Model-Based Verification: Claim Creation Guidelines

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

October 2001

Performance Critical Systems

Unlimited distribution subject to the copyright.

Technical Note
CMU/SEI-2001-TN-018

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

Copyright 2001 by Carnegie Mellon University.

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.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. Requests for permission to reproduce this document or prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

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 52.227-7013.

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

Abstract **v**

1 Introduction **1**

2 Expected Properties and Claim Building **4**
  2.1 Visualization 6
  2.2 Natural Language Translation 6
  2.3 Patterns 7

3 A Template-Based Approach to Claim Building **8**

4 Using the Templates **10**

5 Template List **13**
  5.1 Occurrence group 13
    5.1.1 Initial State Reachability 13
    5.1.2 Transition Firability 13
    5.1.3 Global Reachability 13
    5.1.4 Infinitely Often 14
    5.1.5 P Until Q 14
    5.1.6 Co-occurrence 15
    5.1.7 Error Free Execution 16
    5.1.8 Permanent Occurrence 16
    5.1.9 Mutual Exclusion 16
  5.2 Cause - Effect group 17
    5.2.1 Cause - Effect 17
    5.2.2 Permanent cause - Effect 18
    5.2.3 Cause - Scoped Effect 18
    5.2.4 Cause - Chained Effects 19
    5.2.5 Immediate Precondition 19
    5.2.6 Chained Causes - Effect 20
  5.3 Non-Progress (a.k.a. Liveness) 20
    5.3.1 Investigating Deadlock 20
    5.3.2 Investigating Starvation 21

6 Conclusions **22**

References **23**

List of Figures

Figure 1: Model-Based Verification Process and Artifacts 1

Figure 2: Tree Representation of the State Space 6

Figure 3: A Diagram for Template Selection 11

Abstract

Model Based Verification (MBV) is a systematic approach to finding defects (errors) in software requirements, designs, or code. MBV involves creating essential models of system behavior and analyzing these models against formal representations of expected properties, known as claims. Claim generation has been identified as a particularly complex activity within model-based verification. This technical note describes a pattern-based approach to facilitate claim generation. The report includes a list of directly usable patterns for the most frequent expected properties found in system specifications.

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

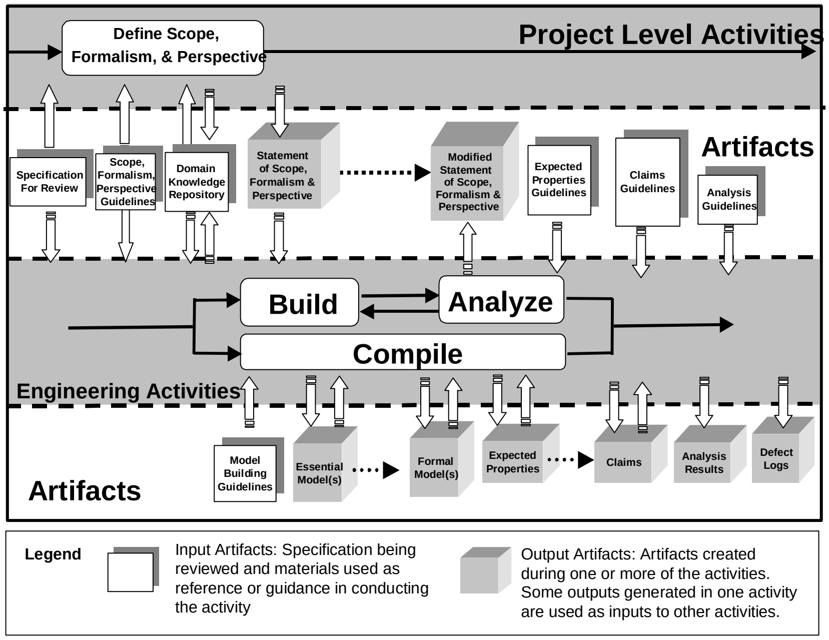
Define Scope, Formalism, & Perspective
Project Level Activities
Artifacts
Specification For Review
Scope, Formalism, Perspective Guidelines
Domain Knowledge Repository
Statement of Scope, Formalism & Perspective
Modified Statement of Scope, Formalism & Perspective
Expected Properties Guidelines
Claims Guidelines
Analysis Guidelines
Build
Analyze
Compile
Engineering Activities
Artifacts
Model Building Guidelines
Essential Model(s)
Formal Model(s)
Expected Properties
Claims
Analysis Results
Defect Logs
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

Essential models are simplified formal representations that capture 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. Within this analysis, potential defects are identified 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.

A variety of different formal modeling and analysis techniques are employed within model-based verification [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. This decision on the technique(s) involves an engineering trade-off 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.

This technical note focuses on claim generation. Specifically, it describes a template-based approach to simplify ad speed up this resource-intensive task. Section 2 introduces the problem of claim generation and illustrates its tradeoffs and incoherent complexities. Section 3 gives a high-level description of the proposed technique. Section 4 explains how to use the technique with some practical consideration and recommendations. Section 5 contains the actual list of claim templates. Finally, some conclusions and future lines of work are presented.

2 Expected Properties and Claim Building

Models in MBV are formal representations of system behavior. Similarly, claims are the formal specification of expected properties of the system. The formality of models and claims enables automated tools to verify whether particular claims hold against a model. If a claim is well written and the model is a faithful representation of the system, the verification using model checking will indicate whether the system possesses or does not possess the expected property represented by the claim. Additionally, the rigor required to build claims demands a level of system understanding that, in and of itself, promotes the uncovering of defects.

Depending on the kind of expected property and the type of model checker, different notations can be used to formalize the claim. For example, classical propositional logic can be used if we are concerned about static properties of the system. Using classical propositional logic, we can claim that *if it is raining then there are clouds*. In this report, we cover an extension to propositional logic that also considers time. This is called temporal logic and enables us to make such interesting claims as *if there are clouds then it may finally rain*.

Temporal logic is a formal approach for specifying and reasoning about the dynamic characteristics of a system. The formalism of temporal logic does not include time explicitly (i.e., counting or measuring time in the sense of a timing device). Rather time is represented as a sequence of states (a state trace) in the behavior of a system. These sequences of states (traces) can be finite <s0,s1, s2, s3, … sn> or infinite <s0, s1, s2, s3, ...>. In this report, we will assume infinite traces. States represent finite time intervals of fixed conditions for a system. For example, the state of red for a traffic light is when the color of the light is red.

There are two prominent forms of temporal logic:[1]

1. Linear Temporal Logic (LTL)
2. Branching Temporal Logic - Computational Tree Logic (CTL)

The difference between the two is in the concept of the unfolding of time. In linear temporal logic, the evolution of time is viewed as a sequence of states—a single line of possible states. In contrast, CTL (a version of branching temporal logic) views the evolution of time as possibly proceeding along a multiplicity of paths. Each path is a linear sequence of states. Some expected properties can be expressed using any of these formalisms, others require either CTL or LTL.

---

[1] Interval logic and other temporal logic variants are not covered in this report

Model checking has been used extensively for analyzing concurrent systems. Instead of depending on time explicitly, concurrent systems depend on the state of other system components. Take a producer-consumer example in which no overwriting occurs and where a producer produces messages, eventually filling up disk space. The CTL expression below states that if the producer produces with infinite frequency and the consumer never consumes, the buffer will eventually fill up:

(AGAF(producer.producing) & !EF(consumer.consuming)) → AF(disk.full)

It would be useful to claim that when (if) the disk is full, the producer must stop producing until the consumer consumes some messages and frees up some buffer space. Then the producer is allowed to continue. Such a claim may be expressed in CTL as[2]:

AG(disk.full → A[(!producer.producing) U consumer.consuming])

Defects can usually be categorized as violations of safety, liveness, or fairness properties. In the producer-consumer example, a fairness claim asserts that the consumer will never be indefinitely prevented from consuming. A way to express this is to claim that the consumer will consume infinitely often:

AGAF(consumer.consuming)

If this claim is false, the counterexample may show an execution sequence where the consumer, at some point, is never again allowed to consume a message from the buffer. A liveness claim might assert that both the producer or consumer never stop (i.e., the producer produces infinitely often and the consumer consumes infinitely often):

AG(AF(producer.producing) & AF(consumer.consuming))

If the system deadlocks, the counterexample might demonstrate an infinite loop where the producer or consumer or both are idle.

Even from this simple example, you can realize the difficulty in building temporal logic claims. Temporal logic syntax is obscure and the semantics can be very tricky. A number of strategies have been suggested to deal with this difficult task. Some of these are general to any engineering activity; others are specific to claim building. All have weaknesses and strengths and can be combined to get the best results. The strategies that we are going to mention in this section are visualization, natural language translation, and patterns. The pattern strategy will be the foundation for the technique described in the rest of the paper.

---

[2] There are subtle aspects to this claim that must be considered in interpreting it. If this claim is true for a system then (1) it is possible that the disk is never full, (2) if the disk does become full the consumer must eventually begin to consume, and (3) it is possible that in the same state that the consumer begins to consume again after the disk is full, the producer is also producing.

2.1 Visualization

Visual analogies have successfully been used to attack complex problems in all fields of engineering. In fact, the model-building phase of model-based verification uses statecharts and other graphic representation of state models [Harel 87]. If we can identify a good visual analogy of a temporal logic statement, we can also visually map the expected properties into temporal logic claims. There are various informal graphical analogies to depict temporal logic claims. Figure 2 illustrates a widely used analogy to represent the state space of a given state machine. The state space is represented as a tree, a node represents the system state in a particular point of time, and the siblings of a node are the potential next states. Every branch is a possible trace execution of the system. The figure includes the graphical representation of four basic CTL claims: AG, AF, EG, EF. The states in which the proposition (lunch is free) is true are darkened.

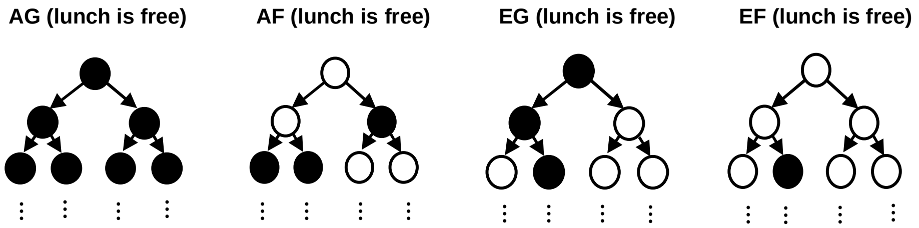

Figure 2: Tree Representation of the State Space

The tree analogy works well to illustrate simple CTL claims but soon becomes unmanageable as the complexity of the claim increases. Additionally, the representation is open to ambiguous interpretation. Attempts have been made to define formal graphical representations to express temporal logic claims. Graphical Interval Logic (GIL) describes linear temporal logic formulas in a way that resembles the informal timing diagrams familiar to designers of hardware systems [Dillon 94a, 94b].

2.2 Natural Language Translation

Another technique to simplify claim building is natural language translation. Expected properties are often expressed in natural language. Ideally, a tool could accept these English expressed properties and generate the claim in the required formalism [Holt 99, Grover 93]. Despite the important advances in natural language interpretation, however, the technique still presents some limitations. First, the tool has to deal with the inherent ambiguity of natural language. Additionally, the practitioner has to learn the natural language style best understood by the translation tool.

One way to work around some of these problems is not to use natural language but some intermediate form, easier to understand than temporal logic but less ambiguous than natural language. The Bandera Specification Language, for example, provides syntax for specifying general assertions and pre/post conditions on methods [Corbett 00]. Assertions are constructs available in many programming languages (e.g., as assert () statements and if-statements in sequential programming languages). This syntax is more familiar and manageable than temporal logic to the target user; it is, however, more formal than natural language.

2.3 Patterns

Design patterns have their roots in the architectural work of Christopher Alexander [Alexander 77]. Gamma et al. introduced patterns to software development as a means of leveraging the experience of expert system designers [Gamma 94]. There are multiple definitions of patterns in the literature; one of the shortest defines patterns as *the abstraction from a concrete form, which keeps recurring in specific non-arbitrary contexts* [Riehle 96].

Patterns have been proposed for expected property specification [Dwyer 99]. A property specification pattern is a generalized description of a commonly occurring requirement, on the permissible state/event sequences, in a finite state model of a system. A property specification pattern describes the essential structure of some aspect of a system's behavior and provides expressions of this behavior in a range of common formalisms. Given the proper property specification pattern, writing claims is simple. Instantiating a claim pattern is trivial in most cases, as the substitution of symbols by specific propositions is straightforward.

Dwyer et al. describe a set of very general patterns that can be customized with temporal scopes and used in a wide range of situations [Dwyer 98, 99]. There are five basic kinds of scopes:

1. global (the entire program execution)
2. before (the execution up to a given state/event)
3. after (the execution after a given state/event)
4. between (any part of the execution from one given state/event to another given state/event)
5. after-until (like between but the designated part of the execution continues even if the second state/event does not occur)

3 A Template-Based Approach to Claim Building

The approach adopted on this report is based on templates. We define templates as very specific fine-grained patterns. Our goal has been to provide constructs that are directly usable in a model-based verification with little or no customization. The resulting templates are too specific and narrow to be called patterns according to the accepted meaning for the term.

In the last section, we gave a short definition of patterns. However, in order to highlight the differences and likeness of our templates to traditional patterns, we need a more extensive definition. A popular definition in the research community characterizes a pattern as *a named nugget of instructive information that captures the essential structure and insight of a successful family of proven solutions to a recurring problem that arises within a certain context and system of forces* [Appleton 00]. As shown below, our templates comply with some but not all the characteristics of the “standard” patterns definition.

- *Named nugget of instructive information*: Templates do have names for easy reference and contain all the information necessary to apply the template in practice. We have tried to avoid names that are heavily used in temporal logic research but do not correspond to common terminology in requirements specification (for example, “fairness”).
- *that captures the essential structure and insight*: The templates capture the basic temporal logic structure to express the concept encapsulated by the template. The templates use symbols instead of specific propositions to make the solution general.
- *of a successful family of proven solutions to a recurring problem*: As we stated previously, there are very few documented uses of MBV in software settings. We have collected the solutions that worked for us in real MBV practice. We cannot prove, but we believe, that they will be useful for the general user.
- *that arises within a certain context and system of forces*: The context considered in this report is model-based verification of software systems in general. We have tried to keep the context general in order to make the templates useful for as wide an audience as possible. However, the templates were developed primarily in an application context associated with concurrent systems. Other application contexts would potentially produce different templates.

Given the mismatches between our constructs and the accepted definition of patterns, we did not want to use the word “pattern” to refer the approach adopted in this report. However, we do want to highlight the similarities between the solution described here and pattern theory.

Specification templates are less general than specification patterns but may be easier to use for the novice practitioner. This level of granularity also allows us to express the templates in terms closer to the expected properties an engineer is likely to find in a requirement.

specification. In fact, we have different templates with the same temporal logic structure but associated with different expected properties.

After describing what we mean by specification template, we must address the interesting question of which templates to include. There is an unlimited number of expressions that can be built in either CTL or LTL. In the decision to include or not include a particular template, we used two criteria: 1) simplicity and 2) correspondence with a known and commonly used expected property. We elaborate on these criteria below.

- **Simplicity**: Many potential CTL templates are too complex to be of any value. For example, the claim EG(A (p U s) -> AG(r->t | u)) is a perfectly valid claim. However, we should wonder whether the claim "There is a trace in which if p holds until s and s occurs then for every possible state if r then t or u or both" would be useful in any reasonable context. As claim complexity increases, understanding what we are verifying becomes increasingly difficult. Complexity also makes the interpretation of the results of the model checker more difficult and error prone. Yet, we don't want to include templates that are too simple to present any challenge even to a novice practitioner.
- **Correspondence to a known expected property**: CTL claims are the formalization of expected properties. Expected properties are often shared by multiple systems. Deadlock-free execution, for example, is a common property that is expected from most concurrent systems. In fact, even those expected properties that are not shared (system specific properties), can often be classified into common categories. For example, expected properties about the relation between two or more system events are very common in every system. These similar properties often translate into claims with the same structure, i.e., claims that are instantiations of the same claim template.

This report contains a list of those simple claims templates that we have found to match a common expected property. This list is neither comprehensive nor finalized; we plan to update the list as new templates emerge from common MBV practice. Our goal is not to compile every possible claim template, but to have a sufficiently complete list to simplify the practice for novice model-based verifiers.

Apart from the evident benefit of speeding up the building of claims, the template approach has some beneficial side effects. First, the templates are good learning tools. Engineers with limited modeling experience will first instantiate templates without further manipulation. Soon, they will start combining templates to address issues specific to their system. Finally, they will be able to create completely new templates that address very specific properties of their system. Second, the template approach can help to achieve consistency between different verification efforts. Different model-based verifications often focus on different aspects of the system, possibly overlooking some properties important enough to be checked every time. The templates can be used as a checklist to verify that all important properties of a system have been verified.

4 Using the Templates

Using the templates is fairly straightforward. We present the claims in a very simple layout based on the Alexandrian or Canonical form:

1. **Name**: In general we try to use the name of the expected property that the claim is formalizing. The name should be familiar to an engineer, as it is based on common terminology found in requirements specifications.
2. **Description**: Immediately following the name, we give a textual description of the meaning of the template and the temporal logic structure (either in CTL or LTL) that corresponds to that meaning.
3. **Examples and known uses**: Recommendations and guides for properly using the claim are provided. These include: examples of the template in a specific context, general situations in which the template can be used, clarifications, and caveats.
4. **Related templates (optional)**: Relations with other templates are provided, either because of their similarity or because they are often used together.

Given the relatively large number of templates in this report, sequentially parsing through the templates to find the appropriate one is not efficient. For this reason, we have classified the templates using behavioral groupings (occurrence, cause effect, and non progress). In addition to the classification, we have also created a wizard-like diagram (Figure 3) that guides the engineer to a suggested set of claims through a series of questions. The suggested claims must be reviewed to determine which one corresponds best to the expected property.

After selecting a claim template, we still have to instantiate it for the specific context under consideration. In the trivial case, the symbols must be substituted by specific propositions. This is very simple and we will not go into further detail. However, there is a much more interesting case: a symbol can be replaced by another claim template. For example, suppose that when an error condition emerges the pilot must be notified infinitely often (we are not modeling the warning disconnection mechanism). There is a claim for cause effect: AG (Predicate 1 -> AF (Predicate 2)) and another for infinitely often: AG(AF (Predicate 1)). As the infinitely often notification is the effect and the error condition the cause, we have to replace Predicate 2 in the first template by the infinitely often template. As a result, we obtain the following instantiation AG (error-condition -> AF (AG(AF (warning)))).

We can apply compositions an arbitrary number of times (however, we do not recommend applying more than two or three compositions). In order to explicitly state this iterative composition and not lose track of the meaning of the claim, it is often useful to document the individual compositions as they are applied. In the previous example.

```
-- if an error condition emerges
-- the pilot must be notified infinitely often
-- apply cause effect template :
--      AG (error-condition ->
--      AF (pilot must be notified infinitely often))
-- apply infinitely often template:
--      pilot must be notified infinitely often ==
--      AG( AF ( warning ))
AG (error-condition -> AF ( AG( AF ( warning ))))
```

In some cases, we have considered that a template combination is sufficiently relevant and useful to become a listed template on its own. Remember that the only reason for listing or not listing a claim template is whether it is useful and corresponds to a common expected property.

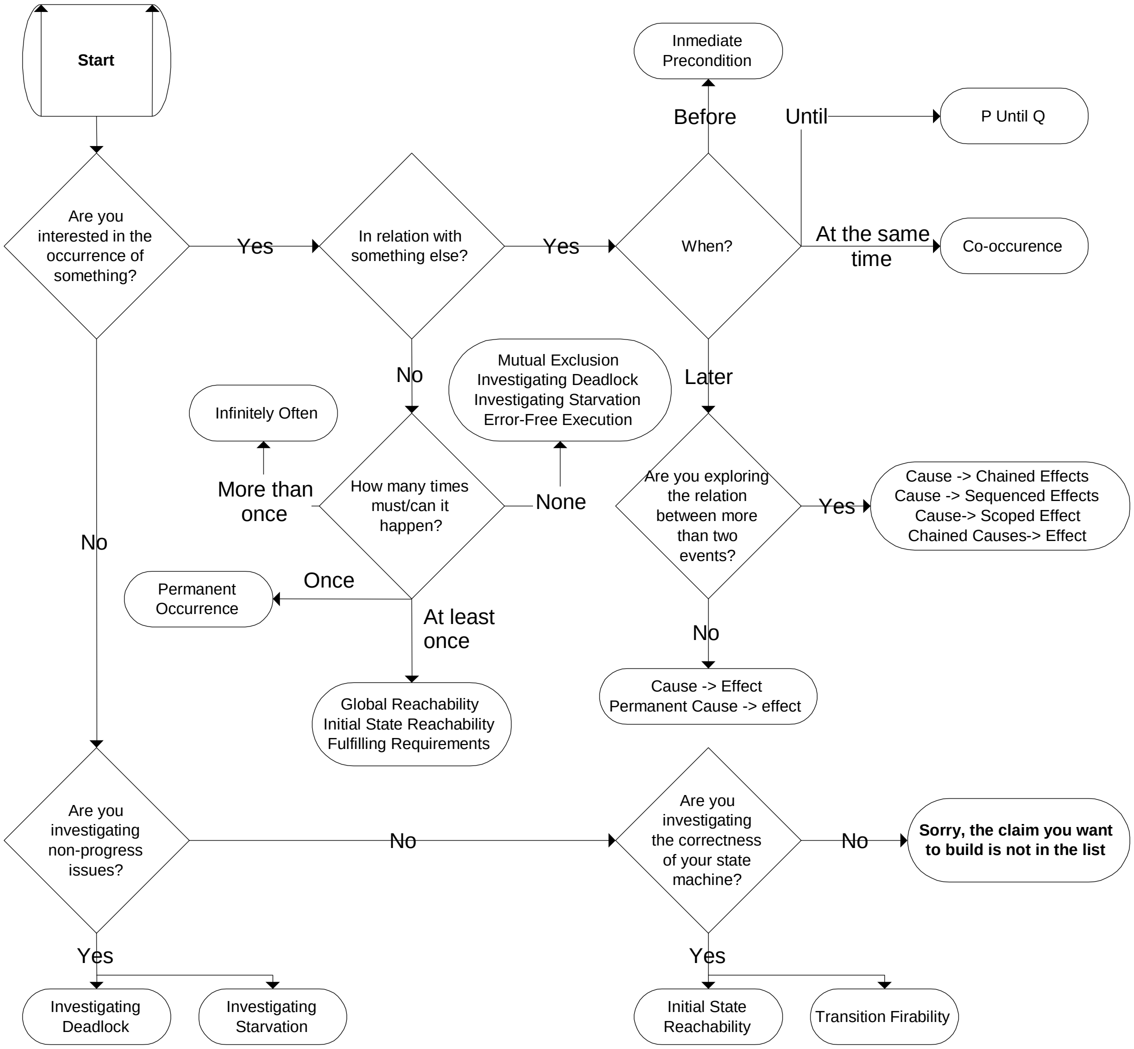
Start
Are you interested in the occurrence of something?
Yes
In relation with something else?
Yes
When?
Before
Inmediate Precondition
Until
P Until Q
At the same time
Co-occurence
Later
No
How many times must/can it happen?
More than once
Infinitely Often
None
Mutual Exclusion
Investigating Deadlock
Investigating Starvation
Error-Free Execution
Once
Permanent Occurrence
At least once
Global Reachability
Initial State Reachability
Fulfilling Requirements
Are you exploring the relation between more than two events?
Yes
Cause -> Chained Effects
Cause -> Sequenced Effects
Cause-> Scoped Effect
Chained Causes-> Effect
No
Cause -> Effect
Permanent Cause -> effect
No
Are you investigating non-progress issues?
No
Are you investigating the correctness of your state machine?
No
Sorry, the claim you want to build is not in the list
Yes
Investigating Deadlock
Investigating Starvation
Yes
Initial State Reachability
Transition Firability

Figure 3: A Diagram for Template Selection

Another useful way to combine templates is through simple logical operators (&, |, ->). In general "&" is used to make a claim more stringent, "|" to make it weaker and "->" to indicate dependencies between claims. For example, suppose the condition "no weight on wheels" can only hold after the plane has taken off: "flying" and "no weight on wheels" must hold thereafter. (This simple model does not consider the plane landing.) In this situation, we can combine the *P until Q* template, the *cause immediate effect template*, and the *permanent occurrence* template into

```
A ( weight-on-wheels U flying ) &
   --until template
   --weight is on the wheels until the plane is flying
AG ( flying ->  !weight-on-wheels) &
   --co-occurrence template
   --when the plane is flying, weight is not on the wheels
AG (!weight-on-wheels-> AG !weight-on-wings)
   --permanent occurrence template
   --there is no weight on the wheels thereafter
```

Every MBV practitioner will eventually need to express an expected property not covered by this template list. In some cases, a similar template can be used as the foundation to develop that specific claim. In others cases, the claim would be completely unrelated to those in the template list. If the expected property formalized in the new claim is common in the domain under consideration, it may make sense to document that new common expected property as a template. This would help to save future work and to ensure a consistent verification of that property in future projects.

5 Template List

The semantics of these templates presume finite state machines and infinite traces.

5.1 Occurrence group

5.1.1 Initial State Reachability

States in the machine can be reached.

```
CTL: EF ( machine_state = state1 ) &
   EF ( machine_state = state2 ) &
   ... --for each state
```

This can be used to show that desired states are reachable and undesired ones are not.

Examples and known uses:

Use for non-trivial models.

Related claims and templates:

Initial state reachability, in contrast to global reachability, only claims that states are reachable from the initial state.

5.1.2 Transition Firability

Transitions in the machine can be fired.

```
CTL: EF ( machine_state=state_x & EX (machine_state=state_y))
      -- there is a transition between state_x and state_y
      EF ( machine_state=state_x & EX (machine_state=state_z))
      -- there is a transition between state_x and state_z
   ... -- for each transition
```

This can be used to show that desired transitions are possible and undesired ones are not.

Examples and known uses:

Use for non-trivial state models.

5.1.3 Global Reachability

A state in which Predicate 1 is true can be reached from any state in the state space.

```
CTL: AG(
   EF (Predicate 1)
)
```

Examples and known uses:

This claim can be used to check that some particular states are reachable in any moment, independent of the condition of the system, e.g., even in error conditions. For example:

The system can always reinitialize

```
AG( EF ( state = reinitializing) )
```

Related claims and templates:

Global reachability is stronger than initial state reachability as the state must be reachable from any other state.

5.1.4 Infinitely Often

Predicate 1 will repeat infinitely often.

```
CTL: AG( AF (Predicate 1))
LTL: G( F (Predicate 1))
```

“Infinitely often” does not connote any particular frequency or regularity; rather, we claim that it will happen again and again. We don’t claim whether it happens twice a second or once a year or sporadically.

Examples and known uses:

There are some periodic actions that the system must perform infinitely often. For example: The system must check the nuclear core temperature infinitely often.

```
AG( AF ( temperature_sensor = on))
```

5.1.5 P Until Q

Predicate 1 is true at least until the first occurrence of Predicate 2 and Predicate 2 will eventually become true (strong until).

```
CTL: A [ Predicate 1 U Predicate 2 ]
LTL: Predicate 1 U Predicate 2
```

Note that Predicate 1 does not need to change to false when Predicate 2 occurs. It may continue to be true.

Examples and known uses:

Sometimes a condition must hold from the initialization of the system until something happens. For example:

```
A [
   Ejection = disabled U
   Plane has taken off
]
```

Note that this is the strong until form that requires the plane to take off. If there is a path where the plane cannot take off (e.g., a take-off abort) then this will evaluate to false. The weak until form (Uw) does not require that the plane take off but rather that ejection is disabled throughout or at least until the plane takes off.

5.1.6 Co-occurrence

Predicate 1 and Predicate 2 always occur simultaneously.

```
CTL: AG (Predicate 1 -> Predicate 2) &
        AG (Predicate 2 -> Predicate 1)
LTL: G (Predicate 1 -> Predicate 2) &
        G (Predicate 2 -> Predicate 1)
```

This does not ensure that Predicate 1 or Predicate 2 will ever happen.

Examples and known uses:

Sometimes a condition must hold when another condition is true. For example: In a dual redundant server system, when the primary reads the data then the backup unit must also read the data and vice versa.

```
AG (
   Primary unit mode = read ->
   Backup unit mode = read
) &
AG (
   Backup unit mode = read ->
   Primary unit mode = read
)
```

This claim can be extended to show that there is a path where the primary unit can be in read mode.

```
EF (Primary unit mode = read)
&
AG (Primary unit mode = read -> Backup unit mode = read) &
AG (Backup unit mode = read -> Primary unit mode = read)
```

Related claims and templates:

Co-occurrence is related to the deferred or immediate precondition template in that Predicate 2 may occur before Predicate 1. But regardless of past history, the co-occurrence template requires that Predicate 2 be a true coincident with Predicate 1.

5.1.7 Error Free Execution

There is an error-free execution of the system. This is one way of expressing safety conditions.

```
CTL: EG ( ! state = error )
```

Examples and known uses:

Most systems should be able to have an execution with no error. Even if the system is intended to run forever regardless of errors, there should be a path with no errors.

5.1.8 Permanent Occurrence

If Predicate 1 ever becomes true, then it is true always thereafter.

```
CTL: AG (Predicate 1 -> AG Predicate 1)
LTL: G (Predicate 1 -> G Predicate 1)
```

This does not ensure that Predicate 1 will happen but if it does become true, it will be true always. This is true throughout all system behaviors.

Examples and known uses:

Sometimes a condition, once it occurs, must always hold. For example, in some rocket launch systems, once the safety release lock is disengaged it must remain disengaged.

```
AG (
   Safety-lock = disengaged ->
   AG (Safety-lock = disengaged)
)
```

Related claims and templates:

- EG (Predicate 1) claims that it is possible for Predicate 1 to be permanent forever, beginning at the initial state. If this claim is true, it is possible for Predicate 1 to occur elsewhere and not be permanent.
- AFEG (Predicate 1) claims that it is inevitable that a state will be reached where it is possible for Predicate 1 to be permanent forever, from the initial state or some state in the future. It is possible for Predicate 1 to occur elsewhere and not be permanent.
- EFEG (Predicate 1) claims that it is possible to reach a state where it is possible for Predicate 1 to be permanent forever. It is possible for Predicate 1 to occur elsewhere and not be permanent.

5.1.9 Mutual Exclusion

Predicate 1 and Predicate 2 do not happen simultaneously (safety condition).

```
CTL: ! EF ( Predicate 1  & Predicate 2 )
```

Examples and known uses:

There are multiple situations that should not happen simultaneously in a system. For example, two processes cannot access the same resource if the resource has exclusive access properties.

```
! EF (
   Subsystem_1_mode = writing in memory  &
   Subsystem_2_mode = writing in memory  )
```

5.2 Cause - Effect group

NOTE: We loosely use the term “cause.” The following temporal logic claims only imply a temporal relation and not a causal one. However, temporal relations are necessary conditions for causal relations, which imply that a counterexample in one of these claims also negates any possible cause-effect relation. The opposite is not true; a positive result does not necessary mean that the causal relation holds.

5.2.1 Cause - Effect

Predicate 1 has Predicate 2 as a future effect such that if Predicate 1 happens (becomes true in a state) then eventually Predicate 2 will happen. This does not guarantee that Predicate 1 will happen.

```
CTL: AG (
   Predicate 1 -> AF (Predicate 2))
LTL: G (
   Predicate 1 -> F (Predicate 2))
```

This does not mean that the effect will immediately follow the event.

Examples and known uses:

If the pilot presses the ejection button, the seat will be ejected.

```
AG (
   Ejection_button = pressed  -> AF (Seat = ejected)
)
```

Note that the seat may be ejected any number of cycles after the ejection button being pressed.

Related claims and templates:

For an immediate effect (in the next state), AX can be used instead of AF.

For a possible but not guaranteed effect, use EF instead of AF.

5.2.2 Permanent cause - Effect

In this claim Predicate 1 must become permanently true in order to cause Predicate 2. This claim does not guarantee that Predicate 1 will become true permanently.

```
CTL: AG (
   AG (Predicate 1) -> AF (Predicate 2))
LTL: G (
   G (Predicate 1) -> F (Predicate 2))
```

Examples and known uses:

In some asynchronous systems, an instantaneous event can go unnoticed, but a continuous condition should not. For example:

System 1 checks system 2 twice a second. We cannot claim that an error in system 2 fires a reaction in system 1 (the error condition may last only a small fraction of a second). We can, however, claim that a permanent error in system 2 will fire a reaction in system 1.

```
AG (
   AG (system2 = error)  ->
   AF (system1 = reacting_to_error_in_system_2)
)
```

Related claims and templates:

For an immediate effect (in the next state), AX can be used instead of AF.

For a possible, but not guaranteed, effect use EF instead of AF.

5.2.3 Cause - Scoped Effect

Predicate 1 has Predicate 2 and Predicate 3 as effects, such that Predicate 2 will follow Predicate 1 and Predicate 2 will hold until Predicate 3 comes true. If Predicate 1 is ever true Predicate 2 and Predicate 3 must happen. This claim does not guarantee that Predicate 1 will become true.

```
CTL: AG (
   Predicate 1  -> A (Predicate 2 U Predicate 3)
)
LTL: G (
   Predicate 1  -> (Predicate 2 U Predicate 3)
)
```

Examples and known uses:

Use to test that the system does some task fired by an event or condition, and keeps doing the task until the task is concluded. It does not guarantee that the request for service occurs.

```
AG (
   request for service -> A[request queued U request complete]
)
```

5.2.4 Cause - Chained Effects

Predicate 1 eventually causes the sequence Predicate 2 followed immediately by Predicate 3.

```
CTL: AG(
   Predicate 1 ->  AF ( Predicate 2 & AX (Predicate 3) )
)
LTL: G(
   Predicate 1 ->  F ( Predicate 2 & X (Predicate 3) )
)
```

Note that Predicate 2 and/or Predicate 3 could have occurred independently, earlier than when the expression (Predicate 2 & AX (Predicate 3)) became true.

Examples and known uses:

In some situations, a happening must fire not one but two sequential consecutive effects in the system.

```
AG(
   Change of state requested -> AF ( previous state exited &
AX (new state entered) )
)
```

Beware of using the expression next (X). It is very difficult to build models that ensure immediate precedence; most of the time, the uncoupled “sequence effects” claim is safer to use.

Related claims and templates:

If the effects are simultaneous, the expression `CTL: AG(p1 -> AF(p2 & p3) )` can be used.

If the effects are not consecutive, the expression `CTL: AG(p1->AF( p2 & AX(p3)))` can be used.

5.2.5 Immediate Precondition

Predicate 2 must be true immediately before Predicate 1 becomes true.

```
CTL: AG (
   EX (Predicate 1)  -> Predicate 2)
LTL: G (
   X (Predicate 1)  -> Predicate 2)
```

Sometimes a function has a precondition that must be valid immediately before something else happens in the system.

Examples and known uses:

Use it when there is a precondition with temporal precedence. For example:

The lock must be open before the rocket is launched.

```
AG (
   EX (rocket = launch) ->
   lock = open
)
```

Note that this does not guarantee that the lock is open during the rocket launch; it only guarantees that it will be open before.

5.2.6 Chained Causes - Effect

The sequence Predicate 1 | Predicate 2 causes Predicate 3 as a future effect.

```
LTL: G(
   (Predicate 1 & X (Predicate 2)) -> F (Predicate 3)
)
```

This claim cannot be built using CTL.

Examples and known uses:

In some situations, two events have no separate effect but they fire some reaction when the happen in immediate precedence. This claim is also useful to eliminate spurious, i.e. conditions that only hold for only one state. For example

```
G(
   (alarm = fired & X (alarm = fired)) ->
   F (evacuation = activated))
```

5.3 Non-Progress (a.k.a. Liveness)

5.3.1 Investigating Deadlock

There is a state in which Predicate 1 becomes permanently true with no option to change.

```
CTL: EF( AG ( Predicate 1) )
```

Examples and known uses:

We typically use negative forms of this claim to detect deadlocks in which the state machine is locked in one state. For example

The system should be able to recover from all non-critical errors.

```
!EF( AG ( state = non_critical_error))
```

In this case, the system is locked in a state (non_critical_error) in which it shouldn't be locked.

5.3.2 Investigating Starvation

There is a state in which and from which Predicate 1 may hold true forever.

```
CTL: EF( EG (Predicate 1))
```

There is a slight but important difference between this expression and the expression EF (AG (Predicate1)). In the case of EF (EG (Predicate1)), it is simply possible for Predicate 1 to be permanently true. It need not be.

Examples and known uses:

The negative form of this claim can be used to detect starvation situations (e.g., possible self loops in a state machine model). Consider the following expected property statement:

```
!EF( EG ( sensor_of_interest = radar))
```

NOTE: This claim will often be false even when there are no defects in the system. We are claiming that there is not a single trace that keeps the radar as sensor of interest forever. Most of the time there will be such a trace; for example, because the other sensors are unavailable or because you have abstracted the replacement algorithm making it non-deterministic. Use this claim cautiously.

6 Conclusions

Claim building is one of the most difficult activities of model-based verification. We have presented a technique that can simplify this activity and make it available to the novice practitioner. The technique uses templates to cover most common expected properties found in requirements specifications. These templates can be instantiated with little effort into specific ready-to-use claims.

The main limitation of this template-based technique is that it only covers a set of common expected properties. Specific properties may not be directly instantiable from the templates listed in this report. Using the templates over time, however, can provide the practitioner with the expertise to customize the templates and even to create completely new claims.

The template list is intended to be a dynamic artifact. We plan to expand it to cover new claims that are found to be useful in day-to-day MBV practice.

References

[Alexander 77]	Alexander, Christopher; Ishikawa, Sara; & Silverstein, Murray. "A Pattern Language." Oxford University Press, New York, NY: Oxford University Press, 1977.
[Appleton 00]	Appleton, Brad. "Patterns and Software: Essential Concepts and Terminology." First appeared in *Object Magazine Online*, *3*,5 (1997). Available WWW< URL:http://www.enteract.com/~bradapp/docs /patterns-intro.html>(1997).
[Clarke 81]	Clarke, E. & Emerson, E.A. "Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic," *Proceedings of the Logic of Program: Workshop*. Yorktown Heights, NY, May 1981. New York, NY: Springer-Verlag, 1981.
[Clarke 86]	Clarke, E.; Emerson, E.A.; & Sistla, A.P. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications." *ACM Transcripts on Program Language Systems 8*, 2 (1986): 244-263.
[Clarke 95]	Clarke, Edmund M., et al. "Verification of the Futurebus+ Cache Coherence Protocol." *Formal Methods in System Design 6*, 2 (March 1995): 217-232.
[Clarke 96]	Clarke, E. M. & Wing, Jeannette. "Formal Methods: State of the Art and Future Directions." *ACM Computing Surveys 28*, 4 (December 1996): 626-643. Also, (CMU-CS-96-178). Pittsburgh, PA: Computer Science Department, Carnegie Mellon University, 1996.
[Corbett 00]	Corbett, James C.; Dwyer, Matthew B.; Hatcliff, John; & Robby. "A Language Framework For Expressing Checkable Properties of Dynamic Software." *Lecture Notes in Computer Science.* New York, NY: Springer-Verlag, 2000.
[Dillon 94a]	Dillon, L. K.; Kutty, G.; Moser; L. E.; Melliar-Smith P. M; & Ramakrishna, Y. S. "A Graphical Interval Logic for Specifying Concurrent Systems," *ACM Transactions on Software Engineering and Methodology 3*, 2 (April 1994).

[Dillon 94b] Dillon, L. K.; Kutty, G.; Moser, L. E.; Melliar-Smith, P. M.; & Ramakrishna, Y. S. “Visual Specifications for Temporal Reasoning,” *Journal of Visual Languages and Computing 5*, 1 (1994): 61-81.

[Dwyer 98] Dwyer, Matthew B.; Avrunin George S.; & Corbett, James C. “Property Specification Patterns for Finite-State Verification,” Proceedings of the Second Workshop on Formal Methods in Software Practice. Clearwater Beach, FL, March 4-5, 1998

[Dwyer 99] Dwyer, Matthew B.; Avrunin George S.; & Corbett, James C. “Patterns in Property Specifications for Finite-State Verification,” Proceedings of the 21st International Conference on Software Engineering. Los Angeles, CA, May, 1999.

[Gamma 94] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA: Addison-Wesley, 1994.

[Gluch 98] Gluch, D. & Weinstock, C. Model-Based Verification: A Technology for Dependable System Upgrade (CMU/SEI-98-TR-009, ADA 354756). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. Available WWW< URL:http://www.sei.cmu.edu /publications/documents/98.reports/98tr009/98tr009abstract.html>(1998).

[Grover 93] Grover, Claire; Carroll, John; & Briscoe, Ted. The Alvey Natural Language Tools Grammar (4th release). (Technical Report 284) Cambridge, England: Computer Laboratory, University of Cambridge, 1993. Available FTP: URL <ftp://ftp.cl.cam.ac.uk/nltools/reports/grammar.ps>

[Harel 87] Harel, D. “Statecharts: A Visual Formalism for Complex Systems.” Science of Computer Programming 8, 3 (June 1987): 231-274.

[Holt 99] Holt, Alexander. “Formal Verification With Natural Language Specifications: Guidelines, Experiments and Lessons so Far.” South African Computer Journal, 24, (November 1999): 253-257.

[McMillan 92] McMillan, K.L. Symbolic Model Checking: An Approach to the State Explosion Problem (CMU-CS-92-131). Pittsburgh, PA: Computer Science Department, Carnegie Mellon University, 1992.

[Riehle 96] Riehle, D. & Zullighoven H. “Understanding and Using Patterns in Software Development.” Theory and Practice of Object Systems 2,1 (April 1996): 313.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** October 2001	3. **REPORT TYPE AND DATES COVERED** Final

4. **TITLE AND SUBTITLE**
Model-Based Verification: Claim Creation Guidelines

5. **FUNDING NUMBERS**
F19628-00-C-0003

6. **AUTHOR(S)**
Santiago Comella-Dorda, Dave Gluch, John Hudak, Grace Lewis, Chuck 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-018

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)

Model Based Verification (MBV) is a systematic approach to finding defects (errors) in software requirements, designs, or code. MBV involves creating essential models of system behavior and analyzing these models against formal representations of expected properties, known as claims. Claim generation has been identified as a particularly complex activity within model-based verification. This technical note describes a pattern-based approach to facilitate claim generation. The report includes a list of directly usable patterns for the most frequent expected properties found in system specifications.

14. **SUBJECT TERMS**
Model Based Verification, MBV, template-based approach, claim building, occurrence templates, cause-effect templates, non-progress templates

15. **NUMBER OF PAGES**
33

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

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102