Kit: A Study in Operating System Verification

William R. Bevier

Technical Report 28 August, 1988

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This research was supported in part by the U.S. Government. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government. This work was sponsored in part at Computational Logic, Inc. by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151, and at the University of Texas at Austin by the Defense Advanced Research Projects Agency, ARPA Order 5246, issued by the Space and Naval Warfare Systems Command under Contract N00039-85-K-0085.

1. Introduction

One of the fundamental goals of a multi-tasking operating system is the implementation of processes. Two processes running on the same machine should be isolated so that execution of one does not interfere with the other in unintended ways. Without this property, neither process can be expected to execute correctly even if each is executing a "verified" program.

The purpose of the Kit project is to address the problem of verifying a process isolation property for a multi-tasking operating system. Kit is a small operating system kernel written for a uni-processor computer with a simple von Neumann architecture. (The name *Kit* suggests the phrase *kernel for isolated tasks*.) It is proved at the machine code level to implement a fixed number of conceptually distributed communicating processes. The kernel provides the following verified services:

- Process scheduling and allocation of CPU time,
- Response to program error conditions,
- Single-word message passing among processes,
- Character I/O to asynchronous devices.

It is important to say what Kit does not do. There is no dynamic creation of processes or communication channels. There is no file system. Inter-process communication occurs only by message passing; there is no shared memory. While Kit is not big enough to be considered a kernel for a general purpose operating system, it does confront some important operating system phenomena. It is adequate for a small special purpose system such as a communications processor.

A requirement on the Kit project is to treat the problem of process isolation in a sufficiently complex setting. Primarily, we want to consider interrupts. We want Kit to maintain process isolation even in the face of asynchronous events arriving at the target machine. We prove, for example, that the context switch operation on an interrupt correctly maintains the state of all processes.

The state of a process (i.e., the set of all data to which a process has access) is machine dependent at the operating system layer in which interrupts are handled. Typically, a process has access to a set of CPU registers, some primary and secondary store, and a number of external devices. Therefore the complete description of a process state requires information about the architecture of the machine on which the process runs.

These considerations motivated the decision to verify Kit at the machine code level. If we understand a computer's fetch-execute cycle, then we can precisely consider the effect of interrupts on program execution. In addition, at the machine code level we have complete information about a target machine, so that we can answer questions about the state of a process. A machine code instruction is defined by its effect on the entire state of a machine so we can determine both what it does and what it does not do. This is critical for considering process isolation which, after all, is concerned with what processes may not do.

This paper is a summary of a longer report [Bevier 87], which is itself a summary of the script of definitions and theorems which make up the specification, implementation and proof of Kit. The Kit script contains

- a formal definition of a communicating process,
- a formal specification (called the *abstract kernel*) of an operating system kernel which manages a fixed number of communicating processes,
- the proof of a theorem which states that the abstract kernel correctly implements each process,

- a formal definition of a von Neumann machine (the target machine, **TM**) on which to implement the operating system kernel,
- the machine code implementation of the kernel, and
- the proof of a theorem which states that the machine code running on **TM** correctly implements the abstract kernel.

The fundamental assumption of this work is that the target machine definition is a legitimate specification for the architecture level of a von Neumann computer and *could* be implemented in hardware.

A process, the abstract kernel, and the target machine are each defined as a finite state machine. Section 2 explains how we state these definitions, and explains the form of the correctness theorem which relates two finite state machines. The remainder of the paper summarizes each of the points mentioned above. See [Bevier 87] for a more detailed discussion.

The formalization of Kit is done in the Boyer-Moore logic, and all proofs are mechanically checked by the Boyer-Moore theorem prover [Boyer 88]. A brief summary of the logic can be found in Appendix A. The logic is very similar to pure Lisp. Except in a few places in section 2 we use the notation of the Boyer-Moore logic, which is the prefix syntax of pure Lisp. For example, we write **(PLUS I J)** where others might write *PLUS*(*I*,*J*) or *I*+*J*. We write

```
(IMPLIES (P X) (EQUAL (F X) (G X)))
```

in place of $P(X) \rightarrow F(X) = G(X)$. We write **(IF A B C)** instead of the expression *IF a THEN b ELSE c*. We use a Lisp convention for naming functions: the names of predicates have a "**P**" at the end. For example, the expression **(ZEROP X)** is Boolean-valued and tests for zero.

2. Interpreters and Interpreter Equivalence Theorems

Kit is verified by proving a correspondence between the behavior of two finite state machines. An abstract finite state machine serves as an operational specification. The kernel running on the bare computer is also defined as a finite state machine. In this section we explain how we define finite state machines, and describe the form of the correspondence theorem between two machines. As an example, we state the correspondence theorem which establishes that Kit's specification (the abstract kernel) is correctly implemented by the Kit machine code.

2.1 Machine States, Interpreters and Oracles

The definition of a finite state machine requires two things: a description of the set of machine states, and a definition of each transition on a machine state. We define the set of machine states by a predicate which recognizes a member of the state set, a so-called *good state*. We define the transitions on a finite state machine by an *interpreter function*.

The state of a machine is typically a record structure. The state of a von Neumann computer may contain, in a simple example, the fields **MEMORY**, **REGISTERS**, **FLAGS**, **PROGRAM-COUNTER**. In the Boyer-Moore logic we define such a record structure with the following notation.

```
Shell Definition.
Add the shell COMPUTER with recognizer COMPUTERP,
defining the record structure
<MEMORY, REGISTERS, FLAGS, PROGRAM-COUNTER>.
```

The expression **(COMPUTER M R FL PC)** represents a computer state with memory **M**, registers **R**,

flags **FL** and program counter **PC**. The expressions **(MEMORY X)**, **(REGISTERS X)**, **(FLAGS X)** and **(PROGRAM-COUNTER X)** access, respectively, the memory, registers, flags and program-counter fields of a computer **X**. If **X** is a computer as defined above, the following expression represents the computer state equal to **X** in every field but the program counter, which in this case takes on a value one less than **X**'s program counter.

```
(COMPUTER (MEMORY X)
          (REGISTERS X)
          (FLAGS X)
          (SUB1 (PROGRAM-COUNTER X)))
```

We place type restrictions on the fields of a record structure by defining a predicate which constrains each shell field. In the **COMPUTER** example, we can constrain each field to be either a string of bits (i.e., a *word*) of a certain size, or a finite array of words. In the following example, the predicate **GOOD-COMPUTER-STATE** constrains memory to be an array of length 2^{16} of 16-bit words, the registers to be an array of length 8 of 16-bit words, the flags to be a 4-bit word and the program counter to be a 16-bit word. (Assume **WORDP** recognizes a word of a given size, and **WORD-ARRAY** recognizes an array of words of a given size.)

```
DEFINITION
(GOOD-COMPUTER-STATE X)
=
(AND (WORD-ARRAY (MEMORY X) 16)
     (EQUAL (LENGTH (MEMORY X)) (EXP 2 16))
     (WORD-ARRAY (REGISTERS X) 16)
     (EQUAL (LENGTH (REGISTERS X)) 8)
     (WORDP (FLAGS X) 4)
     (WORDP (PROGRAM-COUNTER X) 16))
```

An interpreter function models transitions on a machine over an finite but arbitrary time span. It is a dyadic function of the form *Int*: $S \times O \rightarrow S$, where S is a set of machine states and O is a set of oracles for a machine. An oracle has two roles. It determines the finite time span for which a machine invocation operates, and it may introduce non-deterministic state changes into a machine, including communication with other machines.

In a simple situation the set of natural numbers N can be chosen as the oracle set. An interpreter of the form *Int*: $S \times N \rightarrow S$ models a machine which operates in complete isolation. Such a machine can be defined in the Boyer-Moore logic as shown below. The function **STEP** advances the state of this machine. The expression **(MACHINE1 STATE N)** is the state obtained by applying **N** successive applications of **STEP** to **STATE**.

```
DEFINITION
(MACHINE1 STATE N)
=
(IF (ZEROP N)
    STATE
    (MACHINE1 (STEP STATE) (SUB1 N)))
```

If we wish to emphasize that the machine state contains a stored program, we define the step function to be a fetch and an execute operation. **FETCH** takes a machine state as an argument and returns an instruction, possibly a machine word. **EXECUTE** takes an instruction and a machine state and returns the machine state which results from executing the instruction.

```
DEFINITION
(STEP STATE)
=
(EXECUTE (FETCH STATE) STATE)
```

In a more typical situation, an oracle is a list which represents a finite time-sequenced series of external events impinging on a machine. The length of the oracle determines the time span over which the machine operates. An element of the oracle is either a single external event, or a symbol such as **'TICK** indicating no event. The interpreter consumes the next element of the oracle at each step, and runs until the oracle is exhausted. The definition of **MACHINE2** gives the form of such an interpreter. In this example, the function **CONSUME-INPUT** consumes the next element of the oracle, incorporating it into the state of the machine so that the input is visible to **STEP**. (The function **CAR** returns the first element of a list, and **CDR** returns everything but the first element of a list.)

```
DEFINITION
(MACHINE2 STATE ORACLE)
=
(IF (NOT (LISTP ORACLE))
    STATE
    (MACHINE2 (STEP (CONSUME-INPUT STATE (CAR ORACLE)))
              (CDR ORACLE)))
```

The step function in this scenario can be defined to model the interrupt structure of a machine. The function **EXTERNAL-EVENTP** recognizes a condition which must be responded to, e.g., a raised interrupt bit. For a von Neumann machine, **RESPOND-TO-EVENT** is usually a simple interrupt transition which does a partial CPU context switch.

```
DEFINITION
(STEP STATE)
=
(IF (EXTERNAL-EVENTP STATE)
    (RESPOND-TO-EVENT STATE)
    (EXECUTE (FETCH STATE) STATE))
```

2.2 Interpreter Equivalence Theorems

We wish to define an *implements* relation on two machines. Let Int_A: $S_A \times O_A \rightarrow S_A$ and Int_C: $S_C \times O_C \rightarrow S_C$ be interpreter functions which define two machines M_A and M_C. (The subscripts A and C are chosen to suggest *abstract* and *concrete* machines.) Let *MapUp*: $S_C \rightarrow S_A$ be an abstraction function which maps a concrete state to an abstract state, and let *MapDown*: $S_A \rightarrow S_C$ map an abstract state to a concrete state. We say that M_C *implements* M_A if the following theorem holds.

$$
\begin{aligned}
&\forall\, s_A \in S_A, \\
(1)\qquad &\forall\, o_A \in O_A, \\
&\exists\, o_C \in O_C \textit{ such that} \\
&\\
&\textit{MapUp}\,(Int_C\,(\textit{MapDown}\,(s_A), o_C)) = Int_A\,(s_A, o_A).
\end{aligned}
$$

Figure 1 depicts this relation. Notice that if $\forall\, s_A \in S_A$, $MapUp(MapDown(s_A)) = s_A$, then it is sufficient to prove

$$
\begin{array}{ll}
 & \forall\, s_C \in S_C, \\
(2) & \quad \forall\, o_A \in O_A, \\
 & \exists\, o_C \in O_C \textit{ such that} \\
 & \\
 & \textit{MapUp}\,(\textit{Int}_C\,(s_C, o_C)) = \textit{Int}_A\,(\textit{MapUp}\,(s_C), o_A).
\end{array}
$$

To see this, substitute *MapDown* (s_A) for s_C in (2). Formula (2) gives the form of the correctness theorem we prove for Kit. Figure 2 illustrates the correspondence which the theorem establishes.

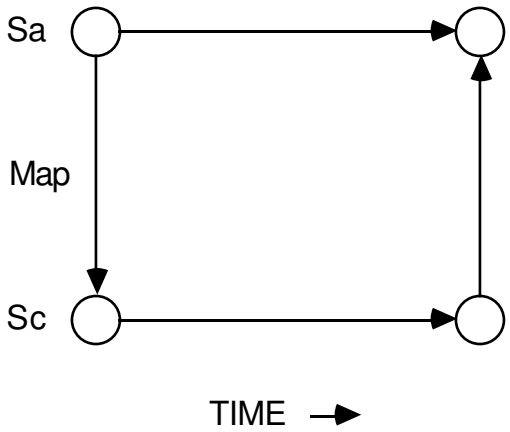

Figure 1: Interpreter Equivalence (version 1)

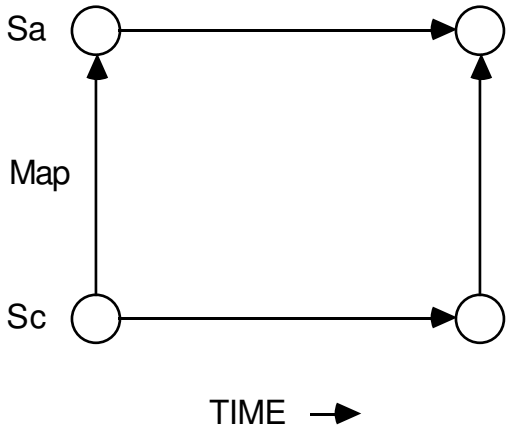

Figure 2: Interpreter Equivalence (version 2)

We cannot state (1) or (2) in the quantifier-free Boyer-Moore logic. For (2) we replace the existential variable o_C with a function **CORACLE** which computes an oracle which is sufficient to allow $\textit{Int}_C$ to match the behavior of $\textit{Int}_A$. Typically, this is a function both of the initial concrete state and the value of o_A. We re-state (2) in the Boyer-Moore logic as follows. The predicate **GOOD-CSTATE** identifies an element of the set of concrete machine states.

```
THEOREM IMPLEMENTS-RELATION
(IMPLIES (GOOD-CSTATE CSTATE)
         (EQUAL (MAPUP (INT-C CSTATE
                              (CORACLE CSTATE ORACLE)))
                (INT-A (MAPUP CSTATE) ORACLE)))
```

2.3 The Correct Implementation of Kit's Specification

The theorem **CORRECTNESS-OF-OPERATING-SYSTEM** establishes that Kit running on the target machine **TM** implements an operational specification of the kernel called the *abstract kernel*. In this theorem, the functions **AK-PROCESSOR** and **TM-PROCESSOR** are interpreter functions. The function **MAPUP-OS** is a mapping function which maps the state of the operating system as implemented on our von Neumann machine up to a corresponding abstract kernel state. The predicate **(PLISTP ORACLE)** just says that the oracle is a well-formed list, and is merely a technical requirement. We discuss this theorem in more detail in Section 8.

```
THEOREM CORRECTNESS-OF-OPERATING-SYSTEM
(IMPLIES
      (AND (GOOD-OS OS) (PLISTP ORACLE))
      (EQUAL (MAPUP-OS (TM-PROCESSOR OS
                                     (OS-ORACLE OS ORACLE)))
             (AK-PROCESSOR (MAPUP-OS OS) ORACLE)))
```

3. A Communicating Process

Figure 3 depicts a network of communicating processes. Single-headed arrows indicate message communication in the direction of the arrowhead. Double-headed arrows abbreviate two single-headed arrows, one going in each direction. Each node of Figure 3 represents a process.

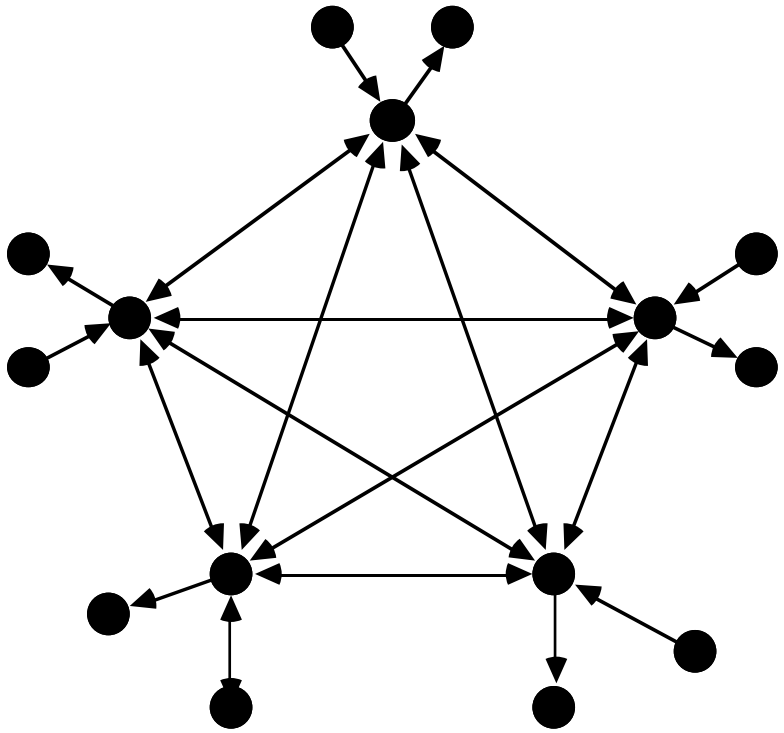

Figure 3: Network

We take this picture as the starting point for the specification and implementation of Kit. Embedded in this network is a star with five points. (The number *five* is arbitrary.) Each node at the point of the star is

intended to be implemented as a Kit process, hereafter called a *task*. Each node at the extreme perimeter, which communicates with a task in one direction only, is intended to be implemented as an input or output device.

The formal definition of a task gives a single task's view of this network. The state of a task consists of two parts: a private state which is accessible only to the owning task, and a shared state which is used for implementing inter-task communication. We distinguish two categories of transitions on a task: private transitions update only the private state, communication transitions update the shared state.

The precise nature of the private state of a task is unimportant to the formal definition of a task. Our goal, though, is to prove that a particular machine code operating system kernel implements multiple tasks. It is convenient to identify the private state of a task with the task-visible state of the implementation machine. The private transitions can then be exactly the machine instructions available to a task on the target machine. We have done this in the Kit project, defining the private state of a task to be a target machine "address space", and making the set of private transitions exactly the non-privileged instruction set of the target machine. We could substitute any other implementation machine which satisfies the requirement that execution of an instruction in behalf of the current task modifies only the private state of that task. We have performed this proof for our chosen target machine.

The shared state of a task is a set of buffers for message communication. Each buffer is a FIFO message queue of bounded length. The communication transitions are *send* and *receive*, for message passing among tasks, *input*, for receiving a message from an input device, and *output*, for sending a message to an output device. Each of these is a blocking transition. A task blocks on a *send* or *output* if the designated buffer is full. A task blocks on a *receive* or an *input* if the designated buffer is empty. Tasks are formally distinguished from I/O devices only by the the name of the operation used to perform a communication. There is no notion of a task being interrupted.

These ideas are formalized in the Boyer-Moore logic by a "good state" predicate **GOOD-TASK** which recognizes a proper task state, and an interpreter function **TASK-PROCESSOR** which defines the transitions on a state.

The function **GOOD-TASK** is a predicate on a state which requires that the private state component be "good" target machine address space. For the case of our particular target machine (see Section 6), this means that the private state consists of a certain set of CPU registers, and a segment of memory. **GOOD-TASK** requires that the shared state be a set of message buffers. Each buffer is represented as a list of bounded length.

The interpreter function which defines the transitions on a task is called **TASK-PROCESSOR** (see below). The first formal argument, **TASK**, is a task state. The argument **I** is the identifier of the task in the network which the task can sense only through its shared state. The task identifier is a non-negative integer in some bounded range. The argument **ORACLE** is a list each of whose elements is either **T**, indicating that the task is active and should take a step on its own initiative, or not **T**, indicating that the task is not active at this step. In the latter case, the oracle supplies the value of the shared state at the end of the current step. The oracle thus determines both the speed at which at task advances in "real time", and supplies changes to the shared state which are non-deterministic with respect to a task.

```
DEFINITION
(TASK-PROCESSOR TASK I ORACLE)
=
(IF (LISTP ORACLE)
    (IF (TASK-ACTIVEP (CAR ORACLE))
        (TASK-PROCESSOR (TASK-STEP TASK I)
                        I
                        (CDR ORACLE))
        (TASK-PROCESSOR (TASK-UPDATE-CHANNELS TASK
                                              (CAR ORACLE))
                        I
                        (CDR ORACLE)))
    TASK)
```

The function **TASK-UPDATE-CHANNELS**, which updates a task shared state on a non-active step, preserves the private state of the task. Therefore a task's private state is not altered when the task is not active. (The function **TASK** constructs a task state from two arguments - a private state and a shared state. **TASK-PSTATE** accesses the private state of a task.)

```
DEFINITION
(TASK-UPDATE-CHANNELS TASK NEW-SHARED-STATE)
=
(TASK (TASK-PSTATE TASK)
      NEW-SHARED-STATE)
```

An active task step is defined by the function **TASK-STEP**. The predicate **TASK-COMMUNICATIONP** determines if the current transition is a communication transition. If so, the task executes a communication step, otherwise a private step. A private step is defined to be a target machine fetch-execute operation. It does not change the shared state of a task. The function **TASK-COMMUNICATION-STEP** defines each of the four communication transitions *send*, *receive*, *input* and *output*.

```
DEFINITION
(TASK-STEP TASK I)
=
(IF (TASK-COMMUNICATIONP TASK)
    (TASK-COMMUNICATION-STEP TASK I)
    (TASK-PRIVATE-STEP TASK))
```

We present the definition of the *send* operation **TASK-EXECUTE-SEND**. This function takes four arguments: a message, the identifier of the sending task, the identifier of the destination task and the sending task's state. It returns the sending task's updated state. If the destination message buffer is full, then the returned state is identical to the initial state. Otherwise, the task state is updated as follows. The private state is modified in an implementation-dependent way to update control past the communication operation (the function **TASK-UPDATE-CONTROL**). The shared state is updated by queueing the message on the destination message buffer.

```
DEFINITION
(TASK-EXECUTE-SEND MSG SRCID
                   DESTID TASK)
=
(IF (QFULLP SRCID DESTID
            (TASK-MBUFFERS TASK)
            (TASK-MBUFFER-CAPACITY))
    TASK
    (TASK (TASK-UPDATE-CONTROL (TASK-PSTATE TASK))
          (LIST (TASK-IBUFFERS TASK)
                (TASK-OBUFFERS TASK)
                (ENQ MSG SRCID DESTID
                     (TASK-MBUFFERS TASK)))))
```

The functions **GOOD-TASK** and **TASK-PROCESSOR** give our highest level specification for Kit. Kit is required to implement a fixed number of these tasks on a single von Neumann machine. The property that a task's private state is not modified when it is not active is a simple consequence of the definition of a task. The definition also has the property that no private transition updates shared state. Any implementation of tasks must preserve these properties. An implementation must also carry out the communication transitions according to specification. These two points - the protection of a task's private state, and communication only as specified - encompass our notion of process isolation.

4. The Specification for Kit

The task-level specification for Kit defines the communication transitions in which a task may engage, but says nothing about how tasks are scheduled. A lower-level specification for Kit, called the *abstract kernel*, defines a scheduling algorithm for a fixed number of tasks, implements the communication primitives (including the delay of tasks which block on a communication), and handles communication with asynchronous devices. The distinction between a task and an I/O device is made more visible: each task has a state known completely to the abstract kernel, while the state of an I/O device is unspecified. Devices communicate with the kernel only through shared ports.

The abstract kernel is defined by two functions **GOOD-AK** which recognizes a "good" abstract kernel state, and the interpreter function **AK-PROCESSOR** which gives the set of transitions on a state. **GOOD-AK** is an invariant of **AK-PROCESSOR**. That is, the following theorem holds.

```
THEOREM GOOD-AK-AK-PROCESSOR
(IMPLIES (GOOD-AK AK)
         (GOOD-AK (AK-PROCESSOR AK ORACLE)))
```

The predicate **GOOD-AK** recognizes a structure consisting of the following fields.

- **AK-PSTATES** is a fixed-length list containing the private state of each task. Each private state is disjoint from the others.
- **AK-IBUFFERS**, **AK-OBUFFERS** and **AK-MBUFFERS** are the input, output and message buffers. These are identical to the shared state at the task level.
- **AK-READYQ** is a queue of task identifiers. When non-empty, the first element of the queue identifies the current task.
- **AK-STATUS** is an array, one element for each task, which gives the current status of the task. The status of a task is one of: {*ready*, *error*, *waiting-to-send*, *waiting-to-receive*, *waiting-to-input*, *waiting-to-output*}. The ready queue is a permutation of the set of ready tasks as defined by **AK-STATUS**.

- **AK-RWSTATE** is a running/wait state flag. This flag is set if and only if the ready queue is empty.
- **AK-CLOCK** is the program timer used to control allocation of CPU time to a task. The clock counts down, and when it reaches zero causes a call to the scheduler.
- **AK-IPORTS**, **AK-OPORTS** is an array of input and output ports, respectively, for communication with devices. This is the physical interface to asynchronous devices.

The interpreter function which defines the transitions on the abstract kernel is **AK-PROCESSOR**. The argument **AK** represents the state of the abstract kernel. The argument **ORACLE** is an oracle used to represent asynchronous inputs. An oracle is a list some of whose elements are I/O interrupts. An input interrupt is a 2-tuple which gives an input character and a device id. An output interrupt merely contains a device id. On each step, the **AK-INTERPRETER** consumes the current input signal and then applies the step function **AK-STEP** to the resulting state. (**AK-POST-INTERRUPT** makes an interrupt visible within the state of the machine by updating the input or output port associated with the interrupting device.)

```
DEFINITION
(AK-PROCESSOR AK ORACLE)
=
(IF
 (LISTP ORACLE)
 (AK-PROCESSOR (AK-STEP (AK-POST-INTERRUPT (CAR ORACLE) AK))
               (CDR ORACLE))
 AK)
```

The function **AK-STEP** defines the single-step function of the abstract kernel. It defines an interrupt structure and applies an interrupt handler in response to each type of interrupt. Each interrupt handler function returns an updated **AK** state. The definitions of the interrupt handlers provide a specification for the services which must be provided by the implementation of Kit on the target machine.

```
DEFINITION
(AK-STEP AK)
=
(COND
 ((AK-INPUT-INTERRUPTP AK)
  (AK-INPUT-INTERRUPT-HANDLER
                (AK-INTERRUPTING-INPUT-PORT (AK-IPORTS AK))
                AK))
 ((AK-OUTPUT-INTERRUPTP AK)
  (AK-OUTPUT-INTERRUPT-HANDLER
               (AK-INTERRUPTING-OUTPUT-PORT (AK-OPORTS AK))
               AK))
 ((AK-WAITING AK) AK)
 ((AK-ERRORP AK) (AK-ERROR-HANDLER AK))
 ((AK-CLOCK-INTERRUPTP AK)
  (AK-CLOCK-INTERRUPT-HANDLER AK))
 ((AK-SVC-INTERRUPTP AK)
  (AK-SVC-HANDLER AK))
 (T (AK-PRIVATE-STEP AK)))
```

Input and output interrupt processing has the highest priority. The functions **AK-INPUT-INTERRUPT-HANDLER** and **AK-OUTPUT-INTERRUPT-HANDLER** define the input and output interrupt handlers. **AK-WAITING** determines if the machine is in the wait state. If so, no state change occurs. If none of the above conditions hold, the error status of the current task is checked. The function **AK-ERROR-HANDLER** defines the kernel's error handler. A clock interrupt signals the end of

the current task's time slice. The function **AK-CLOCK-INTERRUPT-HANDLER** defines the task switch on a clock interrupt. The function **AK-SVC-INTERRUPTP** detects a request to call a kernel function in behalf of the current task ("svc" abbreviates "supervisor call"). The services provided by the kernel are exactly the communication primitives of the task layer: *send*, *receive*, *input* and *output*. The function **AK-SVC-HANDLER** defines these operations at the abstract kernel layer. Finally, if none of the above conditions hold, the current task takes a private step as defined by **AK-PRIVATE-STEP**.

Recall that a task's private state is a target machine address space, and the transitions on a private state are the machine instructions available to a task. **AK-PRIVATE-STEP** applies the function which defines the target machine's fetch-execute algorithm, **TM-FETCH-EXECUTE**, to the current task's private state. More precisely, the *i*th element of the private state array is replaced by the application of **TM-FETCH-EXECUTE** to that element, where *i* is the identifier of the current task. The isolation of private states is a simple result of the properties of the list access functions **GETNTH** and **PUTNTH**. **GETNTH** accesses the *n*th element of a list. **PUTNTH** stores a value in the *n*th location in a list.

```
DEFINITION
(AK-PRIVATE-STEP AK)
=
(AK (AK-FETCH-EXECUTE (AK-TASKID AK)
                      (AK-PSTATES AK))
    (AK-IBUFFERS AK)
    (AK-OBUFFERS AK)
    (AK-MBUFFERS AK)
    (AK-READYQ AK)
    (AK-STATUS AK)
    (AK-RWSTATE AK)
    (SUB1 (AK-CLOCK AK))
    (AK-IPORTS AK)
    (AK-OPORTS AK))

DEFINITION
(AK-FETCH-EXECUTE ID PSTATES)
=
(PUTNTH (TM-FETCH-EXECUTE (GETNTH ID PSTATES))
        ID PSTATES)
```

An **AK** step is an application of one of five interrupt functions, or is a private step, or is a no-op in the case of a waiting machine with no I/O interrupts. The definitions of the five **AK** interrupt handlers provide a specification for the services provided by the implementation of Kit on the target machine. The definition of a private step establishes a constraint on the protection mechanism provided by the target machine's architecture. We examine the definition of clock interrupt handler.

AK-CLOCK-INTERRUPT-HANDLER defines a simple round-robin scheduling algorithm. The identifier of the current task is the first element of the ready queue. On a clock interrupt, the first element of the ready queue is removed and enqueued at the end of the ready queue. The dispatcher senses an empty ready queue and sets the kernel state accordingly: the kernel is put in the wait state if the ready queue is empty, otherwise the kernel is put in the run state and the program clock is initialized. On a clock interrupt the length of the ready queue is not changed, so the former condition does not hold. The same primitives which manipulate buffers also manipulate the ready queue. All are finite queues represented as list structures.

```
DEFINITION
(AK-CLOCK-INTERRUPT-HANDLER AK)
=
(AK-DISPATCHER (AK (AK-PSTATES AK)
                   (AK-IBUFFERS AK)
                   (AK-OBUFFERS AK)
                   (AK-MBUFFERS AK)
                   (ENQ (AK-TASKID AK)
                        (DEQ (AK-READYQ AK)))
                   (AK-STATUS AK)
                   (AK-RWSTATE AK)
                   (AK-CLOCK AK)
                   (AK-IPORTS AK)
                   (AK-OPORTS AK)))

DEFINITION
(AK-DISPATCHER AK)
=
(AK (AK-PSTATES AK)
    (AK-IBUFFERS AK)
    (AK-OBUFFERS AK)
    (AK-MBUFFERS AK)
    (AK-READYQ AK)
    (AK-STATUS AK)
    (IF (QEMPTYP (AK-READYQ AK))
        (AK-WAIT-STATE)
        (AK-RUN-STATE))
    (IF (QEMPTYP (AK-READYQ AK))
        (AK-CLOCK AK)
        (AK-TIME-SLICE))
    (AK-IPORTS AK)
    (AK-OPORTS AK))
```

The remaining interrupt handlers are defined in the same way as the clock interrupt handler. Each is a Boyer-Moore function which constructs a new **AK** state in response to a particular interrupt. The result is an abstract operational specification for a kernel implementation. **AK** is abstract in the following ways.

- The private state spaces of tasks are transparently isolated. This provides an important constraint on the implementation.
- The data structures used to manage tasks are represented as high-level list structures.
- The transitions on the kernel state are specified functionally. All kernel operations take place in a single abstract step.

Several features of the abstract kernel are defined to coincide with the target machine. These features could be replaced with details of a different implementation machine.

- The private state of a task is target machine address space, and a transition on a private state is a target machine instruction.
- The I/O interface is identical to that of the target machine. That is, the representation of asynchronous inputs is chosen to match the implementation.
- The interrupt structure of the abstract kernel (as defined by the case structure of **AK-STEP**) is defined to match the interrupt structure of the target machine.

5. The Abstract Kernel Implements Processes

The relationship between the abstract kernel and an individual task is pictured in Figure 4, and is formalized by the theorem **AK-IMPLEMENTS-PARALLEL-TASKS**. Intuitively, this theorem says that for a given good abstract kernel state **AK** and abstract kernel oracle **ORACLE**, the final state reached by task **I** can equivalently be achieved by running **TASK-PROCESSOR** on the initial task state, with an oracle constructed by the function **CONTROL-ORACLE**. The oracle constructed for **TASK-PROCESSOR** accounts for the precise sequence of delays to task **I** in the abstract kernel.

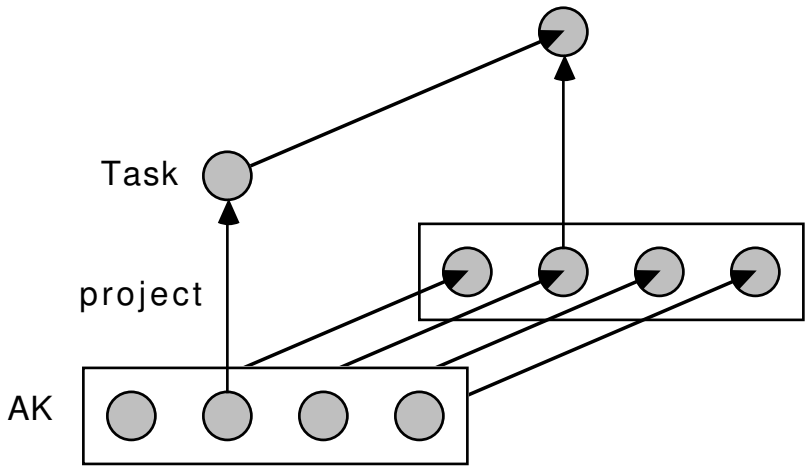

Figure 4: AK Implements Parallel Tasks

```
THEOREM AK-IMPLEMENTS-PARALLEL-TASKS
(IMPLIES
      (AND (GOOD-AK AK)
           (FINITE-NUMBERP I
                           (LENGTH (AK-PSTATES AK))))
      (EQUAL (PROJECT I (AK-PROCESSOR AK ORACLE))
             (TASK-PROCESSOR (PROJECT I AK)
                             I
                             (CONTROL-ORACLE I AK ORACLE))))
```

6. The Target Machine

The target machine **TM** is a simple von Neumann computer. It is not based on an existing physical machine because we are not interested in the task of formalizing an existing machine. Post hoc formalization tends to be difficult, if not impossible. We intend for **TM** to be straightforward.

TM is a 16-bit machine. Main memory consists of 2^{16} 16-bit words. The processor state contains 8 general purpose registers, one of which is the program counter and another a stack pointer. There are four flag fields: a 2-bit condition code, a 6-bit error code, a supervisor call flag, and a 7-bit supervisor call identifier. **TM** has simple architectural support for multi-programming. This support consists of a base/limit register pair mechanism for memory protection, and a supervisor/user mode flag for protecting privileged operations. Processor registers which are accessible only in the supervisor mode are the base/limit register pair, a supervisor address limit register, the supervisor/user mode flag, a running/wait

state flag and the program clock. **TM** is capable of asynchronous character I/O. It communicates with 16 input devices and 16 output devices by an array of input ports and an array of output ports. Figure 5 gives a summary of the **TM** architecture in PMS notation [Bell 71].

```
Memory state

  Mp[0:65535]<0:15>   main memory of 2^16 16-bit words

Pc state

  R[0:7]<0:15>        8 general purpose registers;
                      R[0] is the PC; R[1] is the SP

  CC<0:1>             2-bit condition code
  ERROR<0:5>          6-bit error code
  SVCFLAG             1-bit svc call flag
  SVCID<0:6>          7-bit svc identifier

  BASE<0:15>          16-bit address base register
  LIMIT<0:15>         16-bit address limit register
  SLIMIT<0:15>        16-bit address defining the upper limit
                      of the supervisor based at address 0 in
                      memory

  SVMODE              supervisor/user mode flag
  RWSTATE             running/wait state flag
  CLOCK<0:15>         program clock used for time slicing

I/O interface

  IPORTS[0:15](<0:1>;<0:1>;<0:7>) an array of 16 input ports;
                                  each port is a 3-tuple
                                  (interrupt-flag, overrun-flag, char-buffer)

  OPORTS[0:15](<0:1>;<0:1>;<0:7>) an array of 16 output ports;
                                  each port is a 3-tuple
                                  (interrupt-flag, busy-flag, char-buffer)
```

Figure 5: PMS Description of TM

The target machine is formally defined by two functions - the "good state" predicate **GOOD-TM** and the interpreter **TM-PROCESSOR**.

The predicate **GOOD-TM** formalizes the contents of Table 5. It recognizes a data structure with the following fourteen fields: **TM-MEMORY**, **TM-REGS**, **TM-CC**, **TM-ERROR**, **TM-SVCFLAG**, **TM-SVCID**, **TM-BASE**, **TM-LIMIT**, **TM-SLIMIT**, **TM-SVMODE**, **TM-RWSTATE**, **TM-CLOCK**, **TM-IPORTS** and **TM-OPORTS**. These are the names of the accessor functions to a **TM** state. **GOOD-TM** imposes size restrictions on each field as indicated by Table 5.

The function **TM-PROCESSOR** is the interpreter function which defines the transitions on a **TM** state. The formal argument **TM** represents a machine state, and the formal argument **ORACLE** represents an oracle. An oracle is a list, some of whose elements are I/O interrupts. An input interrupt is a 2-tuple which gives an input character and a device id. An output interrupt merely contains a device id.

```
DEFINITION
(TM-PROCESSOR TM ORACLE)
=
(IF
 (LISTP ORACLE)
 (TM-PROCESSOR (TM-STEP (TM-POST-INTERRUPT (CAR ORACLE) TM))
               (CDR ORACLE))
 TM)
```

TM-POST-INTERRUPT incorporates interrupts into the state of the machine so that they can be sensed. An input interrupt for device *i* is posted by changing the value of the *i*th input port as follows: the interrupt flag is raised, the error flag gets the previous value of the interrupt flag to signal an overrun condition, and the input character is placed in the character buffer. An output interrupt for device *i* is posted by changing the value of the *i*th output port as follows: the interrupt flag is raised, the busy flag is cleared, and the character buffer is cleared (although this action is superfluous). When the current oracle element is not an I/O interrupt, **TM-POST-INTERRUPT** makes no change to the state of the machine.

The function **TM-STEP** defines the single step function for the **TM** interpreter. It gives the interrupt structure of the target machine. Each of the interrupt branches of **TM-STEP** (an input interrupt, an output interrupt, a program error, a clock interrupt and a supervisor call interrupt) does a *PSW swap*, which partially saves the state of the CPU in a fixed location of memory and loads a new program counter giving the address of an operating system interrupt handling routine. When no I/O interrupt occurs and **TM** is in the wait state, **TM-STEP** returns the current machine state unchanged. The function **TM-FETCH-EXECUTE** defines the instruction fetch-execute cycle of the target machine.

```
DEFINITION
(TM-STEP TM)
=
(COND ((TM-INPUT-INTERRUPTP TM)
       (TM-EXECUTE-INPUT-INTERRUPT TM))
      ((TM-OUTPUT-INTERRUPTP TM)
       (TM-EXECUTE-OUTPUT-INTERRUPT TM))
      ((TM-WAITING TM) TM)
      ((TM-ERRORP TM)
       (TM-EXECUTE-ERROR-INTERRUPT TM))
      ((TM-CLOCK-INTERRUPTP TM)
       (TM-EXECUTE-CLOCK-INTERRUPT TM))
      ((TM-SVC-INTERRUPTP TM)
       (TM-EXECUTE-SVC-INTERRUPT TM))
      (T (TM-FETCH-EXECUTE TM)))
```

We examine interrupts and the fetch-execute cycle more closely. Figure 6 describes what happens on a clock interrupt: the current program counter, stack pointer and flags fields are stored in memory locations [0:2]. A new program counter is loaded from a fixed location in memory giving the address of the clock interrupt handler, the stack pointer is loaded with the supervisor limit address (a stack occupies the high address end of a memory segment), and the machine is put in supervisor mode. All of the other interrupt transitions referenced in **TM-STEP** are defined in a similar fashion.

The function **TM-FETCH-EXECUTE** defines an algorithm for fetching instructions from memory and decoding opcodes. For each opcode, **TM-FETCH-EXECUTE** applies a function which defines a transformation on the state of the machine. Figure 7 documents **TM**'s small instruction set. The purpose of the figure is to suggest the extent of the instruction set. We have defined only those instructions required to program the operating system. Other instructions can be added at the cost of proving that each

```
mem[0:2]   <-  [pc,sp,flags]
pc         <-  mem[3]
sp         <-  slimit - 1
svmode     <-  supervisor-mode
```

Figure 6: The TM Clock Interrupt

one satisfies the **GOOD-TM** invariant. **TM** has instructions of zero, one and two arguments. The parameters which occur in Figure 7 should be interpreted as real addresses: one of memory address, register address or immediate operand. In the case of binary operations, a result is stored at the location indicated by the first argument. The condition code is a 2-bit value which indicates two ALU conditions: zero/non-zero and carry/no-carry.

```
Non-Privileged Operations

  ADD a b      add, set the condition code
  BR a         set the pc unconditionally
  BRZ a        set the pc if cc = <zero,no-carry>
  BRNZ a       set the pc if cc not = <zero,no-carry>
  CALL a       save the pc on the stack, load a new pc
  COMPARE a b  set the condition code based on numerically
               comparing a and b
  DECR a       decrement, set the condition code
  DECR-MOD a b decrement a modulo b, set the condition code
  INCR a       increment, set the condition code
  INCR-MOD a b increment a modulo b, set the condition code
  MOD a b      a mod b, set the condition code
  MOVE a b     move b to location indicated by a
  MULT a b     multiply, set the condition code
  RETURN       set the pc to the top element of the stack
  SVC addr     raise the svcflag, set the svcid

Privileged Operations

  LBASE a      load the base register
  LLIMIT a     load the limit register
  LPSW a       load the pc, sp and flags; put the machine in user mode
  POST a       raise the output interrupt flag in the output
               port given by the argument
  RUN          put the machine in the run state
  TIME a       set the clock
  STOUT a b    start output on the device indicated by a;
               the output character is given by b
  SVCR a       load the pc, sp and flags; put the machine
               in user mode; clear the svcflag
  TESTI a      test the indicated input port for an overrun error
  TESTO a      test the indicated output port for busy
  WAIT         put the machine in the wait state
```

Figure 7: TM's Instruction Set

TM's ALU performs the following operations: *plus*, *difference*, *times*, *remainder*, *increment*, *decrement*, *increment-mod* and *decrement-mod*. *Increment-mod* takes two arguments and increments its first argument modulo its second argument. *Decrement-mod* decrements its first argument modulo its second argument. Besides returning an integer value, each ALU operation also sets a carry bit. Remainder is a powerful operation. The kernel in fact uses this operation only to take the remainder of a number by some power of two. Therefore the remainder operation in the ALU could be replaced by a simpler shift operation to satisfy the needs of the kernel.

This completes our summary of **TM**. It is a very simple von Neumann machine. It provides a minimal amount of protection hardware: supervisor/user mode for protecting privileged operations, and a base/limit register pair for protecting memory access. The operating system kernel is responsible for implementing isolated tasks using this hardware.

7. The Machine Code Implementation of Kit

In this section we describe the implementation of the kernel on the target machine. The kernel is written in an assembler language for **TM**. The assembler is written in the Boyer-Moore logic, but the only role it plays in the proof is to generate the large constant which is the Kit machine code.

The source code contains about 620 lines including assembler directives, data declarations and comments. There are about 300 lines of actual assembler language instructions which, when assembled, expands to 750 machine words. The size of the entire kernel including data structures occupies about 3K words of memory.

To make more vivid the claim that we verify machine code, we exhibit a small portion of the assembler language listing of the kernel in Figure 8. This is the code which saves the CPU state of the current task on an interrupt. The code places the address of an entry in a task table into the target machine's register 2, and saves the task-visible state of the CPU of our target machine in that entry. The verification of Kit proves that the processor state is saved correctly so that the fiction that each process owns the processor is maintained. Each operation contains 0, 1 or 2 operands. An operand contains an optional address mode, an address, and an optional displacement. The address modes are: 0 - immediate (the default if omitted), 1 - register address, 2 - memory address, 3 - register indirect address.

```
SAVE-STATE
move  2,temp-r2  1,r2             ;; Save R2 in memory
move  2,temp-r3  1,r3             ;; Save R3 in memory
move  1,r3       readyq           ;; Point R3 to ready queue
call  qfirst                      ;; Put current taskid in R2
SAVE-STATE-RETURN
mult  1,r2  task-table-entry-length ;; Multiply by task table entry length
add   1,r2  task-table              ;; R2 points to current task table entry
;; Now save CPU registers and flags in consecutive words.
move  3,r2,pc-field   2,reg-save-area,interrupt-pc-field
move  3,r2,sp-field   2,reg-save-area,interrupt-sp-field
move  3,r2,r2-field   2,temp-r2
move  3,r2,r3-field   2,temp-r3
move  3,r2,r4-field   1,r4
move  3,r2,r5-field   1,r5
move  3,r2,r6-field   1,r6
move  3,r2,r7-field   1,r7
add   1,r2,flag-field             ;; Bump index register
move  3,r2            2,reg-save-area,interrupt-flag-field
move  1,r2            2,temp-r2   ;; Restore R2 & R3,
move  1,r3            2,temp-r3   ;; This is necessary for SVC interrupts.
return
```

Figure 8: Assembler Language Listing of State Saving Code

Formally, the target machine loaded with the Kit machine code is another example of a finite state machine. Its "good state" predicate is **GOOD-OS**, and its interpreter function is **TM**'s interpreter, **TM-PROCESSOR**.

GOOD-OS defines the state of the target machine when loaded with the kernel, and when running a user

process. That is, a target machine state may violate **GOOD-OS** while a Kit routine is executing, but each Kit routine is proved to return the machine to a **GOOD-OS** state upon exiting. A user process always executes in a **GOOD-OS** state.

GOOD-OS says formally how **TM**'s memory is laid out. It states that the kernel occupies a memory segment beginning at location 0. Within that segment, **GOOD-OS** gives invariants on many of the kernel data structures. Here are some of its provisions.

- Each element of the interrupt vector is required to contain the address of the appropriate interrupt handling code. These addresses never change.
- A particular segment of the kernel is required to contain exactly the machine code which results from assembling the kernel. This code is never modified.
- The segment table contains a base/limit register pair for each task, defining the location and length of each task's memory segment. It is constrained so that each base/limit pair is mutually disjoint, and is disjoint from the memory occupied by the kernel. Kit maintains this invariant, and therefore guarantees the **TM**'s address protection mechanism is sufficient to isolate process memory segments, and isolate the kernel from processes.
- When the kernel exits, it leaves the machine in user mode. A user process cannot get into supervisor mode.

8. The Machine Code Correctly Implements its Specification

The correctness of the machine code implementation of Kit is stated by the theorem **CORRECTNESS-OF-OPERATING-SYSTEM**, previously mentioned in section 2.3. It is an interpreter equivalence theorem which establishes the correspondence between the Kit code running on **TM** and the abstract kernel.

```
THEOREM CORRECTNESS-OF-OPERATING-SYSTEM
(IMPLIES
      (AND (GOOD-OS OS) (PLISTP ORACLE))
      (EQUAL (MAPUP-OS (TM-PROCESSOR OS
                                     (OS-ORACLE OS ORACLE)))
             (AK-PROCESSOR (MAPUP-OS OS) ORACLE)))
```

The correspondence between the two machines (the abstract kernel and the target machine loaded with the kernel) is defined by the function **MAPUP-OS**. **MAPUP-OS** serves two roles. First, it maps up the machine implementation of kernel data structures to their abstract representation. This is an application of Hoare's method for proving the correctness of data representations [Hoare 72]. Second, **MAPUP-OS** maps the current state of each process at the low level to an array of completely isolated private states at the abstract level.

One thing must be understood about this theorem. The function **OS-ORACLE** computes the time required by Kit to simulate the behavior of the kernel. The abstract kernel can handle an interrupt, that is, put an input character in a buffer and wake up a process waiting for input, in a single abstract step. It takes the target machine many steps to run a program which does the same thing. The correctness theorem says that the target machine simulates the behavior of the abstract kernel when given the right oracle. This theorem can be informally read as "Kit simulates the abstract kernel when I/O interrupts don't arrive too quickly." The basic requirement is that I/O interrupts arrive with a minimum time interval equal to the maximum length of time Kit requires to execute any of its services. This condition is met by the oracle constructed above, and it can be shown that many other oracles suffice. In particular, if this property could be proved about **ORACLE**, then **ORACLE** would suffice at the target machine level, eliminating the need for the function **OS-ORACLE**.

At the highest level the proof is a simple induction on the oracle. The proof requires the following two results.

First, each kernel routine implements its specification at the abstract kernel level. The proof of kernel routines is accomplished by symbolically interpreting paths through the machine code. The final state reached by each path is proved to correspond to an operation at the abstract kernel level.

Second, a target machine fetch-execute operation in behalf of a user process implements an abstract private step. Recall that an abstract private step is an application of the target machine's fetch-execute algorithm to one of an array of private task states (see the definition of **AK-PRIVATE-STEP** in Section 4). The requirement on the fetch-execute algorithm is summarized by two lemmas. One is a correctness lemma which says that **TM-FETCH-EXECUTE** behaves on the current process exactly as if the process state were a completely isolated machine. The second is a protection lemma which says that **TM-FETCH-EXECUTE** modifies no process but the current process.

The two theorems, **AK-IMPLEMENTS-PARALLEL-TASKS**, Section 5, and **CORRECTNESS-OF-OPERATING-SYSTEM**, compose to give a simple proof that the **I**th process projects out of a **GOOD-OS** state in a way which corresponds to **TASK-PROCESSOR**. This is the final result in the Kit proof script. Figure 9 gives a picture of this theorem.

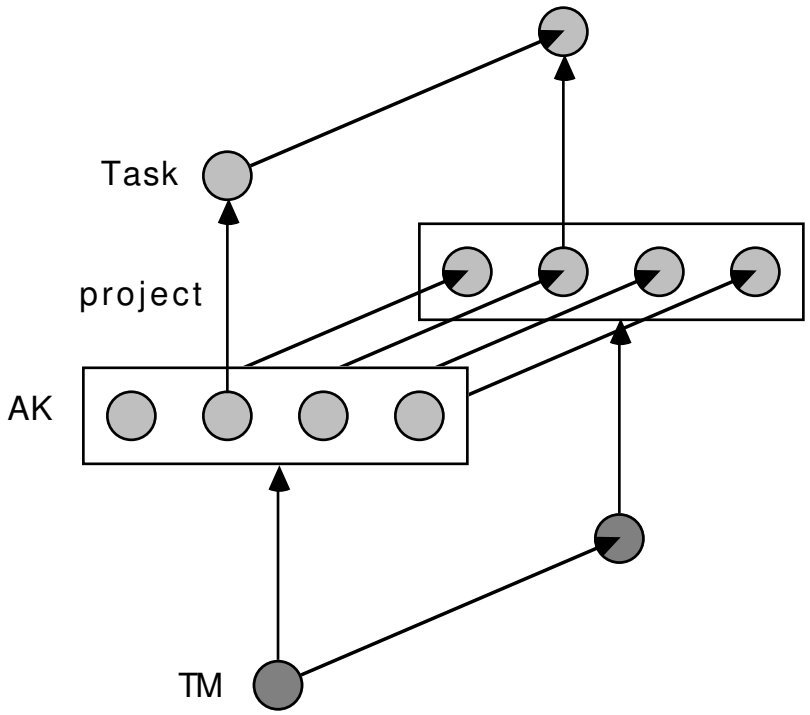

Figure 9: Final Theorem

```
THEOREM OS-IMPLEMENTS-PARALLEL-TASKS
(IMPLIES
 (AND (GOOD-OS OS)
      (PLISTP ORACLE)
      (FINITE-NUMBERP I
                      (LENGTH (AK-PSTATES (MAPUP-OS OS)))))
 (EQUAL
     (PROJECT-ITH-TASK I
                       (TM-PROCESSOR OS
                                     (OS-ORACLE OS ORACLE)))
     (TASK-PROCESSOR (PROJECT-ITH-TASK I OS)
                     I
                     (CONTROL-ORACLE I
                                     (MAPUP-OS OS)
                                     ORACLE))))
```

The entire statement of the problem, given only the axioms of the Boyer-Moore logic, requires 1020 definitions and 3561 lemmas. The script of definition and lemmas includes

- facts about numbers, sets and lists,
- the definition of the target machine and a proof that **GOOD-TM** is an invariant of the target machine,
- the definition of the abstract kernel and a proof that **GOOD-AK** is an invariant of the abstract kernel,
- the definition of the kernel, and a proof that the kernel always leaves a target machine in a **GOOD-OS** state,
- the proof of correspondence of each of the kernel routines with its specification,
- the correctness and protection proof about **TM-FETCH-EXECUTE**,
- the definition of a task, and
- the proof that the abstract kernel correctly implements tasks.

We spent an inordinate amount of time defining and then fine-tuning our database of lemmas about sets, number and lists. Even in its final state, the script does not have a clean set of rules in these domains. This resulted in a number of ugly "help" lemmas throughout the script. The development of useful rewrite rules in these areas would make the Boyer-Moore theorem prover vastly easier to use.

9. Related Work

9.1 Specification and Proof Methods

Our approach to the specification and verification of Kit derives from well known earlier work. The *implements* relation established by an interpreter equivalence theorem is similar to Milner's *weak simulation* relation [Milner 71]. Hoare's approach to proving the correctness of data representations [Hoare 72], similar to Milner's work, is also a precursor. There has been more recent interest in the issues surrounding the verification of data representations, this work goes under the names *data refinement* [Hoare 87], or *data reification* [Jones 86].

Several attempts to verify operating systems cite the work of Milner, Hoare and others who have suggested similar approaches to verification. The methodology for designing operating system software proposed by Robinson and his co-workers [Robinson 77] calls for a sequence of abstract machines, each

related by an *implements* relation. Kemmerer [Kemmerer 82] acknowledges a debt to Milner and Hoare in the verification of a portion of the security kernel of UCLA Secure Unix. Rushby [Rushby 81a] described an approach to kernel verification similar to ours.

Several other research efforts have used the Boyer-Moore logic and theorem prover to specify and verify components of computing systems. Hunt [Hunt 85] proved an interpreter equivalence theorem to establish the correctness of the FM8501 microprocessor. A successor to FM8501, called FM8502, has also been verified. Moore [Moore 88] proved the correctness of the translator of a high-level assembly language called Piton. The Piton assembler is targeted to FM8502 machine code. Young [Young 88] verified the correctness of a micro-Gypsy compiler targeted to Piton. A forthcoming special issue of *The Journal of Automated Reasoning* is devoted to the description of this stack of verified systems components, including the conceptual place of Kit in this stack [Bevier, et. al. 89].

There are many formal specification languages other than the Boyer-Moore logic some of which are supported by mechanical tools. A list of approaches to specification and verification must include Affirm [Gerhart 80], Gypsy [Good, et. al. 78], HDM [Robinson & Levitt 77], HOL [HOL 87], VDM [Jones 86], and Z [Spivey 88]. The purpose of the Kit project was to specify an operating system kernel with a particular process isolation property, and mechanically check the proof of a correct implementation of that specification at the machine code level. We chose the Boyer-Moore logic for two reasons: first, our previous expertise in the logic, and second, the existence of the Boyer-Moore theorem prover. Previous work with the Boyer-Moore theorem prover [Shankar 86, Hunt 85] had demonstrated that it could be guided through very large and complex proofs. We attempted no comparison of Kit's specification in the Boyer-Moore logic with a specification in a different language.

One technical advantage in pushing operational specifications to an abstract level in the Boyer-Moore logic deserves mention. The Boyer-Moore logic's definitional principle requires a proof of the unique existence of every function defined, and therefore a proof of consistency of the specification. The theorem prover handles this proof automatically in most cases.

9.2 Operating System Verification

Two areas predominate in operating system verification: verification of parallel processes, and verification of security properties.

The area of parallelism is primarily concerned with proving safety and liveness properties of sets of processes under various models of process communication. Above the kernel level, an operating system can be viewed as a set of cooperating parallel processes. So, techniques for verifying parallel processes can be applied to operating system verification above the kernel level. Our work logically supports this work. The purpose of our work is to verify a kernel implementation. We don't reason about the correctness of a particular set of concurrent processes, but prove that any set of processes which can be implemented on Kit is implemented without errors introduced by Kit.

The seminal work in this area is the "THE"-multiprogramming system [Dijkstra 68] in which process synchronization via semaphores is implemented at the lowest layer. This work reveals to what advantage an operating system can be designed as a system of communicating sequential processes. Saxena [Saxena 76] considers low level issues of processor and memory sharing in a multiprogrammed operating system. The design of a scheduler and memory manager, synchronized via monitors, is verified. Flon [Flon 77] treats two subjects related to the correctness of operating systems. First, a methodology for the design, implementation and verification of operating systems is discussed. Second, the problem of the total correctness of parallel programs is considered. Karp [Karp 83] proposes an extension of Pascal to

include a method of process communication called a *module*. Concurrent systems expressed in this language can be demonstrated to be *failure free*, which is a notion of non-termination. The application of this communication model to operating systems is demonstrated.

Security is the other major area in operating system verification. In the early seventies the notion became current that a security policy should be implemented in the nucleus of an operating system, a *security kernel*. A number of efforts attempted to design, implement and verify a security kernel. A security policy given by Bell and LaPadula [Bell 75] was the first attempt to formalize a specification for a security kernel. Alternative formulations of security were given by Feiertag, Levitt and Robinson [Feiertag 77], and by Popek and Farber [Popek 78].

The goals of each security kernel project were similar in outline: design a security kernel, prove that the design satisfies a formally described security policy, implement the kernel, and prove the implementation correct. Some projects were intended to complete only an initial portion of this sequence of goals. The goals were met with varying degrees of success.

Many security kernel projects are reported in the literature: PSOS [Feiertag 79, Neumann 77], KSOS [McCauley 79, Berson 79], UCLA Secure Unix project [Popek 79, Walker 80], KVM/370 [Gold 79], and SCOMP [Fraim 83]. The Secure Ada Target (SAT, now called LOCK) [Boebert 85] is an ongoing project at Honeywell. Landwehr [Landwehr 83] gives a useful summary of the state of the art circa 1983. Rushby criticizes the kernel approach to system security [Rushby 81b]. We do not repeat his argument, but point out that the alternative approach to security which he proposes results in a mandate for the type of verification carried out for Kit: a proof of the isolation of processes implemented in a shared environment. Rushby calls this a *separation kernel*.

Outside of these two categories, mention should be made of the SIFT project [MelliarSmith 81], which tentatively explored some of the problems of implementing processes, but did not formally prove an *implements* relation or do code level proofs.

The relationship between our work and that previously reported in the literature can be summarized as follows. There are two main threads in operating system verification: verification of parallel processes, and verification of security. The work in parallel processes lies inherently above the level of verification reported for Kit. The work in security reaches in principle down to the implementation level of Kit, but no work has previously reached that level.

10. Remarks

The purpose of Kit is to provide verified task isolation. That is, tasks can communicate only in specified ways. As a result, a verified set of communicating processes will run as specified on Kit provided there are no hardware errors. Kit is guaranteed not to introduce implementation bugs, since all code is verified.

A number of significant results are required to establish the main theorem.

- The termination of kernel routines.
- The correctness of the address space abstraction, i.e., that an address space can be viewed as an independent machine.
- The isolation of the operating system from tasks on the target machine.
- The inability of a task to enter supervisor mode.

Therefore, the verification of Kit checks important security properties. We have proved task isolation, the protection of the operating system from tasks, and the inability of tasks to enter supervisor mode. Our small system is tamper proof. These results are fundamental to computer security, but are a by-product of the correctness proof. They have received scant attention in formal verification since the focus has been on sophisticated security models rather than correctness. The issues involved in correctly implementing multiple processes on shared resources have been largely ignored.

The proof of Kit is accomplished by establishing a machine simulation theorem which relates Kit to a definition of a process which appears to be running on its own machine. Kit is shown to implement a fixed number of conceptually distributed communicating processes. The specification machine is so abstract that the proof of its properties is quite tractable. An example of a property which is trivial to establish at this level is the protection of a process's private state. We have not stated and proved other properties, but clearly it is preferable to do so at the high end than at the low end. Because the *implements* relation is proved, properties established at the high end hold (under some state space mapping) at the low end.

The verification of Kit revealed a number of bugs. Simple bugs, like naming an incorrect register, or using the wrong address mode, were found by symbolically interpreting paths through the machine code. During this process it became obvious when a data structure was manipulated incorrectly. More difficult bugs were revealed during the proof that each Kit routine implements the corresponding abstract kernel operation. The most insidious bug revealed at this stage was one in which the state of the current task was not accurately restored after processing an I/O interrupt. The bug caused a supervisor call request to be ignored if an I/O interrupt occurred immediately after the request, but before the request had been handled. Such time-dependent errors are difficult to find by testing.

The Kit project suffers from a number of problems and limitations. An obvious limitation is the sheer simplicity of Kit. We have not addressed the problem of dynamic allocation of objects (e.g., tasks, message ports) and the possibility of resource exhaustion in an implementation. Moore's subsequent work on Piton in the Boyer-Moore logic does address this problem. Also, Kit's memory management relies on an archaic method of memory protection. The specification and correct implementation of a virtual memory management scheme remains to be done. Kit does not directly address concurrency. In particular, it would be desirable to specify and prove correct a system which manages devices (e.g. a disk) operating concurrently and asynchronously with the processor.

The main value of the Kit project is its demonstration of what is possible. Kit is the first example of a mechanically checked proof of the correct implementation of a complete operating system kernel. The proof is carried out at the level of von Neumann machine code. The next obvious step is to extend the complexity of Kit as outlined above, and to target Kit down to a formalization of an existing processor.

The method used to verify Kit may also have applications in areas other than operating systems, such as small embedded systems. Kit demonstrates the feasibility of direct machine code verification. We believe that the difficulties we had verifying Kit's machine code can be overcome with an adequate set of libraries for the Boyer-Moore theorem prover. The method of arranging the theorem prover to act as a symbolic evaluator of machine code can scale up to systems larger than Kit.

Appendix A
The Boyer-Moore Logic

A complete and precise definition of the logic can be found in [Boyer 88].

We use the prefix syntax of Pure Lisp to write down terms. For example, we write **(PLUS I J)** where others might write *PLUS*(*I*,*J*) or *I*+*J*. We write **(IMPLIES (P X) (EQUAL (F X) (G X)))** in place of $P(X) \rightarrow F(X) = G(X)$.

The logic is first-order and contains no quantifiers. It is defined as an extension of propositional calculus with variables, function symbols, and the equality relation. Axioms define the following:

- the Boolean objects **(TRUE)** and **(FALSE)**, abbreviated **T** and **F**;
- The if-then-else function, **IF**, with the property that **(IF X Y Z)** is **Z** if **X** is **F** and **Y** otherwise;
- the Boolean "connector functions" **AND**, **OR**, **NOT**, and **IMPLIES**; for example, **(NOT P)** is **T** if **P** is **F** and **F** otherwise;
- the equality function **EQUAL**, with the property that **(EQUAL X Y)** is **T** or **F** according to whether **X** is **Y**;
- inductively constructed objects, including:
    - Natural Numbers. Natural numbers are built from the constant **(ZERO)** by successive applications of the constructor function **ADD1**. The function **NUMBERP** recognizes natural numbers, e.g., is **T** or **F** according to whether its argument is a natural number or not. The function **SUB1** returns the predecessor of a non-**0** natural number.
    - Ordered Pairs. Given two arbitrary objects, the function **CONS** returns an ordered pair containing them. The function **LISTP** recognizes such pairs. The functions **CAR** and **CDR** return the two components of such a pair.
- Each of the classes above is called a "shell", which can be thought of as a data type. **T** and **F** are each considered the elements of two singleton shells. Axioms insure that all shell classes are disjoint;
- the definitions of several useful functions, including:
    - **LESSP** which, when applied to two natural numbers, returns **T** or **F** according to whether the first is smaller than the second;
    - **COUNT** which, when applied to an inductively constructed object, returns its "size;" for example, the **COUNT** of an ordered pair is one greater than the sum of the **COUNT**s of the components.

The user can add of new shells, i.e., new data types. A shell defines a new class of **n**-tuples with type restrictions on each component. For each shell there is a recognizer (e.g., **LISTP** for the ordered pair shell), a constructor (e.g., **CONS**), an optional empty object (e.g., there is none for the ordered pairs but **(ZERO)** is the empty natural number), and **n** accessors (e.g., **CAR** and **CDR**).

In our work we use shells solely to define record structures with no type restrictions. The shell **FOO** below defines a record structure with three fields, **A**, **B** and **C**.

```
SHELL DEFINITION
FOO
with recognizer FOO-SHELLP,
and with accessors
A, type restriction: none,
B, type restriction: none,
C, type restriction: none.
```

The expression **(FOO 1 2 3)** builds one of these structures, with **1** in the **A** field, **2** in the **B** field and **3** in the **C** field. The expressions **(A X)**, **(B X)** and **(C X)** access the **A**, **B** and **C** fields of a **FOO** structure **X**, respectively. If **X** is a **FOO** structure, the expression **(FOO (A X) (B X) (BAR X))** builds a **FOO** structure equal to **X** in every field but the **C** field, which in this case takes on the value **(BAR X)**. We place type restrictions on the fields of a shell by defining a predicate which constrains each shell field.

The logic provides a principle of recursive definition under which new function symbols may be introduced. Consider the definition of the list concatenation function **APPEND**.

```
DEFINITION
(APPEND X Y)
=
(IF (LISTP X)
    (CONS (CAR X) (APPEND (CDR X) Y))
    Y)
```

The equations submitted as definitions are accepted as new axioms under certain conditions that guarantee that one and only one function satisfies the equation. One of the conditions is that certain derived formulas be theorems. Intuitively, these formulas insure that the recursion "terminates" by exhibiting a "measure" of the arguments that decreases, in a well-founded sense, in each recursion. A suitable derived formula for **APPEND** is the following.

```
(IMPLIES (LISTP X)
         (LESSP (COUNT (CDR X)) (COUNT X)))
```

However, in general the user of the logic is permitted to choose an arbitrary measure function (**COUNT** was chosen above) and one of several relations (**LESSP** above).

The rules of inference of the logic, in addition to those of propositional calculus and equality, include mathematical induction. The formulation of the induction principle is similar to that of the definitional principle. To justify an induction schema it is necessary to prove certain theorems that establish that, under a given measure, the inductive hypotheses are about "smaller" objects than the conclusion.

Using induction it is possible to prove such theorems as the associativity of **APPEND**.

```
THEOREM ASSOCIATIVITY-OF-APPEND
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C)))
```

References

[Bell 71] C.G. Bell, A. Newell.
Computer Structures: Readings and Examples.
McGraw-Hill, New York, 1971.

[Bell 75] D.E. Bell, L.J. LaPadula.
Secure Computer Systems: Unified Exposition and Multics Interpretation.
Technical Report MTR-2997, The Mitre Corporation, July, 1975.

[Berson 79] T.A. Berson, G.L. Barksdale, Jr.
KSOS - Development Methodology for a Secure Operating System.
In *AFIPS Conference Proceedings*, pages 365-371. 1979.

[Bevier 87] W.R. Bevier.
A Verified Operating System Kernel.
Technical Report 11, Computational Logic, Inc., 1717 W. 6th St., Suite 290, Austin, TX, 78703, October, 1987.

[Bevier, et. al. 89] W.R. Bevier, W.A. Hunt, J S. Moore, W.D. Young.
An Approach to Systems Verification.
To appear in The Journal of Automated Reasoning.
1989

[Boebert 85] W.E. Boebert, W.D. Young, R.Y. Kain, S.A. Hansohn.
Secure Ada Target: Issues, System, Design, and Verification.
In *Proceedings of the Symposium on Security and Privacy*, pages 176-183. 1985.

[Boyer 88] R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[Dijkstra 68] E.W. Dijkstra.
The Structure of the "THE"-Multiprogramming System.
CACM 11(5):341-346, May, 1968.

[Feiertag 77] R.J. Feiertag, K.N. Levitt, L. Robinson.
Proving Multilevel Security of a System Design.
In *Proceedings 6th ACM Symposium on Operating System Principles*, pages 57-65. 1977.

[Feiertag 79] R.J. Feiertag, P.G. Neumann.
The Foundations of a Provably Secure Operating System (PSOS).
In *AFIPS Conference Proceedings*, pages 329-334. 1979.

[Flon 77] L. Flon.
On the Design and Verification of Operating Systems.
PhD thesis, Carnegie-Mellon University, 1977.

[Fraim 83] L. Fraim.
Scomp: A Solution to the Multilevel Security Problem.
Computer 16(7):26-34, July, 1983.

[Gerhart 80] S. L. Gerhart, D. R. Musser, D. H. Thompson, D. A. Baker, R. L. Bates, R. W. Erickson, R. L. London, D. G. Taylor and D. S. Wile.
An Overview of AFFIRM: A Specification and Verification System.
In *Information Processing 80, S. H. Lavington (Ed.)*, pages 343-348. October, 1980.
North Holland Publishing Company.

[Gold 79] B.D. Gold, R.R.Linde, R.J. Peeler, M. Schaefer, J.F. Scheid, P.D. Ward.
A Security Retrofit of VM/370.
In *AFIPS Conference Proceedings*, pages 335-344. 1979.

[Good, et. al. 78] D. Good, et. al.
Report on the Language GYPSY Version 2.0.
Technical Report ICSCA-CMP-10, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1978.
Also available through Computational Logic, Inc., Suite 290, 1717 W. 6th Street, Austin, TX 78703.

[Hoare 72] C.A.R. Hoare.
Proof of Correctness of Data Representations.
Acta Informatica 1:271-281, 1972.

[Hoare 87] C.A.R. Hoare, J. He, J.W. Sanders.
Prespecification in Data Refinement.
Information Processing Letters 25:71-76, May, 1987.

[HOL 87] M. Gordon.
HOL: A Proof Generating System for Higher-Order Logic.
Technical Report 103, University of Cambridge, Computer Laboratory, 1987.

[Hunt 85] W.A. Hunt, Jr.
FM8501: A Verified Microprocessor.
Technical Report 47, Institute for Computing Science, University of Texas at Austin, December, 1985.

[Jones 86] C.B. Jones.
Systematic Software Development Using VDM.
Prentice-Hall, Englewood Cliffs, N.J., 1986.

[Karp 83] R.A. Karp.
Proving Operating Systems Correct.
UMI Research Press, Ann Arbor, Michigan, 1983.

[Kemmerer 82] R.A. Kemmerer.
Formal Verification of an Operating System Security Kernel.
UMI Research Press, Ann Arbor, Michigan, 1982.

[Landwehr 83] C.E. Landwehr.
The Best Available Technologies for Computer Security.
Computer 16(7):86-100, July, 1983.

[McCauley 79] E.J. McCauley, P.J. Drongowski.
KSOS - The Design of a Secure Operating System.
In *AFIPS Conference Proceedings*, pages 345-353. 1979.

[MelliarSmith 81] P.M. Melliar-Smith, R.L. Schwartz.
Hierarchical Specification of the SIFT Fault-Tolerant Flight Control System.
Technical Report CSL-123, SRI Computer Science Laboratory, March, 1981.

[Milner 71] R. Milner.
An Algebraic Definition of Simulation Between Programs.
Technical Report AIM-142, Stanford AI Project, February, 1971.

[Moore 88] J S. Moore.
A Mechanically Verified Language Implementation.
Technical Report CLI-30, Computational Logic, Inc., 1717 W. 6th St., Suite 290, Austin, TX, 78703, September, 1988.

[Neumann 77] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, L. Robinson.
A Provably Secure Operating System: The System, Its Applications, and Proofs.
Technical Report, SRI, February, 1977.

[Popek 78] G.J. Popek, D.A. Farber.
A Model for Verification of Data Security in Operating Systems.
CACM 21(9):737-749, September, 1978.

[Popek 79] G.J. Popek, M. Kampe, C.S. Kline, A. Stoughton, M. Urban, E. Walton.
UCLA Secure Unix.
In *AFIPS Conference Proceedings*, pages 355-364. 1979.

[Robinson 77] L. Robinson, K.N. Levitt, P.G. Neumann, A.R. Saxena.
A Formal Methodology for the Design of Operating System Software.
Current Trends in Programming Methodology, Volume I: Software Specification and Design.
Prentice-Hall, Englewood Cliffs, N.J., 1977.

[Robinson & Levitt 77]
L. Robinson and K. Levitt.
Proof Techniques for Hierarchically Structured Programs.
Comm. ACM 20(4), April, 1977.

[Rushby 81a] J. Rushby.
Proof of Separability: A Verification Technique for a Class of Security Kernels.
Technical Report SSM/8, Computing Laboratory, University of Newcastle upon Tyne, May, 1981.

[Rushby 81b] J. Rushby.
Specification and Design of Secure Systems.
Technical Report SSM/6, Computing Laboratory, University of Newcastle upon Tyne, March, 1981.

[Saxena 76] A.R. Saxena.
A Verified Specification of a Hierarchical Operating System.
PhD thesis, Stanford University, 1976.

[Shankar 86] N. Shankar.
Checking the proof of Godel's incompleteness theorem.
Technical Report, Institute for Computing Science, University of Texas at Austin, 1986.

[Spivey 88] J.M. Spivey.
Understanding Z: a Specification Language and its Formal Semantics.
Cambridge University Press, 1988.

[Walker 80] B.J. Walker, R.A. Kemmerer, G.J. Popek.
Specification and Verification of the UCLA Unix Security Kernel.
CACM 23(2):118-131, February, 1980.

[Young 88] W.D. Young.
A Verified Code Generator for a Subset of Gypsy.
Technical Report CLI-33, Computational Logic, Inc., 1717 W. 6th St., Suite 290, Austin, TX, 78703, November, 1988.

Table of Contents

1. Introduction 1
2. Interpreters and Interpreter Equivalence Theorems 2
   2.1. Machine States, Interpreters and Oracles 2
   2.2. Interpreter Equivalence Theorems 4
   2.3. The Correct Implementation of Kit's Specification 6
3. A Communicating Process 6
4. The Specification for Kit 9
5. The Abstract Kernel Implements Processes 13
6. The Target Machine 13
7. The Machine Code Implementation of Kit 17
8. The Machine Code Correctly Implements its Specification 18
9. Related Work 20
   9.1. Specification and Proof Methods 20
   9.2. Operating System Verification 21
10. Remarks 22

Appendix A. The Boyer-Moore Logic 24

References 26

List of Figures

Figure 1: Interpreter Equivalence (version 1) 5
Figure 2: Interpreter Equivalence (version 2) 5
Figure 3: Network 6
Figure 4: AK Implements Parallel Tasks 13
Figure 5: PMS Description of TM 14
Figure 6: The TM Clock Interrupt 16
Figure 7: TM's Instruction Set 16
Figure 8: Assembler Language Listing of State Saving Code 17
Figure 9: Final Theorem 19