A Survey on Kernel Specification and Verification

Ahmed Helmy
email: ahelmy@usc.edu

May 3, 1997

Abstract

Formal methods have been traditionally used to model and verify operating systems. Different methods verify different operating systems properties, such as process management, mutual exclusion and inter-process communication. Moreover, various methods may capture different design errors, such as deadlocks or unspecified receptions.

The system kernel supports higher-level system services. Hence, kernel verification is essential for the proper operation of the system. In addition, providing clear kernel specification improves the interoperability between its various implementations.

In this paper, we describe commonly used methods for kernel specification and verification. Some methods provide a mathematical model, and use logic to prove properties of interest. These include PVS and Boyer-Moore logic. Others use a programming language to simulate the system, then apply verification tools to capture system errors. These include the SPIN tool.

Distributed operating systems are susceptible to unexpected failure events, complicating the issue of system robustness. This issue is not addressed explicitly by traditional verification methods. We present a new STRESS method that can be used to analyze system robustness. STRESS is based on a simulation framework, and facilitates fault simulation for distributed operating systems and kernels.

Finally, our comparison shows that some of the methods discussed are complementary. Thus, several methods may be used to obtain better fault coverage of the target system.

Keywords: Operating Systems, Kernel, Formal Methods, Verification, Specification, Robustness.

1 Introduction

In general, kernels are modeled as communicating finite state machines (FSM). The definition of a FSM requires a set of machine states, and a definition of transitions on these states [6]. In our case, kernel operations control the changes to the machine states.
Goals for obtaining a kernel model are manifold. First, to provide a precise documentation, by defining the required behavior of the kernel interface, and ignoring implementation issues. Second, to define a contract between kernel users and implementors. This enhances the program's portability, by providing unambiguous statement of the required features of a kernel implementation. Third, to facilitate proof of correctness of application programs which run on the kernel, using mechanical proof checkers [9, 2].

In this research paper, a survey of the methods used commonly to verify and validate kernel specifications and implementations is presented. This study is not meant to be exhaustive, rather illustrative with emphasis on the limitations and shortfalls of the conventional methods.

In addition, we describe a new method, based upon a simulation model, to analyze the robustness of distributed operating systems and kernels under loss and failure scenarios. We call our method Systematic Testing of Robustness by Examination of Selected Scenarios (STRESS).

All the methods described share a common ultimate goal, that is to test correctness aspects of the system\(^1\). In general, correctness implies the absence of errors. System errors are discussed next.

1.1 System Errors

There are two types of errors addressed by kernel design: (1) design errors, and (2) operational errors [1].

Design errors address: (a) safety, (b) liveness, and (c) responsiveness, properties of a system. Safety properties ensure that the system never enters undesirable states and include freedom from deadlocks, assertion violations, improper termination and unspecified receptions. Liveness properties ensure that the system performs its intended functions with respect to service specification, and include detection of acceptance cycles (those that pass through acceptance states) and absence of non-progress cycles (those that do not pass through any progress-state). Responsiveness properties include timeliness, and fault tolerance, which recovers the system to a legal state to resume normal execution from an illegal state.

Operational errors relate to the implementation environment. Recovery mechanisms deal with transient operational errors, which may change the state of a system, but may not change its behavior.

This wide spectrum of potential errors motivated the development of various approaches for correct system design. A high-level classification of these approaches is outlined next.

1.2 Taxonomy

The main issues addressed by the methods discussed in this paper are specification and verification.

---

\(^1\) Throughout this document we conveniently interchange system and kernel, since the kernel is the target system of this study.
**Specification** is the process of describing a system and its desired properties. Formal specification uses a language with a mathematically-defined syntax and semantics. System properties include functional behavior, timing behavior, performance characteristics, or internal structure.

**Verification** goes one step beyond specification, to analyze a system for the desired properties. The two approaches for verification are **theorem proving** and **model checking** [7].

- **Theorem proving**
  
  In theorem proving, system properties are expressed in logic formulas, defining a set of axioms and rules. In contrast to model checking, theorem proving can deal with infinite state spaces. However, interactive theorem provers require human intervention, and hence are slow and error-prone.

  Theorem proving includes **model-based** and **logic-based** formalisms. Model-based formalisms such as Z and VDM are suitable for kernel specifications in a succinct manner, but lack the tool support for effective proof of properties. The use of first order logic allows the use of theorem provers such as Nqthm, but may result in specifications that are difficult to read. Higher order logic, such as PVS, provides expressive power to provide clear descriptions, and proof capabilities for kernel properties.

  We shall describe the application of Nqthm and PVS to kernel verification, in section 2.1.

- **Model checking**
  
  Model checking relies on building a finite model of a system and checking that a desired property holds. It is usually used for **reachability analysis** [13]. The check is performed as an exhaustive search, and may suffer from the state explosion problem.

  In this model, a program is used to simulate the behavior of the system. SPIN and STRESS fall under this category, and are described further in section 2.2.

For brevity, we shall only address a subset of these methods, that have actually been used for modeling and verification of kernels. The rest of the paper is organized as follows. Section 2 discusses the surveyed methods with the corresponding case studies. A comparison of the methods is presented in section 3. Finally, section 4 provides the conclusion of this study.

# 2 Verification Methods

In this section we discuss individual verification tools, and present some case studies on kernel verification. As previously discussed, verification methods include theorem proving and model checking. Theorem proving is presented first. This discusses the Nqthm prover, with a case study on Mach [4, 3], and the PVS system with a case study on a real-time kernel [8]. Then model checking
methods are discussed, presenting the SPIN tool \cite{11,12}; with a case study on the Harmony kernel \cite{5}. Our new STRESS method concludes this section.

\section{Theorem Proving}

A kernel state consists of entities such as memory objects, processes, and threads. Axioms constrain the relations between entities in a valid kernel state. This is illustrated by the following case studies.

\subsection{Mach Math model using Nqthm}

This case study investigates a simple model of some properties of the Mach kernel using the Boyer-Moore logic and Nqthm.

**Nqthm**  Nqthm is an automated reasoning system, also known as ‘the Boyer-Moore Theorem Prover’. Nqthm is a Common Lisp program for proving mathematical theorems. The logic of Nqthm is a quantifier-free first order logic with equality. The rules of inference of the logic are those of propositional logic and equality with the addition of mathematical induction. Using Nqthm logic, concepts and axioms are introduced, and are used to derive lemmas. New functions must satisfy the concepts and axioms. To better understand how such logic is used in kernel modeling, we discuss its application to Mach.

**Mach math model**  Mach is a micro-kernel that minimizes the abstractions provided by the kernel and provides a rich set of semantics associated with these abstractions.

The Mach kernel state consists of entities such as tasks, threads, ports, messages, pages, and memory objects.

Entities may participate in relations. For example, a port right is a relation involving a task and a port that characterizes a task’s capability on a port. Axioms place constraints on the relations that may hold in a kernel state – for instance, at most one task may hold a receive right on a given port. The description of entities, relations, and constraints characterizes a legal Mach state.

Kernel requirements are expressed as a collection of functions, predicates and axioms. A recognizer predicate is identified for each Mach entity class. For example, an entity \( x \) that satisfies the predicate \( \text{task}p(x, \sigma) \) is a task in state \( \sigma \). A thread is recognized by the predicate \( \text{thread}p(x, \sigma) \).

Following are four axioms defining the relations between tasks and threads.

**Axiom 1**  Disjointness of \( \text{task}p \) and \( \text{thread}p \)

\[ \text{task}p(x, \sigma) \rightarrow \neg \text{thread}p(x, \sigma) \]

**Axiom 2**  The relation \( \text{task} \downarrow \text{thread} \) formalizes thread ownership. The expression \( \text{task} \downarrow \text{thread}(t, th, \sigma) \) holds in state \( \sigma \) if \( th \) is a thread in task \( t \).

\[ \neg \text{task}(t, \sigma) \lor \neg \text{thread}(th, \sigma) \rightarrow \neg \text{task} \downarrow \text{thread}(t, th, \sigma) \]

**Axiom 3**  A thread can only be owned by one task

\[ \neg \text{thread}(x, \sigma) \rightarrow \neg \text{thread}(x, \sigma) \]
\[ \text{taskThread}(t_1, \text{th}, \sigma) \land \text{taskThread}(t_2, \text{th}, \sigma) \rightarrow t_1 = t_2 \]

**Axiom 4** The set of threads associated with task \( t \) is expressed as \( \text{threads}(t, \sigma) \). A thread \( \text{th} \) is an element of \( \text{threads}(t, \sigma) \) if and only if it is owned by \( t \).

\[ \text{th} \in \text{threads}(t, \sigma) \leftrightarrow \text{taskThread}(t, \text{th}, \sigma) \]

To define the Mach state we construct the *closures* for the axioms. For example, the closure of Axiom 1 is

\[ \forall x \ (\text{taskp}(x, \sigma) \rightarrow \neg \text{threadp}(x, \sigma)) \]

**Definition** A legal Mach state can be defined as the conjunction of the closures of the axioms.

\[ \text{legalState}(\sigma) \equiv \forall x \ (\text{taskp}(x, \sigma) \rightarrow \neg \text{threadp}(x, \sigma)) \land \ldots \]

Kernel transitions (e.g. Mach atomic actions) must preserve legality; i.e. produce legal state from legal state.

One can methodically examine all the relations given and identify the desired actions in each class which can be derived from that relation. This procedure, however, is not entirely algorithmic since within each class practical issues must be addressed.

**Limitations** The Mach model (for atomic actions and locks) includes more than 50 relations, 400 axioms and 250 functions. The fact that axiomatization is not algorithmic, may limit the use of Nqthm.

### 2.1.2 Real-time kernel model using PVS

We present a specification analysis of a real time kernel using the PVS (Prototype Verification System) system.

**PVS** PVS is a verification system: that is, a specification language integrated with support tools and a theorem prover. It is based on classical higher order logic, extended with a typing system. The following case study gives the approach of PVS to kernel specification and verification.

**Real-time kernel specification** Real-time operating systems are vital for safety critical systems and their correctness is essential for the integrity of such systems. Here, we present a simplified specification of a real-time kernel. The specification includes abstract model of the functional and timing requirements for the kernel, and a minimal model of the system supported.

The kernel requirements include no dynamic task creation (for simplify of analysis), asynchronous IPC via shared data allowed by Ada’s protected objects, and mutual exclusion provided by a priority protocol.

The properties of the kernel that need to be verified are: (a) the highest priority task is executed, and (b) mutual exclusion is maintained for protected objects.
The approach produces a high level kernel specification, including minimal model of application program and interactions between application and kernel. This provides a description of the kernel functionality, and application properties required for proper operation of the system. The model of interactions allows global system properties to be analyzed with the assistance of a theorem prover.

The high level kernel specification comprises of the timing properties, the kernel environment and the kernel state.

Timing properties are expressed using real-time logic (RTL). The logic is based on an event-action model, where actions are time consuming sections of work, and events are temporal markers, corresponding to external stimuli, the start and end of actions, and system state changes.

The basic types for the kernel environment include TASK, protected objects (PO) and interrupts. The state of a task can be either ‘ready to run’ or ‘suspended’. For protected objects, the value of the guard for each PO entry is updated by the kernel under mutual exclusion. Axioms are used to express these requirements. For example, each task is assigned a unique base priority.

The kernel state consists of state variables, state of each task in the system and state of each protected object.

The kernel operations consist of Delay operations (to suspend tasks), operations on protected objects, and the Dispatcher. Interfaces between kernel and the system in terms of kernel state and defined operations. The state of the system is the state of the kernel combined with the state of the program.

To execute the given operation in the system, the good-operation predicate is evaluated. It consists of the conjunction of the following four properties:

(a) no potentially blocking operations are executed by a task within a protected object,
(b) a task can only exit from a protected object if it is in one,
(c) the object must be available for a task that is attempting to enter it, and
(d) the object from which a task is exiting must be the last one that it entered.

The operation of the system is modeled as a finite sequence of operations. Each operation takes a finite, bounded time to execute, ensuring the termination of execution.

Sequences of kernel operations are modeled by the type HISTORY, and then the state of the kernel after such a sequence is defined by an interpreter function \( \text{Int} \). \( \text{Int} \) takes an initial state and a history, and returns the state of the kernel after this sequence of operations.

Modeling all possible states, in which the system may exist, is achieved by applying the interpreter function to the initial system state with all possible sequences of operations. Hence the validity of an invariant property, \( P(s) \), is expressed as:

\[
\forall (h : \text{History}) : P(\text{Int} (\text{Init}, h))
\]

where \( \text{Init} \) is the initial state of the system.
The general class of properties that can be analyzed in this manner are those that can be expressed as an invariant on the system state.

**Limitations**

(a) In general, PVS suffers a similar limitation to that of Nqthm, which is the definition of a large number of relations and axioms.

(b) PVS is a large complex system. Learning it may consume a long time and requires lots of expertise.

(c) Another limitation is related to the use of Ada. Ada rendezvous requires two processes to both reach synchronization points, which can cause unbounded delays and is not easily amenable to standard analysis techniques.

### 2.2 Model Checking

One may classify model checking methods into *logic model-checker* and *on-the-fly*. *Logic model-checker* works with a two-pass verification process. In the first pass the basic behavior of the system is explored and an abstract representation is obtained. In the second pass, the representation is used to prove or disprove the system correctness.

*On-the-fly* verification works with one-pass verification process. It stores in memory minimum information needed for correctness verification, hence on-the-fly systems can handle larger problem sizes and faster than a model checker. SPIN lies in this category, and is discussed next.

#### 2.2.1 Harmony kernel verification using SPIN

**SPIN**

SPIN is a general verification tool for proving correctness properties of distributed or concurrent systems. These systems interact through shared memory, rendezvous operations or buffered message passing. Problems that these interactions may create can be debugged by SPIN. Once the system design is obtained, a proof of its correctness can be provided.

SPIN uses the validation modeling language PROMELA, which is based on Hoare’s CSP language [10]. PROMELA models communicating sequential processes which may be created dynamically and communicate asynchronously or synchronously by messages through channels.

SPIN supports the following features:

(a) **Automated verification** based on the principles of reachability analysis [13], with the use of space reduction techniques.

(b) **Complexity profiling** statistics gathered during verification can be used to identify the hot spots in the validation model.

(c) **Search for safety and liveness properties** Safety errors captured include assertion violation, invalid end states, unspecified receptions, and unreachable code segments. Also liveness properties, such as acceptance cycles and non-progress cycles, are detected.

Two types of simulations are supported: (i) random simulation that uses a randomizer for non-deterministic selection of the process scheduling algorithm, and (ii) guided simulation, where an
error trail in terms of states or transitions is followed. A guided simulation can only be performed if a verification run was done first that revealed an error in the system.

SPIN is intended to be both a testbed for the development and evaluation of new verification techniques and to present an environment for verification of concurrent systems.

**Harmony** Harmony is a real-time multitasking multiprocessor operating system. It is based on a microkernel and system servers. It features interrupts, priority preemptive scheduling, intertask communication (send-receive-reply scheme), and dynamic tasks. Mutual exclusion across processor boundaries is achieved by an ownership protocol rather than locks.

The first step is to formalize the models of the system, the scenarios and the properties that are to be checked. Focus is given on deadlocks, livelocks and other safety properties for the primitives of Harmony. PROMELA channels are used for abstracting intertask communications. Then exhaustive verification of the intertask communication and task management features of Harmony was carried out by model-checking.

Following we discuss modeling and verification of intertask communication and task management.

I) The **intertask communication** model is composed of the following modules: (a) kernel module, including parameters such as number of processors, and definitions of each part of the system and scenarios, (b) task descriptor module, including the task state, its correspondent, its processor and its send queue, (c) processor module, models the interrupt masking primitives. Local mutual exclusion relies on interrupt masking, (d) interrupt model, uses non-deterministic scheduling, and (e) queuing primitives.

Verification of intertask communication includes absence of deadlocks or livelocks, appropriate return code, and checking that all the used code. To achieve this, an attempt is made to find a set of scenarios that exhibit all possible communication schemes. Reduction techniques can be used to reduce the number of scenarios investigated, for example, by considering system symmetries (such as, all processors are considered identical).

II) **Task management** and creation may be dynamic, but may introduce infinite state space at validation time. Hence, a bounded number of application tasks is scheduled at initialization. The Interprocessor interrupt handling takes priority over the running tasks if they are not masked.

Verification is attained by simulating and verifying basic scenarios involving the use of kernel primitives.

**Limitations**

(a) The fact that the validator indicates that all the code is reached does not prove that every situation in the code was reached, though. For example,

\[
\text{if } (c1 \&\& c2) \text{ then proc1 else proc2}
\]

Four possible values for \(c1\) and \(c2\), but may encounter only 2 of them to cover the code.

(b) It takes a long time to get acceptable models for validation.
(c) Even with a partial model and a restricted scenario, we may be faced with a state explosion. Partial order reduction technique integrated into SPIN ameliorates this problem. Nevertheless, high rate of communication interaction obviates the benefit of reduction techniques.

(d) When the problem size is large, it is difficult to follow the path that lead to the error.

### 2.2.2 STRESS

One of the shortfalls of the methods discussed so far, is that they do not explicitly support fault and error modeling. In reality, systems consist of various software and hardware components. The failure of any component may affect the correctness of the overall system. In order to design robust and correct systems, it is crucial to consider failure scenarios as an integral part of the system design.

This motivates our next method, which we call ‘STRESS’, for Systematic Testing of Robustness by Examination of Selected Scenarios. By robustness we refer to the ability of a system to react correctly in the face of failures. For brevity, we shall only present an overview of STRESS.

STRESS is based on a simulation framework, and supported by a set of verification tools. The main approach aims to capture and analyze a set of error prone scenarios, under various failure and loss conditions. This is achieved by investigating representative parts of the state space, and the definition of error conditions. As shown in figure 1, the three main stages of our approach are: (a) scenario generation, (b) tracing, and (c) output analysis.

#### Scenario generation
Scenarios are the collection of topologies and sequences of events (input stimuli and state transitions), that describe the simulation context. Analysis of these sequences can unveil weaknesses in the system design. Elements of a scenario include:

(i) **topology**: is the system infrastructure, including the communication bus(es) and interconnect(s), memory, registers, and queues. Topology modules are designed with full controllability over failures and loss. For example, a queue delivers messages to a programmable loss module, that either delivers the message or drops it.

(ii) **end-point scenarios**: are the combination of possible actions by the end point. Depending on the system modeled, the end point may correspond to micro-actions, such as read/write to
registers, or macro-actions such as exchanging messages between user tasks. We adopt the latter approach in general, where the end point is the user task.

To reduce the number of end-point scenarios, we introduce the notion of representative scenarios. These are simple scenarios that cover a large portion of the state space of (part of) the kernel under study. A scenario filter is used in choosing such scenarios, removing impractical combinations or redundant events due to symmetry or equivalence in topology.

(iii) failures: include (a) loss or corruption of data on the communication bus, and (b) loss of state kept in memory or in registers.

**Tracing** Tracing collects information during simulation. Traces are used in the output analysis stage, and include:

(i) **end-point** or (macro) traces, giving information about end-point events, such as a system call and its returned code.

(ii) **state transition** or (micro) traces, providing fine grain information about the kernel detailed transitions, such as reads and writes to registers.

(iii) **specific** traces, for given message type, call type or component. For example, one may want to trace the event of opening raw sockets as a call. This information is fed back into the scenario generator to program the infrastructure loss modules to lose such a call in further simulations. This guides the simulation, and gives controllability over the loss scenarios.

(vi) **annotated code** tracing, for use in the profiler in a later stage. Partial tracing of key points is allowed, reducing the time and space required to trace other pieces of code.

**Output Analysis** At this stage the data traced during the simulation are analyzed for errors and profiling. It includes the following elements:

(i) **end-point errors** are extracted from the end-point trace, according to a specified error condition. The error condition may involve correctness or performance. For example, a correctness error may indicate the loss of a return code. While a performance error may indicate the arrival of the return code after a time limit. End-point errors are typically used to guide the analysis of transition errors in the kernel.

(ii) **transition errors** are obtained from both the end-point errors and the transition traces. After an end-point error is discovered, the trace is rolled-back in time to relate that error to an error in the kernel transitions.

(iii) **profiling** conveys information about the segments of annotated code that were not executed, and the hot spots in the executed code. This is similar to reachability analysis.

Once the simulation environment is set-up, the verifier runs the simulations, feeds back the specific traces, then analyzes the output. To obtain the scenarios causing the error, the basic steps are revisited in reverse order; i.e. from the output analysis to the tracer then to the scenario generator.
The scenarios obtained can be further used to stress kernel implementations.

**Limitations**  
(a) Code coverage may not correspond directly to state coverage, and is not accurate as has been shown for SPIN.  
(b) May suffer explosion in state space and number of failure scenarios. Scenario and space reduction techniques are used to ameliorate this problem. Efficiency of such techniques depends on the feedback from the specific trace, and the size of the system and topology.  
(c) If the error trail is long, it may be hard to follow. Nonetheless, the roll-back technique, and the use of end-point traces, guide the error trace.

3 Summary and Comparison  
In this section we present a summary and comparison for general properties of the discussed methods. Although the comparison is not detailed, it gives an outline of the major differences we have inferred from the study.

<table>
<thead>
<tr>
<th></th>
<th>Nqthm</th>
<th>PVS</th>
<th>SPIN</th>
<th>STRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Verification</strong></td>
<td>formal proof (first order logic)</td>
<td>formal proof (higher order logic)</td>
<td>reachability analysis (state &amp; code coverage)</td>
<td>reachability analysis (code &amp; scenario coverage)</td>
</tr>
<tr>
<td><strong>Used for</strong></td>
<td>Specification &amp; analysis</td>
<td>Specification &amp; analysis</td>
<td>Analysis &amp; partial conformance testing</td>
<td>Analysis &amp; implementation stress testing</td>
</tr>
<tr>
<td><strong>Robustness</strong></td>
<td>no explicit support</td>
<td>no explicit support</td>
<td>no explicit support</td>
<td>supports loss &amp; failure analysis</td>
</tr>
<tr>
<td><strong>Limitations</strong></td>
<td>No. of axioms</td>
<td>No. of axioms</td>
<td>state explosion</td>
<td>scenario explosion</td>
</tr>
<tr>
<td><strong>Reusability</strong></td>
<td>low</td>
<td>low</td>
<td>moderate (channels)</td>
<td>high (infrastructure)</td>
</tr>
</tbody>
</table>

As shown in the table, formal methods (e.g. Nqthm or PVS) provide precise documentation and hence can be used for specification as well as analysis through proofs. On the other hand, SPIN and STRESS are mainly used for analysis and testing.

All methods have limitations. For SPIN and STRESS, reachability analysis, suffers from the state space explosion. Both methods, however, develop space reduction techniques to circumvent this problem. With the availability of more computation and storage resources, we do not consider this a severe limitation. For PVS and Nqthm, the number of relations, and axioms becomes a problem with the growth of the system.

For Reusability, formal methods re-define entities, relations and axioms for the new system. SPIN provides communication channels, while STRESS provides a system infrastructure that can be re-used.

4 Conclusion  
We have presented various systematic methods for kernel verification. These methods address a wide array of system errors, such as liveness, safety, responsiveness and operational errors.
Verification methods use either theorem proving or model checking. Theorem proving methods be used for specification as well as analysis, but require the construction of mathematical proofs. Model checking methods are used in analysis and testing, but may experience state explosion problems.

Our comparison shows that for large systems, no single method can be used effectively to specify, analyze and test the system. Integration of these methods in a useful manner may solve this problem, but requires clear understanding of the methods and the target system.

References


