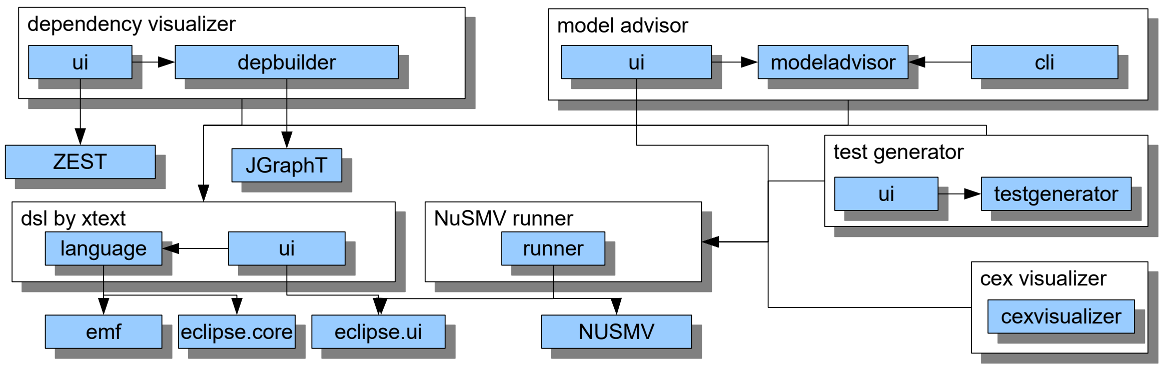


Figure 13: NuSeen architecture

and outline), and tools for model validation, namely a model advisor, a counterexample visualizer, and a visualizer of the dependency graph. In addition, it also supports model-based testing by a generator of abstract tests from the model.

Currently, we assume that the user has already installed NuSMV which must be in the system path. As future work, we plan to integrate the installation of NuSMV together with NuSeen, using technologies like JNA.

Right now, the output of the model advisor is shown in the console and the designer must find in the model the source of the meta-property violation. As future work, we plan to directly show the cause of a violation in the editor by using appropriate markers.

Regarding the test case generator, we plan to support additional coverage criteria and to implement a monitoring approach to minimize the size of the generated test suite. Currently, generated tests are meant for model-based testing, i.e., to be executed on the system implementation after a suitable concretization; however, they could also be used for validating the model itself: we plan to show them to the designer that should assess whether the tests are correct scenarios of the expected behaviour.

Finally, we plan to support the new infinite state model checker nuXmv [20].

VII. Acknowledgements

The research reported in this paper has been partially supported by the Czech Science Foundation project number 17-12465S.

We would like to thank Paolo Vavassori and several master students of the universities of Bergamo and Milano for their work on the NuSeen implementation.

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[2] P. Arcaini, A. Gargantini, and E. Riccobene. A model advisor for NuSMV specifications. *Innovations in Systems and Software Engineering*, 7:97–107, 2011.

[3] P. Arcaini, A. Gargantini, and E. Riccobene. An abstraction technique for testing decomposable systems by model checking. In M. Seidl and N. Tillmann, editors, *Tests and Proofs*, volume 8570 of *Lecture Notes in Computer Science*, pages 36–52. Springer International Publishing, 2014.

[4] P. Arcaini, A. Gargantini, and E. Riccobene. Improving model-based test generation by model decomposition. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 119–130, New York, NY, USA, 2015. ACM.

[5] P. Arcaini, A. Gargantini, and P. Vavassori. NuSeen: an eclipse-based environment for the NuSMV model checker. In E. Riccobene, editor, *Eclipse-IT 2013: Proceedings of VIII Workshop of the Italian Eclipse Community*, volume abs/1310.2464, 2013.

[6] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[7] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. NuSMV 2.5 User Manual. http://nusmv.fbk.eu/, 2010.

[8] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[9] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, Mar. 2000.

[10] A. Cimatti, E. M. 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 *Proceedings International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*. Springer, July 2002.

[11] M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[12] G. Fraser and A. Gargantini. An evaluation of model checkers for specification based test case generation. In *ICST 2009, 1-4 April 2009, Denver, Colorado, USA*, pages 41–50. IEEE Computer Society, 2009.

[13] G. Fraser, A. Gargantini, and F. Wotawa. On the order of test goals in specification-based testing. *The Journal of Logic and Algebraic Programming*, 78(6):472–490, 2009.

[14] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: A survey. *Softw. Test. Verif. Reliab.*, 19(3):215–261, Sept. 2009.

[15] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Softw. Eng. Notes*, 24(6):146–162, Oct. 1999.

[16] R. Hierons and J. Derrick. Editorial: special issue on specification-based testing. *Software Testing, Verification and Reliability*, 10(4):201–202, 2000.

[17] 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.

[18] K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[19] The NuSMV website. http://nusmv.fbk.eu/, 2017.

[20] nuXmv. https://es-static.fbk.eu/tools/nuxmv/.

[21] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.

[22] Xtext. http://www.eclipse.org/xtext/.