Carnegie Mellon
Software Engineering Institute

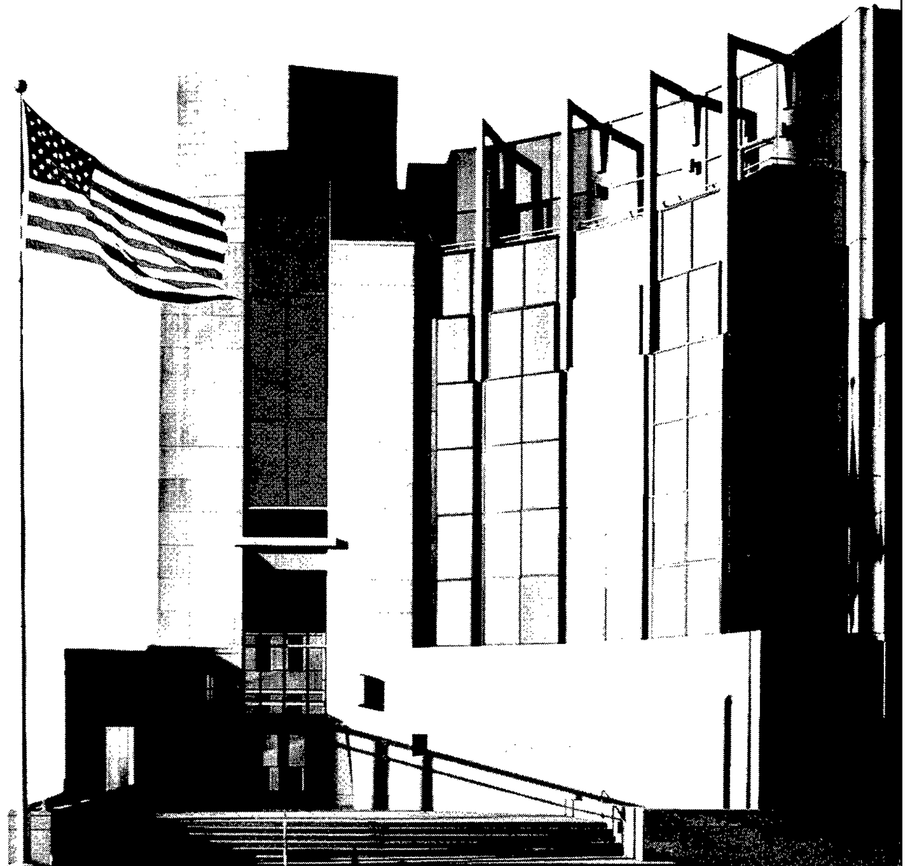
Model-Based Verification: Analysis Guidelines

Grace A. Lewis
Santiago Comella-Dorda
David P. Gluch
John Hudak
Charles Weinstock

December 2001

Performance Critical Systems Initiative

Technical Note
CMU/SEI-2001-TN-028



Model-Based Verification: Analysis Guidelines

Grace A. Lewis
Santiago Comella-Dorda
David P. Gluch
John Hudak
Charles Weinstock

December 2001

Performance Critical Systems Initiative

Technical Note
CMU/SEI-2001-TN-028

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Contents

List of Figures

Abstract

This technical note provides guidance for the analysis activity that occurs during the interpretation of results produced by model-checking tools. In the model-checking analysis activity, the main question is, "Does the system behave correctly?" To answer this question, a model and a set of expected properties are used as input to a model checker. The expected output is a confirmation or refutation of the specified expected properties. In most cases, if the model checker does not confirm the property, it provides a counterexample.

Counterexamples are executions of the model showing the sequence of steps that refutes the expected property. Sometimes the state space to be explored in order to find this counterexample is so large that it cannot be completely covered. This is the state explosion problem. Models must be tuned to reduce the state space; this is a manual and intuitive task.

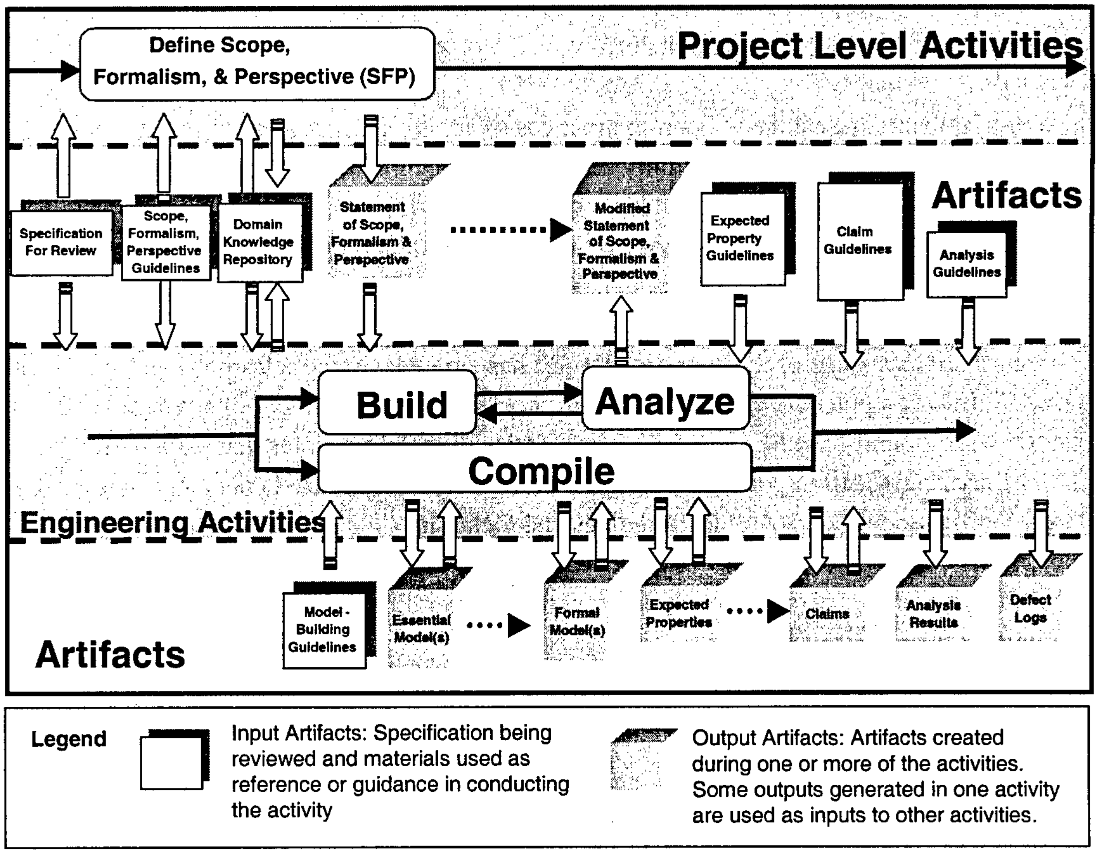
Interpreting the model checker's output can also be difficult. The significance of the output must be assessed; its interpretation may suggest an error in the claims or the model, or a defect in the actual system.

This document presents the problems related to interpreting results. It provides strategies to overcome state explosion, analyze results, and provide feedback to the system designers and developers.

1 Introduction

Model-Based Verification (MBV) is a systematic approach to finding defects (errors) in software requirements, designs, or code [Gluch 98]. The approach judiciously incorporates mathematical formalism, in the form of models, to provide a disciplined and logical analysis practice, rather than a "proof" of correctness strategy. MBV involves creating essential models of system behavior and analyzing these models against formal representations of expected properties.

The artifacts and the key processes used in Model-Based Verification are shown in Figure 1. Model building and analysis are the core parts of Model-Based Verification practices. These two activities are performed using an iterative and incremental approach, where a small amount of modeling is followed by a small amount of analysis. A parallel compile activity gathers detailed information on errors and potential corrective actions.



Define Scope, Formalism, & Perspective (SFP)
Project Level Activities
Artifacts
Specification For Review
Scope, Formalism, Perspective Guidelines
Domain Knowledge Repository
Statement of Scope, Formalism & Perspective
Modified Statement of Scope, Formalism & Perspective
Expected Property Guidelines
Claim Guidelines
Analysis Guidelines
Build
Analyze
Compile
Engineering Activities
Model-Building Guidelines
Essential Model(s)
Formal Model(s)
Expected Properties
Claims
Analysis Results
Defect Logs
Artifacts
Legend
Input Artifacts: Specification being reviewed and materials used as reference or guidance in conducting the activity
Output Artifacts: Artifacts created during one or more of the activities. Some outputs generated in one activity are used as inputs to other activities.

Figure 1: Model-Based Verification Process and Artifacts

An essential model is a simplified formal representation that captures the essence of a system, rather than provide an exhaustive, detailed description of it. Through the selection of only critical (important or risky) parts of the system and appropriately abstracted perspectives, a reviewer, using model-based techniques, can focus the analysis on the critical and technically difficult aspects of the system. Driven by the discipline and rigor required in the creation of a formal model, simply building the model, in and of itself, uncovers errors.

Once the formal model is built, it can be analyzed (checked) using automated model-checking tools such as SMV (Symbolic Model Verifier). Within this analysis, the user identifies potential defects both while formulating claims about the system's expected behavior and while formally analyzing the model using automated model-checking tools. Model checking has been shown to uncover the especially difficult-to-identify errors: the kind of errors that result due to the complexity associated with multiple interacting and inter-dependent components. These include embedded as well as highly distributed applications.

Many different formal modeling and analysis techniques are employed within MBV [Gluch 98, Clarke 96]. The choices are based upon the type of system being analyzed and the technological foundation of the critical aspects of that system. Deciding which techniques to use involves an engineering tradeoff among the technical perspective, formalism, level of abstraction, and scope of the modeling effort.

The specific techniques and engineering practices of applying Model-Based Verification to software verification have yet to be fully explored and documented. A number of barriers to adoption of Model-Based Verification have been identified, including the lack of good tool support, expertise in organizations, good training materials, and process support for formal modeling and analysis.

In order to address some of these issues, the SEI has created a process framework for Model-Based Verification practice. This process framework identifies a number of key tasks and artifacts. Additionally, the SEI is working on a series of technical notes that can be used by Model-Based Verification practitioners. Each technical note is focused on a particular Model-Based Verification task, providing guidelines and techniques for one aspect of the Model-Based Verification practice. Currently, the technical notes that are planned address abstraction in building models, generating expected properties, generating formal claims, and interpreting the results of analysis.

The Analysis Guidelines technical note provides guidance for the analysis activity in the interpretation of results produced by model-checking tools. It states the problems related to interpreting results and provides strategies to overcome state explosion, to analyze results, and to provide feedback to the system designers and developers. In an appendix, it provides guidance for different MBV tools.

2 Analyzing Models

A central idea behind Model-Based Verification is to analyze essential models of the system to verify whether the model exhibits a particular behavior. *Build* and *Analyze* are MBV activities that cannot be separated. As shown in Figure 2, these activities follow an iterative approach, where a small amount of building is followed by a small amount of analysis. A parallel *Compile* activity gathers detailed information on errors and potential corrective actions.

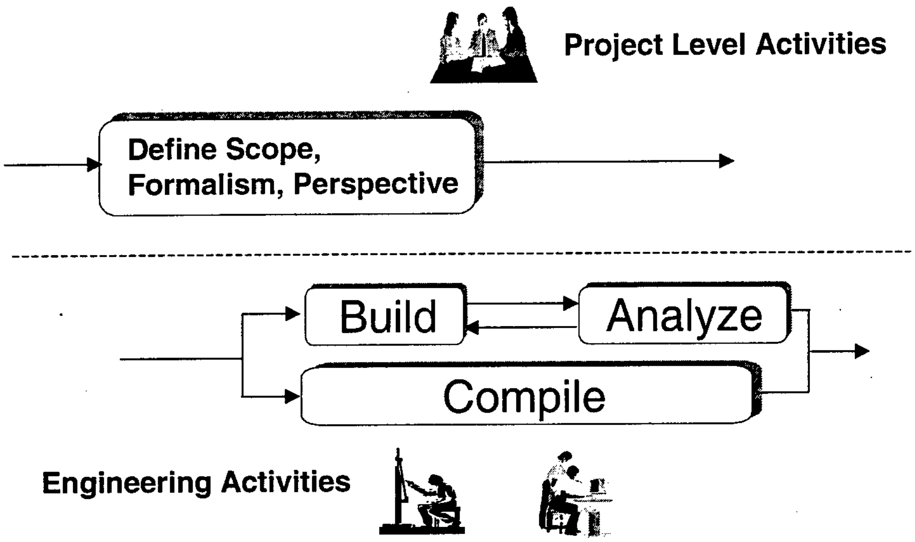


Figure 2: MBV Activities

During the analysis activity, shown in Figure 3, the main question to be answered is "Does the system behave correctly?" Correct behavior is defined in terms of expected properties. Expected properties are natural language statements about the behavior of a system—behavior that is consistent with user expectations[1]. Those expected properties are formalized into claims and checked against the model using an automated model-checking tool, such as Symbolic Model Verifier (SMV) [Comella 01].

[1] Gluch, D.; Comella-Dorda, S.; Hudak, J.; Lewis, G.; & Weinstock, C. *Model-Based Verification: Guidelines for Generating Expected Properties,* Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. To be published.

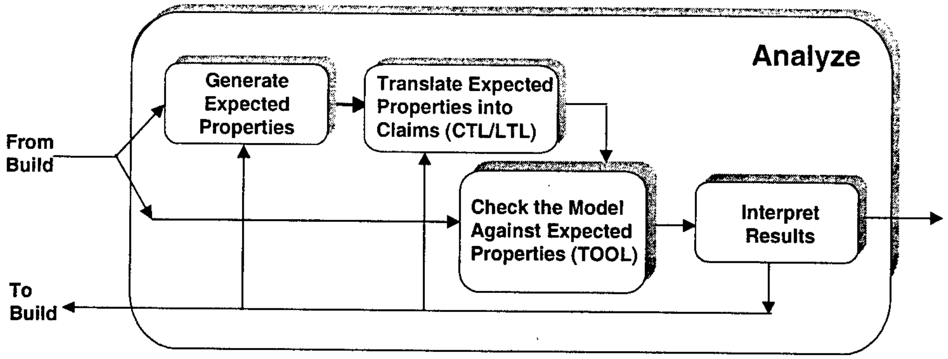

Analyze
Generate Expected Properties
Translate Expected Properties into Claims (CTL/LTL)
Check the Model Against Expected Properties (TOOL)
Interpret Results
From Build
To Build

Figure 3: Analyze Activity

The output of a state-machine-based automated model-verification tool is a confirmation or refutation of the specified expected properties. In most cases, if the property is not confirmed by the model checker, a counterexample will be provided. Counterexamples are executions of the model, showing the sequence of steps that negates the expected property.

A negative result in a model-checking tool will not always produce a counterexample. The temporal logic claim *EF p* (there exists a path where property *p* holds), for example, does not produce any counterexample. This happens because in order to provide a counterexample that this path does not exist, it has to enumerate all possible paths.

Obtaining a confirmation or a counterexample from a model-checking tool is not the end of the problem. Interpreting these results can, by itself, be as difficult as creating the models or the expected properties. The significance of the output of a model-checking tool must be assessed, considering particularly the following:

- If a claim is confirmed,
  - How to verify that in fact the system is correct?
- If a counterexample is returned,
  - Is the claim correct? Does it accurately reflect a valid expected property?
  - Does the model correctly reflect the behavior of the system as specified?
  - Does the interpretation of the counterexample reflect a potential defect in the system?

The interpretation of the result may prompt a number of different actions. For example, it may be necessary to revise the model to better reflect the system. This is part of the iterative nature of MBV: build or formulate part of a model, then evaluate it to see what changes need to be made. When a potential defect based on the interpretation above is discovered, it is recorded in a defect log.

The compile activity encompasses the tasks involved in logging and organizing defect data. Defects are compiled throughout all phases of the practice. Specific tasks in Compile include

- logging all defects
- organizing and analyzing defect data
- generating a defect analysis report

3 Interpreting Results

Three things can happen when a group of claims is assessed against a model using a model checker: the claims are validated, the claims are rejected (and a counterexample is provided), or nothing happens. When nothing happens, it is probably because the model checker is trying to parse an excessively large state space. This is known as the state explosion problem and is described in detail in Section 4.

If the model checker does provide an output it may well be that the claims are validated and the model is confirmed to have satisfied the expected properties. A positive result gives confidence but cannot assure that the model adequately represents the system at the right level of abstraction. These positive results have to be validated: claims should be verified for correctness; the model should be verified as an accurate representation of the system and translation into model-checking notation. Finally, results should be presented to users, customers, and domain experts, to confirm their validity.

Automated model checking is effective in finding cases in which the claims do not hold against the system models. The counterexamples provided by these results can reveal potential defects in the system or design, errors in the models, or errors in the claims. The following are the three possible causes for obtaining a counterexample[2]:

1. Error in the Claim: A counterexample needs to be verified against the system specification. If it does not reflect a valid state of the system, claims should be tested to determine if they have been well constructed and reflect the desired property to be verified.
2. Error in the Model: If the claims are correct, the counterexample could be caused by an error in the model. Errors in the model can be missing or invalid transactions, missing or invalid states, or missing initial values, among others.
3. Potential Defect in the System or Design: If after verifying the claims and the model, the counterexample still is valid, this could reflect a potential defect in the system or design as specified. This potential defect should be validated with domain experts and users and the result should be registered in a defect log.

How are errors in claims, errors in the model, and potential defects in the system distinguished? When results are inconclusive, what are the options? A good approach is to spend a few building-analysis cycles gaining confidence in the correctness of the model before probing the correctness of the system.

[2] These causes are not mutually exclusive; a combination of them is also possible.

Initially, fundamental claims corresponding to key elements are generated to ensure that the model accurately represents the behavior of the system. These first few claims often reveal mistakes made in building the model rather than flaws in the system. They help to provide confidence in the integrity of the model and its validity as a representation of the system. Both expected and unexpected behaviors should be specified and verified through the model checker at this phase. For example, in validating a model for a screen saver, an expected behavior to be verified is that the screen saver will be displayed if there is no input from the keyboard or mouse for a certain period of time $(AG\ (no\_input\_for\_x\_minutes \Rightarrow AF\ (screen\_saver\_active))$. An unexpected behavior to be verified is that the screen saver will never be displayed at the same time a critical alarm is being displayed $(AG\ !(screen\_saver\_active \wedge critical\_alarm))$.

As greater confidence is gained in the model, more probing claims, aimed at better understanding of the system and uncovering defects in the system, can be explored. This process should also be iterative such that claims motivated by investigating the system in depth are interspersed with those that are seeking to confirm the veracity of the model.

4 State Explosion

A major issue in analyzing state machines is the state explosion problem. The number of states grows very quickly as the complexity of the model increases. For example, even a relatively simple system consisting of four state machines or concurrent processes, each with three state variables, each having five values, results in a total state space of approximately 250 million states.

Tools such as Carnegie Mellon University (CMU) SMV use Binary Decision Diagrams[3] (BDDs) to represent sets and relations in the Computational Tree Logic (CTL) model-checking algorithm. The use of BDDs in symbolic model checking allows state spaces of 10^{50} states and up, with refinements of the BDD-based techniques pushing this number up to 10^{100} [Clarke 99]. 10^{50} is roughly equivalent to 2^{160}, meaning that SMV, in theory, could handle models with 160 binary variables.

Most research and literature on the topic deals with incorporating techniques and algorithms inside the tools to automate eliminating or reducing the state explosion problem. The following subsections present heuristics for model developers that have been derived from these techniques and from experience in model checking.

4.1 Variable Ordering

Identifying good variable ordering for BDDs is the focus of many research papers [Chan 98, Kamhi 98, Lu 00, Rudell 93]. This topic is claimed to be one of the central problems in using BDDs effectively because these are very sensitive to variable ordering. The efficiency of BDDs is based on the combination of isomorphic subtrees and the elimination of redundant decision nodes in the tree. Because of this, the size of the final BDD will be closely related to the variable ordering used.

Several SMV tools, for example the ones presented later in this paper, use variants of Rudell's dynamic sifting algorithm for dynamic variable reordering, but this process is very time consuming [Rudell 93]. These tools also provide options to explicitly indicate the preferred variable order; for producing output specifying the variable ordering that is being used; and for enabling or disabling dynamic reordering. Some tools also offer options for specifying the dynamic ordering heuristic to be used. More information is included in Appendix C – Features in CMU SMV, NuSMV, and Cadence SMV.

[3] A *BDD* is a decision tree, in which variables always appear in the same order as the tree is traversed from root to leaf.

Even though most tools provide ways to define or dynamically perform variable ordering, it is good to apply the following heuristics:

1. Declare closely related variables together.
2. Declare global variables first.
3. Order variables as they appear in the assignments [Winter 97].
4. Initially order variables manually or statically and run the model checker iteratively to produce an ideal ordering; do this first even if it allows dynamic reordering.

As sample data on how state space size is affected by variable ordering, the size of the state space in the example listed in Appendix B varies from 4802 to 3544 BDD nodes using CMU SMV (26% reduction) if the third heuristic is applied (the order in which variables are declared is matched to the order in which they are assigned values)[4]. If the first heuristic is applied (keeping together variables that are related) to the same example, the smallest state spaces are accomplished by ordering variables *e1* through *e5*, either *e1, e2, e3, e4, e5* or *e5, e4, e3, e2, e1*. This is because each of the variables *e1* through *e5* is defined in terms of the previous and/or next one. If this order is changed, so that the variables that are related are not declared close together, the size of the state space can go from 3544 to 4716 BDD nodes (33% increase).

4.2 Abstraction

Abstraction, when dealing with state explosion, refers to making the model's state space smaller. Visser states that when there are n concurrent processes or state machines with m states each, the system has m^n states [Visser 00]. The options to reduce m^n are to reduce m (abstraction), to reduce the effect of n (partial-order reduction), or to reduce n (symmetry reduction). There is a great amount of literature on abstraction [Clarke 99, Heitmeyer 98, Emerson 97, Visser 00], but as with variable ordering, most of the work is still in the research stage and is slowly being integrated into model-checking tools. It is still a manual process that requires considerable creativity [Clarke 01].

In general, abstraction reduces the state space by mapping complex components to simple abstract representations. The problem is making sure that the abstraction correctly models the behavior of the system it is modeling. This is also referred to as soundness.

There are some abstraction heuristics where the goal is to reduce the number of different data values:

- SMV tools are extremely inefficient in constructing BDDs for integers and real numbers [Chan 98]. A simple heuristic is to replace integer and real values with abstract representations. For example, if a variable is represented as an integer, but the model is only concerned with the ranges *0-200*, *201-350*, and *351-500* for that variable, it is best to

[4] In the example, the order of the variables *Persistent* and *Token* was interchanged in the VAR section to match the assignments in the ASSIGN section.

model it as *{low, medium, high}* corresponding to the previous ranges. This heuristic has been applied with success by practitioners and is documented in several papers and case studies [Atanacio 00, Gluch 99, Pasareanu 01].

- Also for enumerations, if there are values that are not relevant to the property being verified, these can be substituted for the value "other." For example, if the variable in the previous example could also take values below 0 or above 500, but these are not important for the property being verified, the state space can be reduced by replacing {very low, low, medium, high, very high} with {low, medium, high, other}.
- Remove variables from the model that are not relevant to the property being verified. This is done automatically by some tools and is called slicing[5]. A slice is the set of program statements that affect a given variable. For example, if the property being verified is $AG(x>0 \Rightarrow y>0)$, all variables different from x and y that are not relevant to changes in the values of x and y, as well as statements/transitions that are not relevant to the relationship between x and y, can be removed while this property is verified [Heitmeyer 98, Pasareanu 00].
- Assign initial values to state variables instead of leaving them free and minimizing nondeterminism [Pasareanu 01]. This heuristic is useful to verify portions of the system, but it has to be used with caution because nondeterminism is used to model unpredictable or unknown inputs to the system and by eliminating nondeterminism these inputs become fixed.
- Reduce the number of "symmetric" or "equivalent" components. For example, if a distributed networked client-server system has twenty clients and five servers, the system may be modeled as three clients and two servers [Gluch 99]. This reduction may not preserve soundness or completeness, but it can be effective in exposing defects at relatively low cost [Rushby 99].

As an example, Figure 4 shows a very simple model with a variable *temperature* declared as *0..100,* that counts to *100* and then returns to *0*, and a second variable *condition* that will change with certain values of *temperature*. Verifying the simple claim that it is always possible for *condition* to return to *cold* after it is set to *warm* required 1119 BDD nodes using CMU SMV. If the first heuristic is applied and temperature is changed to type *{low, medium, high}* and the *condition* variable still changes at certain values of *temperature*, and a fairness condition is added so it will not stay in the same state (as shown in Figure 5) verifying the same claim requires 84 BDD nodes using CMU SMV. This is obvious because it is going from 100 different values to 3 different values for a variable, but shows that it pays off to simplify a model in this way if it is possible.

[5] Variable slicing in model checking is also referred to as "cone of influence reduction."

```
MODULE main
VAR
        temperature: 0..100;
        condition: {cold, mild, warm};
ASSIGN
        init(temperature) := 0;
        init(condition) := cold;
        next (temperature) :=
                case
                        temperature = 100: 0;
                        1: temperature+1;
                esac;
        next (condition) :=
                case
                        temperature = 0 : cold;
                        temperature = 40 : mild;
                        temperature = 70 : warm;
                        1 : condition;
                esac;
SPEC
        AG ( condition = warm -> AF condition = cold )
```

Figure 4: Simple Example Using a Variable Modeled as an Integer

```
MODULE main
VAR
        temperature: {low, medium, high};
        condition: {cold, mild, warm};
ASSIGN
        init(temperature) := low;
        init(condition) := cold;
        next (temperature) :=
                case
                        temperature = low : {low, medium};
                        temperature = medium : {medium, high};
                        temperature = high : {high, low};
                        1: temperature;
                esac;
        next (condition) :=
                case
                        temperature = low : cold;
                        temperature = medium : mild;
                        temperature = high : warm;
                        1 : condition;
                esac;
SPEC
        AG ( condition = warm -> AF condition = cold )

FAIRNESS
        AG ( temperature = high -> AF temperature = low)
```

Figure 5: Simple Example Using a Variable Modeled as an Enumeration

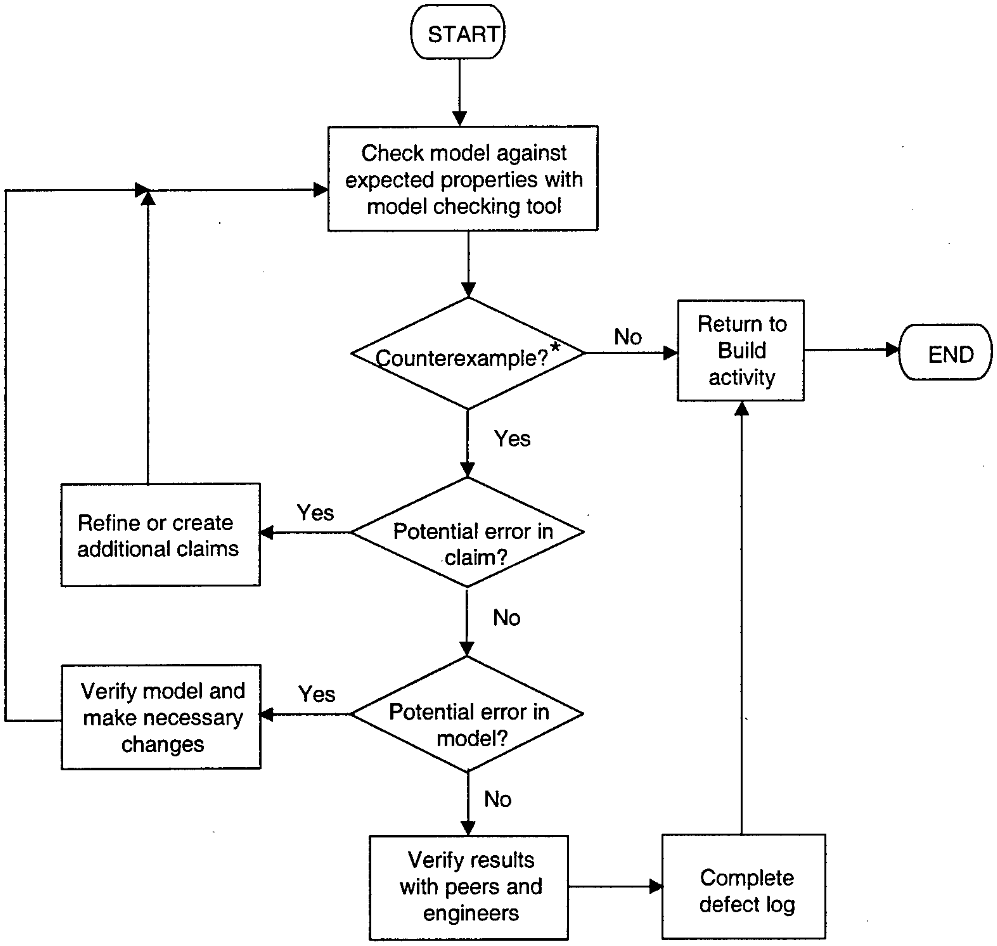
4.3 Decomposition

Another way of handling state explosion is through decomposition. Decomposition refers to breaking big problems into small problems, and then localizing the verification of each subproblem to a small part of the overall model [Clarke 89, McMillan 99]. In decomposition, different parts of the model are reasoned about separately, but the separate results are used to deduce properties of the entire system.

A form of decomposition is assume-guarantee reasoning, in which different components of a model are verified in isolation by making appropriate assumptions about the environment (possibly other portions of the model) [Pasareanu 99]. A heuristic based on assume-guarantee is better explained with an example: if model X can be decomposed into X_1, X_2, and X_3, and if X_1 can be verified, then X_1 can be used as an assumption in verifying X_2 and X_3. This assumption could be included as part of the model or specified as a fairness constraint. If X_1, X_2, and X_3 are all verified, then X is verified. The main difficulty with applying this heuristic is that sometimes properties that hold in some part of the system, when combined with the other parts of the system, may not hold for the original system. In this case, what holds for X_1, X_2, and X_3 separately might not hold when X_1, X_2 and X_3 are combined into X.

5 Corrective Actions

Counterexamples normally prompt some kind of corrective action (except when the claim was expected to be false). Counterexamples provide valuable debugging information, and can be used by the software engineer to modify the specification, the model, or the property checked [Chan 98]. This is the difficult part: What should be modified? Where is the error? Figure 6 summarizes the process of interpreting results. In this section it is assumed that MBV reviewers do not have direct responsibility for repairing the system and will complete a defect log with the discovered errors for others to repair.


START
Check model against expected properties with model checking tool
Counterexample?*
No
Return to Build activity
END
Yes
Potential error in claim?
Yes
Refine or create additional claims
No
Potential error in model?
Yes
Verify model and make necessary changes
No
Verify results with peers and engineers
Complete defect log

* It is assumed that the expected behavior is that a counterexample is not produced.

Figure 6: Interpreting Results Process

The next subsections illustrate each of the possible corrective actions taken after a counterexample interpretation.

5.1 Error in the Claim

Counterexamples show the sequence of states that demonstrate that the property being verified is not valid. If the sequence of states that is being presented by the counterexample does not reflect a valid state in the system, or does not seem related to the property being verified, there could be an error in the claim.

There are several heuristics for testing whether a claim is correct:

- Refer to a claim building guideline such as the one proposed by Comella [Comella 01] that describes a template-based approach to specifying claims in CTL and provides a list of simple claims templates that have been found to match a common expected property.
- Avoid using *EF p* (there exists a path where property *p* holds) because SMV cannot produce a specific counterexample. This happens because in order to provide a counterexample that this path does not exist, it has to enumerate all possible paths. If this is a desired expected property, a better way to represent it is using *!(EF p)*, or its equivalent *AG !p*. If a counterexample is obtained, it means that in fact there was a path where *p* holds. This is a case where a counterexample is a desired state.
- Try weaker or partial versions of the claim to iteratively work towards finding the error.

A way to apply this last heuristic is illustrated by the following examples:

Example 1

- Exact claim: Condition A is always true. Example: *AG (state = busy)*
- Weaker claim: Condition A is true multiple times (an infinite number of times). Example: *AG (AF (state = busy)).*
- Weakest claim: Condition A is true at least once. Example: *AG (EF (state = busy)).*

Example 2

- Exact claim: If Condition A occurs, then Condition B must occur. Example: *AG (state=ready ⇒AF (state=busy))*
- Partial claim: Condition A occurs. Example: *EF (state = ready)*

Weaker or partial claims might not represent the expected property exactly, but can help in working iteratively to find the error.

In Example 1, the idea is to start from the weakest claim and work towards the exact claim: first check if the condition is possible, then if it occurs more than once, and finally if it is always true.

In Example 2, the partial claim is represented by the antecedent in the implication proposition. The proposition will be true as long as the antecedent is true, so it is a good idea to test the antecedent by itself and work towards the complete proposition. In both examples, the claims as well as the model are being tested iteratively.

5.2 Error in the Model

If the claim that generates a counterexample has been tested and appears to be correct, the error could be in the model.

The model should accurately reflect the system or part of the system that is being analyzed. The questions listed below can help in determining if there is an error in the model. Having a graphical representation of the model, such as a state machine diagram or a state chart, is useful when answering these questions because transitions can be more easily traced. Given that this list is independent of the counterexample, it should be used to validate the model even before passing it through the model checker.

- Are all transitions present in the model also present in the SMV representation?
- Is there an initial state?
- Is the model non-deterministic in any initial state? Is this correct?
- Is all non-determinism intentional?
- Are all cases that are supposed to be mutually exclusive, in fact mutually exclusive?
- Do all states have at least one next state, except for end states?
- Can all desired states be reached?
- Are states reached in the proper order?
- Are mandatory states always reached?
- Will temporary states always be exited?

Studying the counterexample is also useful - follow the sequence of states presented by the counterexample and map it to the graphical representation of the model (if one is available) or follow its execution. The abstraction heuristics listed in *Section 4.2 – Abstraction* can be used if the sequence is too long. Before checking the property against the abstract model, it is important to re-verify all basic claims to make sure that the abstraction has not introduced additional errors. A model can be incorrect due to over-abstraction. As a very simple example, suppose a traffic light is abstracted as shown in Figure 7 [Clarke 00].

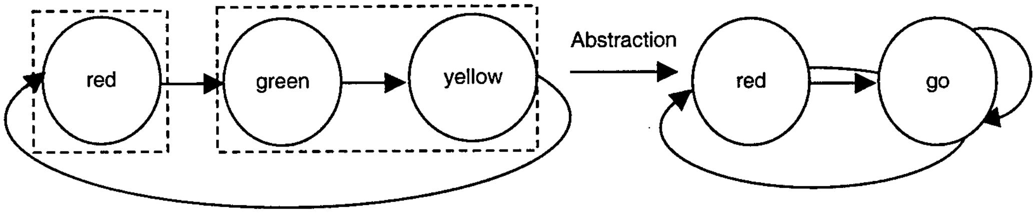


Figure 7: *Abstract Model of a Traffic Light*

If the property to prove is *AG AF(state = red)* (the traffic light is red infinitely often), the abstraction will produce the false counterexample < *red, go, go* >, which does not appear in the original model. In this case the abstraction is not a correct representation of the original model. If the loop from *go* to *go* is removed, the above property would be true in the original model and the abstraction.

The heuristic to be applied is that if a model is abstracted further during the process, basic properties, as well as all properties that had previously been verified, must be re-verified.

5.3 Potential Defect in the System or Design

If after the model has been verified there is still a counterexample that negates the property being verified, it could be an error in the specification or in the system being reviewed. The difficult part is how to make a judgment, especially for problems that are very domain-specific. It is often useful to have peer reviews or presentations to engineers where the defect is presented, along with the model and the analysis. The analysis must be translated into a representation that the attendants will understand. Gluch presents a procedure for peer reviews in model checking [Gluch 99]. In presenting the defect, the trace given by the counterexample should be the basis for explaining what sequence of events led to the error.

System requirements, domain experts, system architects, and artifacts such as specifications, code, and antecedent specifications are the places to look for answers. Interviews with users, customers, or domain engineers can help verify if the detected defect is in fact a system error. The system artifacts and other artifacts are used to map the defect to the system.

All defects have to be recorded in a defect log[6]. No special form is needed. The defect log can be based upon what is already in use within the organization. Items that should be included are

- unique defect identifier
- date the defect was found
- location of the defect in the specification
- comments about and/or a description of the defect

Other items that may be included are

- classification of the type of defect found
- activity where the defect was found
- injection point of the defect
- level of certainty (validity) of the defect

[6] In routine practice only potential defects in the system or design are logged. As an instrument for process improvement, defects could be recorded for all MBV activities.

- level of importance/severity of the defect
- estimation of effort required to fix the defect

A sample of a defect log is presented in Figure 8. Established organizational defect classification approaches can be used to enable the process to be readily integrated into existing practices.

Activity	Date	ID	Type	Doc Name	Page	Section	Description

Figure 8: Sample Defect Log

For significant defects found, it may be useful to create a detailed report that may include a

- defect summary
- system background information
- detailed description of the defect discovered
- statement on how the defect was found
- summary of any supporting documentation

A defect will require modifying the system, and consequently to modify, reanalyze, extend, or completely replace portions of the model. It is very important that modifications to the model stay consistent with the specification. It is also critical that the results from the model checking become a part of the overall verification process for the project.

6 Summary and Conclusions

Obtaining valuable results from a model-checking tool depends on a thorough analysis of these results. Dealing with state explosion is still a very manual and intuitive task, even though there is a great deal of active research on how to overcome it. Determining the validity of a counterexample requires acquisition of domain and system knowledge. When a counterexample is finally categorized as being the result of a potential defect, the analysis results must be communicated properly to the people responsible for system design.

This technical note has presented heuristics for dealing with state explosion; guidelines to help identify the type of result obtained; and finally ways to feed back the results to the system designers and developers. These heuristics and techniques are to be applied after the model is complete and expected properties are expressed as claims; they are not intended to replace the abstraction job and are not to be used as “quick solutions to a poor modeling job.”

There are still very few reports and sparse experience on the application of Model-Based Verification to software. Hopefully, as Model-Based Verification becomes more widely used and more commercial and research tools are available, outcomes of increased experience will feed back into the MBV community and will bring more insight into its application to detecting defects in software at an early stage.

Appendix A – SMV Semaphore Example[7]

This is an example of a semaphore in SMV. It will be used as a reference to illustrate what a counterexample looks like in the different tools that are addressed in this report.

```
MODULE main

VAR
  semaphore : boolean;
  proc1 : process user(semaphore);
  proc2 : process user(semaphore);

ASSIGN
  init(semaphore) := 0;

SPEC
  AG (proc1.state = entering -> AF proc1.state = critical)

MODULE user(semaphore)

VAR
  state : {idle,entering,critical,exiting};

ASSIGN
  init(state) := idle;
  next(state) :=
    case
      state = idle : {idle,entering};
      state = entering & !semaphore : critical;
      state = critical : {critical,exiting};
      state = exiting : idle;
      1 : state;
    esac;
  next(semaphore) :=
    case
      state = entering : 1;
      state = exiting : 0;
      1 : semaphore;
    esac;

FAIRNESS
 running
```

[7] Example taken from the CMU SMV documentation (http://www.cs.cmu.edu/~modelcheck/).

Appendix B – Synchronous Arbiter Example[8]

This is an example of a synchronous arbiter in SMV. It will be used as a reference to illustrate the effects of variable ordering for dealing with state explosion.

```
MODULE arbiter-element(above,below,init-token)

VAR
  Persistent : boolean;
  Token : boolean;
  Request : boolean;

ASSIGN
  init(Token) := init-token;
  next(Token) := token-in;
  init(Persistent) := 0;
  next(Persistent) := Request & (Persistent | Token);

DEFINE
  above.token-in := Token;
  override-out := above.override-out | (Persistent & Token);
  grant-out := !Request & below.grant-out;
  ack-out := Request & (Persistent & Token | below.grant-out);

SPEC
  AG ((ack-out -> Request) & AF (!Request | ack-out))

MODULE main

VAR
  e5 : arbiter-element(self,e4,0);
  e4 : arbiter-element(e5,e3,0);
  e3 : arbiter-element(e4,e2,0);
  e2 : arbiter-element(e3,e1,0);
  e1 : arbiter-element(e2,self,1);

DEFINE
  grant-in := 1;
  e1.token-in := token-in;
  override-out := 0;
  grant-out := grant-in & !e1.override-out;

SPEC
  AG (
      !(e1.ack-out & e2.ack-out)
      & !(e1.ack-out & e3.ack-out)
      & !(e2.ack-out & e3.ack-out)
      & !(e1.ack-out & e4.ack-out)
      & !(e2.ack-out & e4.ack-out)
      & !(e3.ack-out & e4.ack-out)
      & !(e1.ack-out & e5.ack-out)
      & !(e2.ack-out & e5.ack-out)
      & !(e3.ack-out & e5.ack-out)
      & !(e4.ack-out & e5.ack-out)
     )
```

[8] Example taken from the CMU SMV documentation (http://www.cs.cmu.edu/~modelcheck/).

Appendix C – Features in CMU SMV, NuSMV, and Cadence SMV Related to Analysis

The purpose of this appendix is to illustrate references to features of model-checking tools that have been made throughout this report.

Model-checking tools based on SMV take specifications for a given system, written in SMV language, and verify that every possible behavior of the system satisfies the specification. Sometimes the tool will produce a counterexample – an execution path that violates the specified property.

SMV, NuSMV, and Cadence SMV are free research tools available on the Internet. References for these tools are listed in the section "Tool References."

A General Notion about Fairness

One way in which SMV can produce counterexamples that are inconclusive is when an asynchronous system, where processes run in parallel, "stalls" infinitely in the same state because one of the processes is being starved. Fairness constraints will force the model checker to consider only fair execution paths. SMV has an internal variable for each process - called *running* - that is true if and only if that process is running. For example, the statement

```
FAIRNESS
   running
```

will ensure that the process runs eventually and that no process will run indefinitely because it only considers paths where *running = 1* infinitely often.

Running is not the only fairness constraint that can be specified. For example

```
FAIRNESS
   !(state = busy)
```

will not explore paths where the system is "stalled" in the *busy* state indefinitely.

CMU SMV

CMU SMV (Symbolic Model Verifier) is the original version of SMV, developed at Carnegie Mellon University by Ken McMillan. It is a command-line tool that takes a system represented as a set of states and transitions as input. It allows for the specification of expected properties in Computation Tree Logic (CTL) notation and automatically checks these properties against the state machine representation. Because CMU SMV is intended for describing state machines, it only accepts boolean, scalars, and fixed arrays data types. and allows for construction of static structured data types. CMU SMV is available for Windows and Unix platforms.

In CMU SMV, counterexamples are presented as a command-line printout with a trace in which the specification is false. Each state in the trace presents the values of variables that change and lead to the counterexample. Figure 9 shows a counterexample in which the property in the semaphore example listed in Appendix A could not be verified. The trace shows the execution sequence and the variables that changed in each state of the execution. The text -- *loop starts here* – indicates that states 1.3 to 1.9 repeat forever.

```
-- specification AG (proc1.state = entering -> AF proc1.s...
is false
-- as demonstrated by the following execution sequence
state 1.1:
semaphore = 0
proc1.state = idle
proc2.state = idle

state 1.2:
[executing process proc1]

-- loop starts here --
state 1.3:
proc1.state = entering

state 1.4:
[executing process proc2]

state 1.5:
[executing process proc2]
proc2.state = entering

state 1.6:
[executing process proc1]
semaphore = 1
proc2.state = critical

state 1.7:
[executing process proc2]

state 1.8:
[executing process proc2]
proc2.state = exiting

state 1.9:
semaphore = 0
proc2.state = idle


resources used:
processor time: 0.02 s,
BDD nodes allocated: 1131
Bytes allocated: 1045212
BDD nodes representing transition relation: 69 + 1
```

Figure 9: Counterexample in CMU SMV

Variable ordering in SMV can be adjusted manually using an "order file." At any point, the variable ordering can be written to a file, for inspection and possibly reordering. Dynamic variable ordering can also be enabled, but this will slow down the verification process. There is also the possibility of specifying a BDD size that when reached will start the dynamic variable reordering.

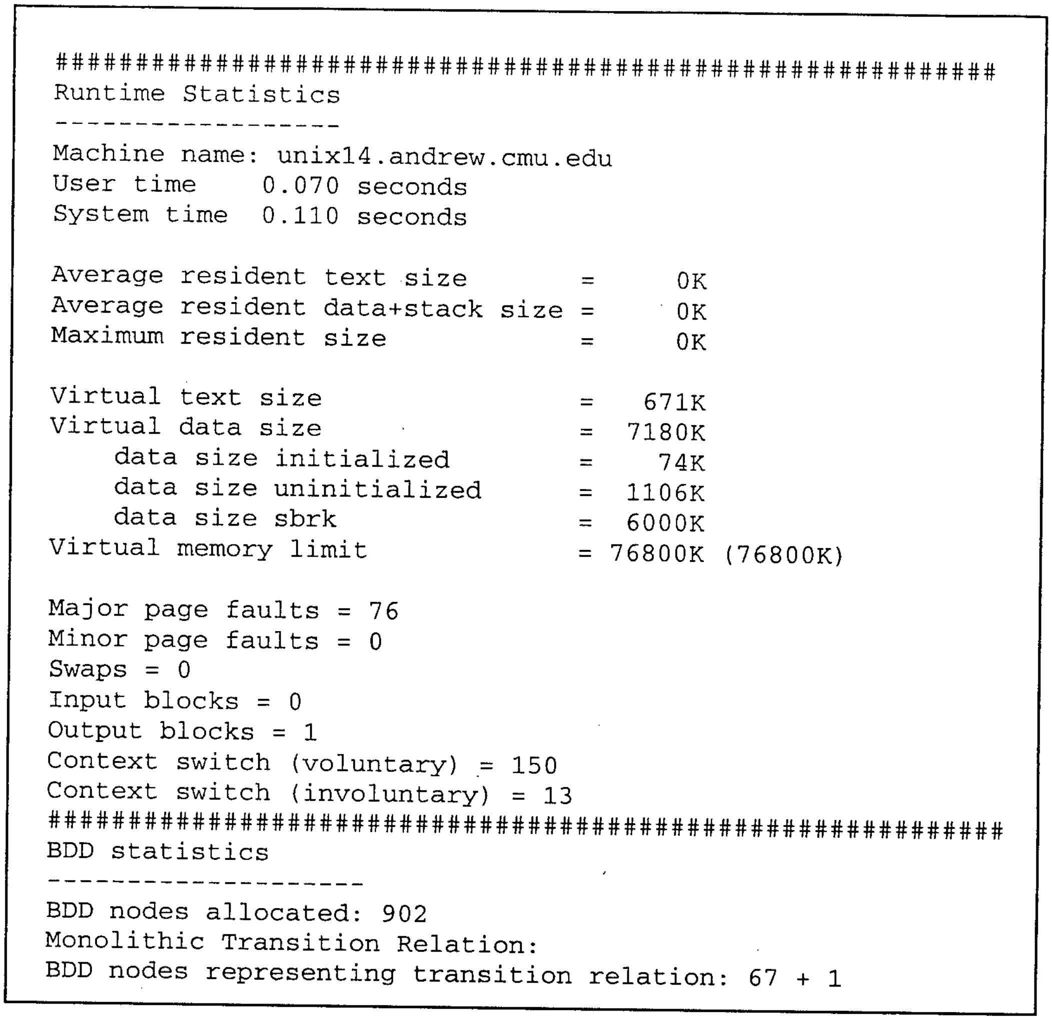
A new experimental feature in CMU SMV allows evaluation of AG specifications while building the set of reachable states. This often helps in finding bugs earlier before the complete model is built, but only works for AG specifications. If a specification is false, it prints a counterexample and removes the specification from the list, so it won't be evaluated again. If no specs are left, it exits immediately. This option is useful for dealing with state explosion, but may slow down verification.

NuSMV

NuSMV is a symbolic model checker developed as a joint project between the Formal Methods group in the Automated Reasoning System division at ITC-IRST and the Model Checking Group at Carnegie Mellon University. It is a newer version of the original SMV that also includes LTL for expressing claims and has command-line interface as well as a graphical user interface (GUI). NuSMV is only available for Unix-based platforms.

In the NuSMV command-line interface version, counterexamples look very similar to the ones produced by CMU SMV. The difference is in the statistics that it provides after showing the trace, as presented in Figure 10. This is the last part of the counterexample in which the property in the semaphore example listed in Appendix A could not be verified. If this data is compared to the counterexample from CMU SMV (Figure 9), a difference that can be seen is the number of BDD nodes allocated (NuSMV: 902 vs. CMU SMV: 1131). This is mainly due to the algorithm and heuristics improvements in NuSMV to deal with state explosion.

In the NuSMV GUI interface version, called xNuSMV, when the model is processed the counterexample information shown in the *Output Messages* window is the same as that produced by the command-line interface with the addition of the line "A Counterexample (trace no. 1) has been generated." This trace can be viewed using the *Trace Viewer*. Figure 11 shows a counterexample in which the property in the semaphore example listed in Appendix A could not be verified. The trace shows the states that lead to the counterexample and the value that each variable had in each state. The arrow pointing from State 9 to State 3 indicates that States 3 to 9 repeat forever.



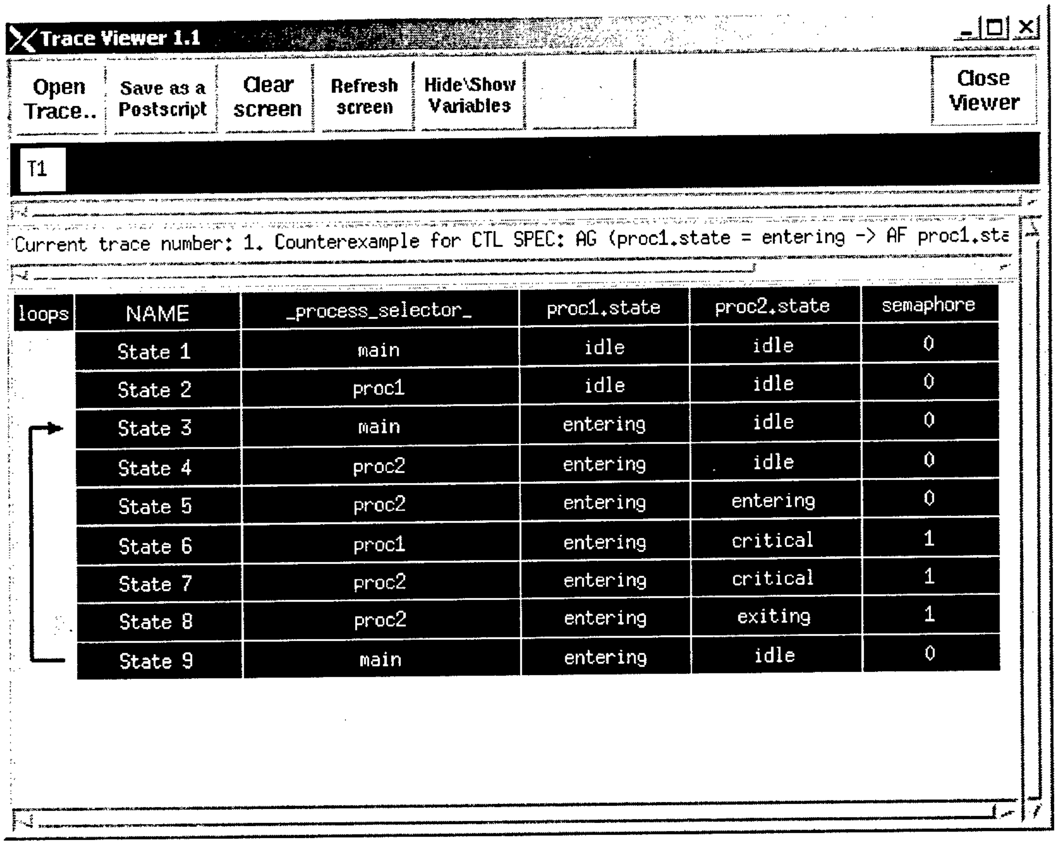
```
###########################################################
Runtime Statistics
------------------
Machine name: unix14.andrew.cmu.edu
User time    0.070 seconds
System time  0.110 seconds

Average resident text size       =     0K
Average resident data+stack size =     0K
Maximum resident size            =     0K

Virtual text size                =   671K
Virtual data size                =  7180K
    data size initialized        =    74K
    data size uninitialized      =  1106K
    data size sbrk               =  6000K
Virtual memory limit             = 76800K (76800K)

Major page faults = 76
Minor page faults = 0
Swaps = 0
Input blocks = 0
Output blocks = 1
Context switch (voluntary) = 150
Context switch (involuntary) = 13
############################################################
BDD statistics
--------------------
BDD nodes allocated: 902
Monolithic Transition Relation:
BDD nodes representing transition relation: 67 + 1
```

Figure 10: *Additional Statistics that are Part of a Counterexample in the Command-Line Version of NuSMV*



```
Current trace number: 1. Counterexample for CTL SPEC: AG (proc1.state = entering -> AF proc1.st
```

loops	NAME	_process_selector_	proc1.state	proc2.state	semaphore
	State 1	main	idle	idle	0
	State 2	proc1	idle	idle	0
	State 3	main	entering	idle	0
	State 4	proc2	entering	idle	0
	State 5	proc2	entering	entering	0
	State 6	proc1	entering	critical	1
	State 7	proc2	entering	critical	1
	State 8	proc2	entering	exiting	1
	State 9	main	entering	idle	0

Figure 11: Counterexample in xNuSMV – GUI Version of NuSMV

There is an open source development of NuSMV called NuSMV2 that combines BDD-based and SAT-based model-checking techniques[9] to deal with the state explosion problem [NuSMV 01].

NuSMV also accepts variable order files. The difference between SMV and NuSMV in this aspect is that NuSMV accepts partial variable specification as well as variables not declared in the model. It accepts specification of a method to use as the heuristic for variable ordering, such as several forms of sifting[10], random selection, windowing, annealing, genetic, and linear transformations[11].

Cadence SMV

Ken McMillan, original author for SMV, created Cadence SMV at Cadence Berkeley Labs. It is a tool oriented to computer hardware verification with more constructs and data types, but accepts CMU SMV as well as Synchronous Verilog as input. It supports composition and

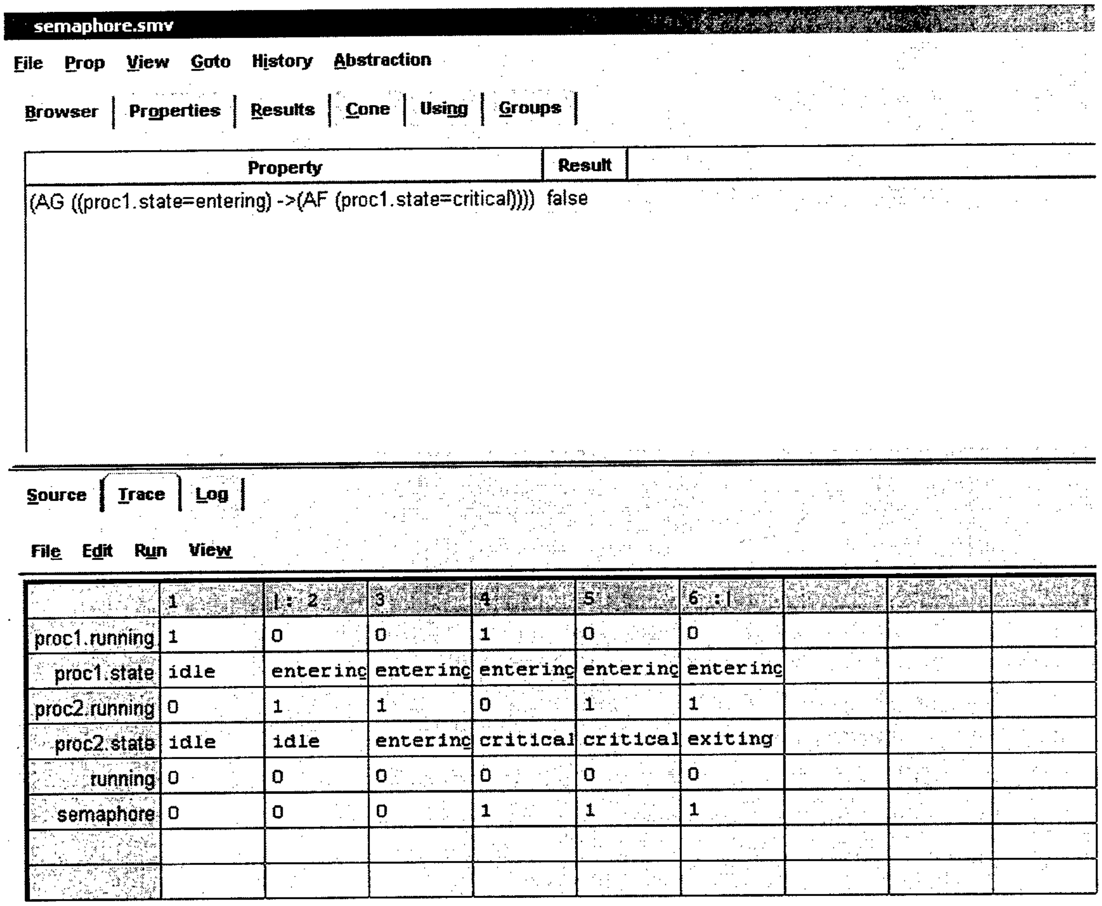
[9] SAT-based model checking techniques are based on prepositional decision procedures and claim to not suffer from the potential state explosion of BDD-based techniques [Biere 99].

[10] *Sifting* is a heuristic for dynamic variable reordering [Meinel 97, Rudell 93, Kamhi 98].

[11] An explanation of all these heuristics can be found in the URL for NuSMV listed in the Tool References section.

refinement; that is, it allows the user to verify properties of one part of the system and use these properties as assumptions when verifying another part of the system. It allows the use of a high-level model of the system as a specification, and verifies separately that each system component implements its part of the high-level specification. It also includes an easy-to-use graphical user interface (GUI) and properties can be expressed in CTL as well as LTL. Cadence SMV is available for Windows and Unix.

In Cadence SMV, counterexamples are presented as traces with a truth assignment to all the state variables that show that a property is false. Figure 12 shows a counterexample in which the property in the semaphore example listed in Appendix A could not be verified. The trace shows the values for each of the variables that lead to a state where the property is false. States 2 and 6 are marked with "repeat" signs, thus: `|: 2 ... 6:|` indicates that states 2 to 6 repeat forever. The log tab will also show the resources used, variable ordering, and other data concerning the verification operation.


semaphore.smv
File Prop View Goto History Abstraction
Browser | Properties | Results | Cone | Using | Groups
Source | Trace | Log
File Edit Run View

Property	Result
(AG ((proc1.state=entering) ->(AF (proc1.state=critical))))	false

	1	|: 2	3	4	5	6 :|			
proc1.running	1	0	0	1	0	0			
proc1.state	idle	enterinc	enterinc	enterinc	enterinc	enterinc			
proc2.running	0	1	1	0	1	1			
proc2.state	idle	idle	enterinc	critical	critical	exiting			
running	0	0	0	0	0	0			
semaphore	0	0	0	1	1	1			

Figure 12: Counterexample in Cadence SMV

The set of variables that a property depends on is referred to as the *cone* of that property. It is generally best to keep the number of variables in the cone small, when possible. Cadence SMV allows the viewer to select a given property to verify, then to view the cone of that property by clicking the “Cone” tab. Some state variables will be listed as “free”. This is because they are unconstrained and thus are free to take on any values in their type.

Cadence SMV supports refinement maps by allowing specification of many abstract definitions for the same state variable using a construct called a “layer”. A layer is a collection of abstract state variable definitions. A layer allows for proof that every implementation behavior is consistent with all of the given assignments. If this is the case, it can be said that the implementation refines the specification. Also, a different abstraction of the implementation can be used to prove each component of the specification.

A simple example of the use of layers in SMV is taken directly from the Cadence SMV documentation [Cadence SMV] and is shown in Figure 13. This sample is not written in CMU SMV input language, but in Cadence SMV input language.

```
module main(){
  x : boolean;

  /* the specification */

  layer spec: {
    init(x) := 0;
    if(x=0) next(x) := 1;
    else next(x) := {0,1};
  }

  /* the implementation */

  init(x) := 0;
  next(x) := ~x;
}
```

Figure 13: Example of Layers in Cadence SMV

The tool checks that the implementation satisfies the conditions in the specification by exhaustive search of the state space of the implementation. In the above example, the two instructions listed under the implementation are checked against the instructions included in the specification. In this case, the implementation satisfies the specification because *next(x) := ~x* satisfies the condition in the specification for values 0 and 1. If more than one state variable is assigned a value in a layer, these two variable definitions can be verified separately. A different abstraction of the implementation can be used to prove each component of the specification. For example, if state variables *x* and *y* are assigned values in the specification, and the specification definition of *x* needs to be verified, either the specification definition of *y* or the implementation definition of *y* can be used. However, if the specification definition of y is used, one state variable is eliminated from the model. Thus, by decomposing a specification into parts, and using one part as the "environment" for another, the number of state variables in the model is reduced.

Cadence SMV has built-in heuristics for selecting the variable ordering, which can also be disabled so that variables appear in the BDDs in the same order in which they are declared in the program. It also allows sifting,[12] which slows down the verification process but reduces the state space. Variable ordering can also be adjusted manually by adding redundant variable declarations to the program, so that the variables are declared in the desired order, or by creating a variable order file. The variable order used can also be output to a file.

[12] See previous footnote.

References

Atanacio 00 Atanacio, B. *Modeling the Space Shuttle Liquid Hydrogen Subsystem* (CMU/SEI-2000-TN-002). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. http://www.sei.cmu.edu /publications/documents/00.reports/00tn002.html

Biere 99 Biere, A.; Cimatti, A.; Clarke, E.; Fujita, M.; & Zhu, Y. *Symbolic Model Checking Using SAT Procedures Instead of BDDs* (CMU-CS-99-145). Pittsburgh, PA: School Of Computer Science, Carnegie Mellon University, 1999.

Chan 98 Chan, W.; Anderson, R.; Beame, P.; Burns, S.; Modugno, F.; Notkin, D.; & Reese, J. "Model Checking Large Software Specifications." *IEEE Transactions on Software Engineering 24*, 7 (July 1998): 498-519.

Clarke 89 Clarke, E; Long, D; & McMillan, K. "Compositional Model Checking." *Proceedings of the Fourth Annual Symposium on Logic in Computer Science - LICS '89*. Pacific Grove, CA, June 1989. New York, NY: IEEE Computer Society Press, 1989.

Clarke 92 Clarke, E.; Grumberg, O.; & Long, D. "Model Checking and Abstraction." Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Asilomar, CA, January 1992. New York, NY: ACM Press, 1992.

Clarke 95 Clarke, E.; Grumberg, O.; Hiraishi, H.; Jha, S.; Long, D.; McMillan, K. & Ness, L. "Verification of the Futurebus+ Cache Coherence Protocol." *Formal Methods in System Design 6, 2* (March 1995): 217-232.

Clarke 96 Clarke, E. & Wing, J. "Formal Methods: State of the Art and Future Directions." *ACM Computing Surveys 28*, 4 (December 1996): 626-643.

Clarke 00 Clarke, E.; Grumberg, O.; Jha, S.; & Veith, H. "Counterexample-guided Abstraction Refinement." *Proceedings of the Twelfth International Conference on Computer Aided Verification*. Chicago, IL, July 2000. Heidelberg, Germany: Springer-Verlag, 2000.

Clarke 01 Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; & Veith, H. "Progress on the State Explosion Problem in Model Checking." in R Wilhelm (Ed.) "Informatics. 10 Years Back. 10 Years Ahead." Heidelberg, Germany: Springer-Verlag, 2001. http://link.springer.de/link/service/series/0558/papers/2000/20000176.pdf(2001).

Comella 01 Comella-Dorda, S.; Gluch, D.; Hudak, J.; Lewis, G.; & Weinstock, C. *Model-Based Verification: Claim Creation Guidelines* (SEI/CMU-2001-TN018). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. http://www.sei.cmu.edu/publications/documents/01.reports /01tn018.html

Emerson 97 Emerson, C.; Jha, S.; & Peled, D. "Combining Partial Order and Symmetry Reductions." *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Enschede, The Netherlands, April 1997. Heidelberg, Germany: Springer-Verlag, 1997.

Gluch 98 Gluch D. & Weinstock, C. *Model-Based Verification: A Technology for Dependable Systems Upgrade* (CMU/SEI-98-TR-009, ADA354756). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1998. http://www.sei.cmu.edu/publications/documents /98.reports /98tr009/98tr009abstract.html

Gluch 99 Gluch, D. & Brockway, J. *An Introduction to Software Engineering Practices Using Model-Based Verification* (CMU/SEI-99-TR-005, ADA366089). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. http://www.sei.cmu.edu/publications /documents/99.reports/99tr005/99tr005abstract.html

Heitmeyer 98 Heitmeyer, C.; Kirby, J.; Labaw, B.; Archer, M.; & Bharadwaj, R. "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications." *IEEE Transactions on Software Engineering 24*, 1 (November 1998): 927-948.

Kamhi 98 Kamhi, Gila & Fix, L. "Adaptive Variable Reordering for Symbolic Model Checking." *IEEE/ACM 1998 International Conference on Computer-Aided Design Proceedings*. San Jose, CA, Nov. 8-12, 1998. In *IEEE/ACM Digest of Technical Papers*. New York, NY. ACM Press, 1998.

Lu 00 Lu, Y.; Jain, J.; Clarke, E.; & Fujita, M. Efficient Variable Ordering Using a BDD Based Sampling. *Proceedings of the 37th Conference on Design Automation*. Los Angeles, CA, June 5-9, 2000. New York, NY: Association for Computing Machinery, 2000.

McMillan 92 McMillan, K. *The SMV System, Symbolic Model Checking - An Approach* to *the State Explosion Problem* (CMUCS-92-131). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University, 1992.

McMillan 99 McMillan, K. *Verification of Infinite State Systems by Compositional Model Checking.* Berkeley, CA: Cadence Berkeley Labs, 1999.

Meinel 97 Meinel C. & Slobodová, A. "Speeding up Variable Reordering of OBDDs." *IEEE 1997 International Conference on Computer Design Proceedings.* Austin, TX, Oct. 12-15, 1997. New York, NY: IEEE Computer Society Press, 1990.

NuSMV 01 Open Source NuSMV Project. http://nusmv.irst.itc.it/open_nusmv/flier.html (2001).

Pasareanu 99 Pasareanu, C.; Dwyer, M.; & Huth. M. "Assume-Guarantee Model Checking of Software: A Comparative Case Study." *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops.* Heildelberg, Germany:Springer-Verlag, 1999.

Pasareanu 01 Pasareanu, C.; Dwyer, M.; & Visser. W. "Finding Feasible Counter-examples when Model Checking Abstracted Java Programs." *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Genova, Italy, April 2-6, 2001.

Rudell 93 R. Rudell. "Dynamic Variable Ordering for Ordered Binary Decision Diagrams." *IEEE/ACM 1993 International Conference on Computer Aided Design Proceedings.* Santa Clara, CA, Nov. 7-11, 1993. NewYork, NY: IEEE Computer Society Press, 1994.

Rushby 99 Rushby, J. "Integrated Formal Verification: Using Model Checking with Automated Abstraction, Invariant Generation, and Theorem Proving." *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops.* Heidelberg, Germany:Springer-Verlag, 1999.

Sreemani 96 Sreemani T. & Atlee, J. "Feasibility of Model Checking Software Requirements: A Case Study." *Proceedings of the 11th Annual Conference on Computer Assurance.* Gaithersburg, MD, June 17-12, 1996.

Winter 97 Winter, K. "Model Checking for Abstract State Machines." *Journal of Universal Computer Science 3,* 5 (May 1997): 689-701.

Visser 00 Visser, W. & Pecheur, C. "Model for Software." Tutorial presented at the

Automated Software Engineering 2000 Conference. Grenoble, France, September 12, 2000.

Tool References

Symbolic Model Verifiers (SMVs)

Cadence SMV	http://www-cad.eecs.berkeley.edu/~kenmcmil/smv/
CMU SMV	http://www.cs.cmu.edu/~modelcheck/
NuSMV	http://sra.itc.it/tools/nusmv/

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY
(Leave Blank)

2. REPORT DATE
December 2001

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Model-Based Verification: Analysis Guidelines

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Grace A. Lewis, Santiago Comella-Dorda, David P. Gluch, John Hudak, Charles Weinstock

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2001-TN-028

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This technical note provides guidance for the analysis activity that occurs during the interpretation of results produced by model-checking tools. In the model-checking analysis activity, the main question is, "Does the system behave correctly?" To answer this question, a model and a set of expected properties are used as input to a model checker. The expected output is a confirmation or refutation of the specified expected properties. In most cases, if the model checker does not confirm the property, it provides a counterexample.

Counterexamples are executions of the model showing the sequence of steps that refutes the expected property. Sometimes the state space to be explored in order to find this counterexample is so large that it cannot be completely covered. This is the state explosion problem. Models must be tuned to reduce the state space; this is a manual and intuitive task.

Interpreting the model checker's output can also be difficult. The significance of the output must be assessed; its interpretation may suggest an error in the claims or the model, or a defect in the actual system.

This document presents the problems related to interpreting results. It provides strategies to overcome state explosion, analyze results, and provide feedback to the system designers and developers.

14. SUBJECT TERMS
Model-Based Verification, model checking, symbolic model verifier, Cadence SMV, CMU SMV, NuSMV, state explosion

15. NUMBER OF PAGES
41

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL