Toward Verified Execution Environments

William R. Bevier
Warren A. Hunt, Jr.
William D. Young

Technical Report #54 February 1987

Also appears in
Proceedings of the 1987 IEEE Symposium on Security and Privacy.

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

1. Introduction

The Department of Defense *Trusted Computer Systems Evaluation Criteria* [1] mandates formal design verification for systems certified at level A1 and code verification for systems above A1. However, even code verification leaves a considerable assurance gap between the specified system and the compiled code executing on a piece of hardware. The security of the delivered system depends on the quality of the system specification and the soundness of the verification techniques. It also depends upon how faithfully the semantics of the verified specification is translated into an executable machine language program, which in turn depends upon the correctness of the underlying hardware and support software in the system's execution environment.

We use the term *vertically verified computing system* to refer to a computing system in which each layer of its architectural hierarchy has been formally proved to be correctly implemented by lower layers. This paper outlines an ambitious research program aimed at building the first vertically verified computing system. It consists of three components: a compiler, an operating system and a processor. The compiler translates Micro-Gypsy, a formally axiomatized subset of the language Gypsy [2], onto an instruction set defined by the processor augmented with operating system services. The translation is proved to preserve the semantics of the language. The operating system, supporting multiple tasks and resource management, is verified to provide task isolation and interprocess communication. The processor, named the FM8501, provides a standard Von Neumann architecture with a verified implementation.

The implementation of this vertically verified system will open the possibility of producing fully verified applications. A *fully verified program* is one which has been proved correct in a high-level verifiable language, such as Micro-Gypsy, and run on a computing system vertically verified down to the hardware level. A Micro-Gypsy application may be proved correct using existing verification tools [3], compiled into executable FM8501 machine code with the verified Micro-Gypsy compiler, and run on the FM8501 by the verified operating system. We can invest a much higher degree of confidence in such a system than is now possible in conventionally designed systems. Our belief in correct program execution in a vertically verified system depends only on the soundness of our verification tools and the correct fabrication of the processor.

The paper is structured as follows. The next section discusses abstractly the design and verification of a vertically verified system. Subsequent sections discuss the verification of the Micro-Gypsy compiler, the operating system, and the FM8501 processor. These sections have a similar structure; each discusses the specification, implementation, and correctness theorem for a component of the system. We follow with a more concrete explanation of how the components are integrated into a single system, and then conclude.

2. Vertically Verified Computing Systems

Each layer in a vertically verified system is a virtual machine implemented by lower layers. We formally define each virtual machine with an *interpreter function*. An interpreter function models transitions to a machine state occurring in a finite time span.

A machine that operates in complete isolation can be modeled simply by an interpreter function $Int : S \times N \rightarrow S$, where S is the set of machine states and N is the set of natural numbers. The result of $Int(s, n)$ is the machine state obtained by running interpreter Int for n steps with initial state s. This function models a machine in which a controlling program is embedded in the state. We find it convenient at times to model a machine in which the program state and data state are separate. Then we use the interpreter function $Int : P \times D \times N \rightarrow D$ as the model, where P is the set of program states, and D the set of data states.

To model a machine which is sensitive to the external world we generalize the integer time argument N. Such a machine is modeled as a function $Int : S \times O \rightarrow S$, where S is the set of machine states (in this case with embedded control state), and O is the set of *oracles* to which the machine responds. An oracle is a finite time-sequenced list of external events impinging on the machine. The length of an oracle defines the time span in which the machine operates. An element of an oracle is either a single external event, or a "tick" indicating no event. The interpreter consumes the next element of the oracle at each step, running until the oracle is exhausted.

The statement of correctness for each layer in a system takes the form of an *interpreter equivalence theorem*. Such a theorem establishes a relationship between two interpreters: an abstract specification interpreter and a concrete implementation interpreter. In each component of our system the abstract interpreter (and its associated state) is in some way "simpler" than its corresponding concrete interpreter. The proof of each component demonstrates the concrete interpreter's ability to mimic the operation of the abstract interpreter.

To prove the correlation between an abstract interpreter Int_A and a concrete interpreter Int_C we describe mapping functions which allow us to convert between the abstract state and the concrete state. The function $Map : S_A \text{ -> } S_C$ maps an abstract state to a concrete state. The function Map^{-1} maps a concrete state to an abstract state. The form of an interpreter equivalence theorem is

$$\begin{array}{ll} & \textit{There exists } k, \\ (*) & \quad \textit{Abstract-State-OK}\ (s_A) \text{ -> } Int_A\ (s_A, n) = Map^{-1}\ (Int_C\ (Map\ (s_A), k))\ . \end{array}$$

Figure 1 illustrates this theorem. The theorem says that given an initial legal abstract state s_A as defined by *Abstract-State-OK*, and time n, there exists a time k such that Map^{-1} of the concrete interpreter result is identical to the final state reached by the abstract interpreter. The theorem form is slightly more complex when an interpreter which requires an oracle argument is under consideration. We refer back to formula (*) throughout this paper to explain the correctness theorems for the components of our system.

Figure 1: Interpreter Equivalence

A vertically verified system can be built from verified components by "stacking" them. The stacking can be done when the concrete implementation interpreter for a component is precisely the abstract specification interpreter for the next lower component. By the transitivity of equality, we can infer from the interpreter equivalence theorems that the lowest layer implements the highest layer.

A picture of this stacking is shown in **Figure 2** for the system components described in this paper. In our system, the virtual machine expected by the Micro-Gypsy compiler is defined by a single task run by the operating system. The specification for the operating system spans two levels of **Figure 2**. Tasks running in parallel are implemented by an abstract multiplexed operating system, which in turn is implemented by machine code running on the FM8501. The processor and I/O resources required by the operating system are provided by the programmer-visible specification for the FM8501. The FM8501 is implemented by a representation of a hardware gate graph.

The composition of the correctness proofs of our three components is made possible by using a single formal logic. The Boyer-Moore logic [4, 5] is used for the specification of each interface and all proofs are constructed within this logic. The logic is a quantifier-free constructive first order logic with equality and rules for defining primitive recursive functions. The language is a form of pure-Lisp and consists of variables and function names combined in a prefix notation; the application of a function *f* to its argument *a* is written *(f a)*, instead of *f(a)*. Because of the absence of the existential quantifier, each instance of the interpreter equivalence theorem (*) which we describe in subsequent sections replaces the existentially quantified variable *k* with a "witness function" which computes the required value.

Figure 2 depicts the integration of three components, the design and proof of which are proceeding independently. Once the verification issues in each domain have been mastered, their integration into a vertically verified system can proceed. Significant issues require resolution in moving from three largely independent research efforts to the vertically verified system we envision. We return to the problem of integration in a later section after discussing the three independent verification efforts.

3. The Micro-Gypsy Compiler

The user of a vertically verified computing system can expect that his application programs will be accurately mapped into machine language programs running on verified hardware. An obvious step toward realizing this goal is the proof of the correctness of the compiler. If it happens that the compiler’s source language is a "verifiable" high level language such as Gypsy [2], the user will have the option of specifying and verifying his programs in the high level language before compiling. The result will be a fully verified system in which a very high degree of confidence can be invested.

In this section we describe the specification and verification of a compiler for a subset of Gypsy 2.05. The target language is the assembly language of the FM8501, the verified microprocessor described in a later section of this paper. The semantics of each language is specified by an interpreter and the correctness proof shows the equivalence of these interpreters as discussed in the previous section. The compiler is written as a collection of functions in the Boyer-Moore logic. Thus, it is a cross-compiler running as Lisp code compiled from these Boyer-Moore functions and generating assembly language for the FM8501.

Figure 2: A Vertically Verified System

3.1 Gypsy and Micro-Gypsy

Gypsy 2.05 is an integrated programming and specification language which is the language of choice for many secure system development efforts. The programming fragment of the language contains sophisticated data types, condition (exception) handling, data abstraction, and concurrency. The specification language permits specifications to be written as program assertions, state machines, or algebraic axioms. Processing of the language and constructing proofs of programs is carried out within a mechanized verification environment (GVE) which includes a parser, verification condition generator, algebraic simplifier, interactive proof checker, database, and other tools [3].

The verified compiler takes as source language Micro-Gypsy, a subset of Gypsy 2.05. The specification components of Gypsy are available in their entirety for specifying Micro-Gypsy programs; only the executable fragment of the language is restricted. The fact that Micro-Gypsy is strictly a subset of Gypsy allows programs written in the subset to be specified and proven using the existing tools of the GVE.

To ensure that the semantics of Micro-Gypsy used in the compiler proof corresponds to the semantics assumed by the GVE, the executable subset was chosen to correspond to the reduced language used in defining the formal semantics of Gypsy [6]. The result is a simple language with seven statement types adequate for expressing most sequential Gypsy programs; concurrency is not handled. The seven statement types are: **`no-op`**, **`signal`**, **`prog2`**, **`begin-when-end`**, **`if-then-else`**, **`loop`**, and **`procedure call`**. The **`signal`** statement allows the programmer to raise an exception; **`begin-when`** allows exception handlers to be placed on arbitrary blocks of code. **`prog2`** is the explicit sequencing operation of Micro-Gypsy used to obviate the need for statement lists of variable length. Other constructs in the language are translated to some combination of these seven statements. Only data of types **`integer`**, **`character`**, **`boolean`**, **`array`**, and **`record`** are handled.

The input to the verified compiler is an abstract syntax *prefix* form which is generated from Gypsy-like syntax by a preprocessor. The preprocessor handles lexical analysis and generation of abstract syntax. Expressions in the language are removed by the preprocessor in favor of calls to predefined procedures. It is only the code generation phase of compilation which is being verified. A necessary hypothesis of the proof, however, is that the prefix form satisfies a Micro-Gypsy prefix recognizer predicate **`ok-micro-gypsy`** which incorporates all of the syntactic and semantic checks required in parsing. A corresponding function **`ok-micro-gypsy-scope`** is used to recognize a list of procedure and type definitions. The eventual verification of the preprocessor will demonstrate that the translation of any correct Micro-Gypsy program is a semantically equivalent prefix form which satisfies this recognizer.

The semantics of the source language is defined by an interpreter **`mg-meaning`** for Micro-Gypsy written in the Boyer-Moore logic. A call to the interpreter takes the form

```
(mg-meaning stmt scope cond-alist-pair clk)
```

where

- **`stmt`** is a legal Micro-Gypsy statement (the execution entry point);
- **`scope`** is a list of legal Micro-Gypsy procedure and type definitions;
- **`cond-alist-pair`** is an abstract execution environment consisting of an assignment of values to program variables (the variable a-list) and the current condition (**`normal`** or some exception value);
- **`clk`** is a integer counter used to guarantee termination of the Boyer-Moore definition of the interpreter.

Notice that **`mg-meaning`** differs from the abstract interpreter discussed earlier; the program has been "unbundled" from the rest of the state. That is, the abstract state on which the interpreter is called is distributed among the three arguments **stmt**, **scope**, and **cond-alist-pair**. This has been done to emphasize that for the compiler, more than for the other interpreter equivalence theorems we will discuss, the translation of the program component of the state is of paramount concern. The interpreter returns the state which results from executing the entry point in the context of the abstract state defined by the procedure and type list, the variable a-list and current condition.

The interpreter semantics for **prog2** given below illustrates the style of the definition.

```
(mg-meaning prog2-form scope cond-alist-pair clk)
=
(mg-meaning (arg2 prog2-form)
            scope
            (mg-meaning (arg1 prog2-form)
                        scope
                        cond-alist-pair
                        clk)
            clk)
```

The symbol **prog2-form** represents a Micro-Gypsy statement of the form **(prog2 <statement1> <statement2>)**. **mg-meaning** tells us that **<statement2>** is evaluated in the state resulting from evaluation of **<statement1>**.

3.2 The Target Language

The target language of the compiler is an abstract assembly language for the FM8501, the verified microprocessor described later in this paper. This is a rather atypical assembly language in that it is has procedures with separate individual name spaces. This facilitates separate compilation of Micro-Gypsy procedures. The proof that this functionality can be correctly provided on the bare FM8501, *i.e.*, a proof of the correctness of the assembler, is being carried on separately and will not be discussed here. The target language is also envisioned to allow calls for the services provided by the verified operating systems; the current compiler, however, does not make any use of these services.

The semantics of this language, as with Micro-Gypsy, is defined by an interpreter. The input to this interpreter is a state consisting of the following components

- **pc** is the 0-based program counter.
- **flags** is a representation of the four hardware flags of the FM8501 processor.
- **stacks** contains a representation of the data, control, and procedure return stacks.
- **locals** and **globals** are association lists binding variable names to numeric values.
- **prog** is the currently executing program.
- **constants** contains a number of execution parameters including the word size, bounds of the stacks, bindings of global constants, and the list of procedure bodies.
- **psw** is the run status or the termination status.
- **clk** is the number of steps to be executed.

Notice that unlike the Micro-Gypsy interpreter, the program to be executed is considered a component of the state.

The assembly language interpreter operates by fetching an instruction from **prog** in accordance with the value of **pc**, executing that instruction, and updating the state. Execution terminates when the value of the **psw** becomes any value besides ’run, either because the value of **clk** has decreased to zero or because of some "error" condition. The Boyer-Moore definition of the interpreter is simply

```
(interpret state)
=
(if (equal (clk state) 0)
    (halt state)
    (interpret (step (fetch-instr (pc state) (prog state))
                     state))),
```

where **halt** terminates execution. The **step** function describes the behavior of each of the FM8501 instruction types on the state. Notice that this differs from the dyadic interpreter model described earlier in that the clk argument is part of the state.

3.3 The Translation and the Correctness Theorem

The translation from Micro-Gypsy to FM8501 assembly language is fairly naive. Temporaries are not reused, for example. We envision adding some optimizations later. The basic data structure of the translator is a list **cinfo** of five elements;

- **code** is a list which is initially **nil** to which the generated code is appended.
- **vars-list** is an association-list binding variables and values. As temporaries are generated they are added to this list along with their initial-values.
- **label-alist** associates conditions which might be signalled with labels designating locations in the code.
- **label-cnt** and **temp-cnt** are integer counters for generating unique label and temporary variable names.

The translation of each of the Micro-Gypsy statement types is handled by the recursive function **translate-statement**. Another function **translate-scope** takes a list of Micro-Gypsy procedure and type definitions and turns it into a list of assembly language procedures. The translation style is illustrated below with the translation for the Micro-Gypsy **prog2** statement.

```
(translate-statement cinfo prog2-form)
=
(translate-statement
     (translate-statement cinfo (arg1 prog2-form))
     (arg2 prog2-form))
```

Again **prog2-form** is an instance of the statement schema **(prog2 <statement1> <statement2>)**. **<statement2>** is compiled in the context of the compilation of **<statement1>**.

The statement of correctness of the compiler is an interpreter equivalence theorem proven by induction on the structure of the Micro-Gypsy code. Conceptually, it can be stated as follows: given an arbitrary correct Micro-Gypsy program, the behavior of that program on a legal abstract state is accurately reflected by the behavior of the translation of that program on a corresponding concrete state. A somewhat simplified formal version of this is given below.

```
Theorem: Correctness-of-the-Compiler

 (implies (ok-micro-gypsy-state stmt scope cond-alist-pair)
          (equal (mg-meaning stmt scope cond-alist-pair clk)
                 (map-up cond-alist-pair
                         (interpret
                             (compile stmt scope
                                      cond-alist-pair clk)))))
```

Notice that this is an instance of the abstract interpreter equivalence theorem (*). The call to

`ok-micro-gypsy-state` implements the compiler’s version of *Abstract-State-OK* by making calls to **`ok-micro-gypsy`**, **`ok-micro-gypsy-scope`**, and other functions to check that the interpreter is called on a legitimate abstract state. The function **`compile`** implements the abstract *Map* relation. It takes a Micro-Gypsy entry point, list of procedures and types, condition/alist pair, and a clock parameter and returns the FM8501 state which results from compiling the Micro-Gypsy program. Values of locals in the FM8501 state are set to correspond to the values in the Micro-Gypsy abstract state. **`map-up`** is the analogue of the abstract Map^{-1} function. It selects out of the FM8501 state (which is its second argument) those components which have correlates in the Micro-Gypsy cond-alist pair. These are: the value of the global variable used to store the current condition value, and a subset of the variables.

3.4 Summary

We have discussed the specification and verification of the code generation phase of a compiler for a subset of Gypsy 2.05. The specification of both source and target languages are written an interpreters over states. The correctness of the translation from Micro-Gypsy to the assembly language for the FM8501 is demonstrated by showing that an arbitrary correctly-formed Micro-Gypsy program has the "same" effect on the abstract state as its assembly language correlate has on the concrete state.

Much work remains before the Micro-Gypsy compiler can a useful tool for generating fully verified systems. The proof described in this section has begun, but a significant amount of work remains. The complete specification of both the Micro-Gypsy preprocessor and the FM8501 assembler will also need to be completed before the claim can be made that there is a verified chain from high level language programs to running machine code. Micro-Gypsy is a realistic language in that many non-concurrent Gypsy programs could be translated straightforwardly into Micro-Gypsy. The principle deficiency of the language is the simplicity of the data types. Future extensions of the language may incorporate the Gypsy dynamic data types. This will involve more attention to the details of dynamic storage allocation than is required in the current version. Treatment of Gypsy concurrency is also possible.

4. The Operating System

The purpose of the work described in this section is to verify the kernel of an operating system. The kernel is responsible for simulating multiple parallel tasks, handling communication among tasks, and responding to asynchronous devices. The absence of unverified code is an important requirement on this project. As a result, we verify the operating system at the machine code level. The assumed base for the verification is only the specification for the target processor.

The verification of the operating system includes the following results:

- Isolation of tasks,
- Protection of the operating system from tasks,
- Correctness of operating system services,
- Termination of the operating system.

4.1 A Specification for Parallel Tasks

We specify an operating system which supports a fixed number of tasks, up to some maximum number. The virtual machine provided by the kernel is sufficient to implement a simple message-oriented system. The services provided are

- Task scheduling and allocation of CPU time,
- Response to program error conditions (e.g. divide by zero),
- Single-word message passing among tasks,
- Character I/O to asynchronous devices.

The operating system implements communicating tasks executing in parallel. The specification for a single task is a task interpreter function. There are two difficulties in the definition of this function: (1) how to model transitions to shared state that are made by other tasks, and therefore appear to be non-deterministic with respect to a given task; and (2) how to model time delays to a task which prevent simultaneous updates to shared state. Each problem requires the presence of a function which knows something about the state of all tasks.

The specification for $\mathtt{task}_i$ in a system of parallel tasks is given by the interpreter function **task-processor**. The purpose of **task-processor** is to perform transitions on the private state space of $\mathtt{task}_i$. To do this, transitions to shared state must also be tracked.

```
(task-processor task i world n)
=
(if (zerop n)
    task
    (task-processor (task-step task i world)
                    i
                    (world-step task i world)
                    (sub1 n)))

(task-step task i world)
=
(if (task-activep task i world)
    (task-private-step task i world)
    task)
```

In this definition **task** is the private state space of a task, **i** is the identifier of the task, **world** contains the entire state of the world, and **n** is an integer which defines the time span for operation of the task interpreter. On each step, **task-step** performs a transition on the private state of **task**, and **world-step** reports the next state of the world. The result of **task-step** depends on whether $\mathtt{task}_i$ is active as defined by the predicate **task-activep**. If so, **task-step** returns the next private state of $\mathtt{task}_i$. Otherwise, **task** is returned unchanged.

The two functions with global knowledge in this definition are **task-activep** and **world-step**. They do not have the same logical status as **task-step** in the specification. **task-step** specifies the legal transitions on a task's private state space. The function **task-processor** models a sequence of task transitions through time as defined by **task-step** *if* the functions **world-step** and **task-activep** can be defined. The definitions of **world-step** and **task-activep** depend on an implementation of parallel tasks. The function **task-activep** need not make the restriction that only one task at a time is active, but it must arrange so that simultaneous updates to shared state are impossible.

The operating system is shown to implement parallel tasks, where each task is defined by the function

`task-processor`. This proof is accomplished in two steps. First, we prove that parallel tasks are implemented by an abstract multi-tasking operating system. This operating system has a state space in which the private state spaces of tasks are easily seen to be isolated, and gives us definitions for **`world-step`** and **`task-activep`**. The second step is to show that this abstract operating system is implemented by machine code running on a hardware processor. In the next section we give the specification for the abstract operating system, and state the theorem which establishes that parallel tasks are implemented. In subsequent sections we discuss the implementation and verification of the abstract operating system.

4.2 A Specification for a Multi-Tasking Operating System

The specification for the operating system is an interpreter which performs transitions over an abstract operating system state. The specification is a low-level one which accomplishes two things: sub-spaces of the abstract state are transparently isolated from one another, and services performed by the operating system are described in a functional notation. All of the finiteness properties of the operating system are present in the specification: message and I/O buffers are bounded in size, the number of tasks is static, control variables are bounded integers, communication channels are fixed. In addition, the specification has an interrupt structure similar to that of the target machine. A higher level specification might abstract away some of these features. For instance, in a more abstract setting we might want to view asynchronous devices as processes and treat interrupts as messages. The specification we give is an intermediate step between such high-level views and the implementation.

The function **`os-processor`** models the abstract interpreter. It behaves like a single task processor which is multiplexed over the tasks. It maps the set **`os-state`** x **`events`** to the set **`os-state`**. An element of **`os-state`** is an abstract operating system state. The set **`events`** is an oracle argument, as described in section 2. The **`events`** argument to **`os-processor`** is a list of events from the world external to the interpreter. For this specification there are three kinds of events: an *input interrupt* is a 2-tuple containing a device identifier and an input character, an *output interrupt* contains just a device identifier indicating completion of an output, and a *tick* is a symbol indicating no interrupt.

```
(os-processor os-state events)
=
(if (not (listp events))
    os-state
    (os-processor (os-step os-state (first events))
                  (rest events)))
```

The interpreter **`os-processor`** applies a sequence of steps to an initial **`os-state`**, consuming the next element of **`events`** on each step. The exact sequence of steps is determined both by the state of the machine and the timing of external events.

An element of the set **`os-state`** is a record structure with the following fields:

- **`os-tasks`** is a finite array, each element of which is the private state space of a single task. The private states are completely isolated. The state space of a task can be thought of as a single address space of a target machine. We leave this space unspecified until the target machine is selected.
- **`os-taskid`** is a bounded non-negative integer which contains the id of the current task. It is an index into **`os-tasks`**.
- **`os-readyq`** is a queue of the task identifiers of ready tasks.
- **`os-status`** is a table which gives the current status of each task.
- **`os-channels`** is a table containing the message buffers used for inter-task communication.

- **`os-inputs`** is an array of buffers, where each buffer queues the input arriving from an asynchronous character-oriented device.
- **`os-outputs`** is an array of buffers, where each buffer queues the output being sent to an asynchronous character-oriented device.
- **`os-wrstate`** is a bit flag indicating the **`wait/running`** state of the machine.
- **`os-clock`** is a bounded non-negative integer which is set to grant a time-slice to a task.

A predicate **`good-os-state`** recognizes a valid state of the abstract operating system. This predicate places all the necessary finite restrictions on a legal abstract state.

The function **`os-step`** defines a single step of the machine. It reveals how we specify device handling, error handling, time-sharing and supervisor services.

```
(os-step os-state event)
=
(if (os-io-interruptp event)
    (os-io-interrupt-handler os-state event)

(if (os-waiting os-state)
    os-state

(if (os-task-errorp os-state)
    (os-error-handler os-state)

(if (os-clock-interruptp os-state)
    (os-clock-interrupt-handler os-state)

    (os-fetch-execute os-state)))))
```

The current machine state and external event may cause an interrupt to the fetch-execute cycle. **`os-step`** senses and responds to interrupts as follows:

1. Check for an I/O interrupt. The function **`os-io-interrupt-handler`** is the specification function for I/O interrupt processing.
2. Check the wait-state bit. Do nothing if the machine is in the wait state. The operating system can put the machine in the wait state if there are no ready tasks.
3. Check for a program error condition in the current task. The function **`os-error-handler`** specifies the actions of an error-trap routine.
4. Check for a clock interrupt. If the clock signals an interrupt, the operating system stops the current task and picks another. The function **`os-clock-interrupt-handler`** specifies this operation.
5. If none of the above conditions exist, fetch and execute the next instruction from the current task. This is done in the function **`os-fetch-execute`**. An instruction may be either a supervisor call which is handled by the operating system, or a primitive instruction which is handled by the target machine. The function **`os-supervisor-call-handler`** (not shown here) is the top-level function in the specification of all supervisor services. To execute a primitive instruction we apply the target machine's instruction execution algorithm to the private state of the current task.

The specification for operating system machine code is contained in the functions **`os-io-interrupt-handler`**, **`os-error-handler`**, **`os-clock-interrupt-handler`** and **`os-supervisor-call-handler`**. Each is a function in the Boyer-Moore logic which completely specifies a service of the operating system.

The function **`os-processor`** serves both as the specification for a multi-tasking operating system, and an implementation of a system a parallel tasks. An interpreter equivalence theorem relating **`task-processor`** to **`os-processor`** establishes that **`os-processor`** implements parallel tasks as follows.

```
Theorem: OS-Implements-Parallel-Tasks

  (implies (and (good-os-state s)
                (good-task task)
                (good-taskid id)
                (equal n (length events)))
           (equal (task-processor task id (inject task id s) n)
                  (project id (os-processor (inject task id s)
                                            events))))
```

This theorem is depicted in **Figure 3**. It states that the same final private state of a task is reached whether it is run as an independent task by **`task-processor`**, or whether it is injected into the state of **`os-processor`** and run as one of many tasks. This proof is simple because of the clean state space of the abstract operating system. The operating system suggests the definitions we need for **`task-activep`** and **`world-step`**. The function **`world-step`** is defined to be **`os-step`**. The function **`task-activep`** is a predicate which has a structure identical to **`os-step`**. It solves the problem of simultaneous updates to shared state because it permits only one task to be active at any instant. (The symbol **`f`** means *False* in the Boyer-Moore logic.)

```
(task-activep task id os-state event)
=
(if (os-io-interruptp event)
    f
(if (os-waiting os-state)
    f
(if (os-task-errorp os-state)
    f
(if (os-clock-interruptp os-state)
    f
    (equal id (os-taskid os-state))))))
```

4.3 The Target Machine

The machine to which the operating system is targeted, called **`TM`**, is similar to but not identical to the FM8501. The FM8501 currently does not have a number of architectural features required to implement multiple tasks. **`TM`** is defined by a collection of functions in the Boyer-Moore logic. It has the following features.

- An 8-bit general purpose processor.
- Two operating states: supervisor and user, with privileged instructions available only in supervisor mode.
- Word addressing.
- Eight general purpose registers which include a program counter and stack pointer.
- A memory containing a static segment structure of sixteen 1024-word segments. Stack instructions access the high-address end of a segment.
- Condition code, error code, and wait/running state flag.
- Two-address instruction format with four address modes: immediate, memory direct, register direct, and register indirect
- An ALU which provides arithmetic operations on natural numbers.
- Interrupt driven character I/O.

Figure 3: Operating System Implements Parallel Tasks

This machine provides simple architectural support for a system which runs a fixed number of tasks. We use one segment of memory for the operating system, allowing fifteen user tasks. We have somewhat under-specified the machine. Only those features which are used by the operating system have been made definite. For instance, the fact that the ALU is specified only for natural numbers is not a requirement, but results from the absence of negative numbers in the algorithms coded in the operating system. The specific word and address sizes could be changed with no impact on the proof. We instantiated the machine size so that we could simulate its operation.

The target processor is modeled as a function **`tm-processor`** which maps the set **`tm-state`** x **`events`** to **`tm-state`**, where **`events`** is the set of external event lists described for the operating system specification, and **`tm-state`** is the set of target machine states. The definition of a target machine state is a formalization of the features described above.

```
(tm-processor tm-state events)
=
(if (not (listp events))
    tm-state
    (tm-processor
      (tm-step tm-state (first events))
      (rest events)))
```

The target machine step function **`tm-step`** responds to all the interrupts to which **`os-step`** is sensitive, plus a supervisor call interrupt. A supervisor call in the target machine requires a state change from user to supervisor mode and a branch to an operating system routine. A supervisor call in the specification appears as just another primitive operation.

```
(tm-step tm-state event)
=
(if (tm-io-interruptp event)
    (tm-io-interrupt tm-state event)

(if (tm-waiting tm-state)
    tm-state

(if (tm-task-errorp tm-state)
    (tm-error-interrupt tm-state)

(if (tm-svc-interruptp tm-state)
    (tm-svc-interrupt tm-state)

(if (tm-clock-interruptp tm-state)
    (tm-clock-interrupt tm-state)

    (tm-fetch-execute tm-state))))))
```

It is no accident that **`os-step`** and **`tm-step`** are similar. **`os-step`** was written so that it closely resembles the interrupt structure of the target machine. The only thing it abstracts away is the target machine's supervisor call mechanism. An interrupt triggers a call to a specification function in the abstract machine, while it causes merely a partial state swap which includes the program counter, and therefore a branch to an operating system program, in the target machine. The similar structure paves the way for an inductive proof of the correctness of the operating system.

4.4 The Operating System Correctness Theorem

The correctness theorem states a correspondence between **`os-processor`** and a **`tm-processor`** running the operating system. The theorem has the form of the interpreter equivalence theorem (*). It says: given an initial legal abstract state **`state`** and abstract external events **`events`**, there exists a concrete event list (constructed by the function **`map-time`**) for which running the operating system on the initial concrete state and mapping up the result gives exactly the final abstract state.

```
Theorem:  Correctness-of-Operating-System

  (implies (good-os-state state)
           (equal (os-processor state events)
                  (map-state-up
                       (tm-processor (map-state-down state)
                                     (map-time state events)))))
```

The function **`map-state-down`** maps the state of the abstract operating system into a target machine state, thereby constructing the initial concrete state. It maps the shared and control data structures into locations in segment 0 of the target machine's memory where they are managed by the operating system. Some of the control registers of an **`os-state`** are mapped onto analogous registers in the target machine. The private state space of a task is a single address space of the target machine. The target machine supports a segmented memory, but requires sharing of its CPU registers. Therefore **`map-state-down`** maps a task's memory resources into a single segment of the target machine's memory, and its registers into a virtual register table in the memory segment owned by the operating system. The context switch performed by the operating system when an interrupt occurs accesses this table and is verified to work correctly. The function **`map-state-up`** inverts the effect of **`map-state-down`**.

The function **`map-time`** accounts for the difference in speed between **`os-processor`** and a **`tm-processor`** running the operating system. Each operating system service in **`os-processor`** is a function call which consumes a single abstract tick. A call to an operating system routine running on **`tm-processor`** requires many concrete ticks. **`map-time`** determines how much time the target machine requires to match the operations of the specification machine.

The interpreter equivalence proof requires tracing through all paths of the operating system machine code, and showing that the final state reached by each path is equivalent to the state produced by a corresponding specification function. We get termination of each operating system routine as a result of this proof method.

The following features of the target machine contribute to the proof of the correctness of the operating system.

- The segment structure of memory. *Some* memory management mechanism must be built into a machine if we hope to prove task separation without having to examine the code of each task. We have chosen an extremely simple segmentation scheme. A more flexible mechanism would of course be desirable.
- The non-interruptibility of the machine when in supervisor mode.

4.5 Summary

We have outlined how we specify and prove correct a small message-oriented operating system. The operating system specification is written as an interpreter over an abstract operating system state. This specification is shown to implement a system of communicating parallel tasks. The sub-spaces of the abstract state are transparently isolated from one another, and the services provided by the operating system are specified in a functional notation. This is a low level specification which can be used as a foundation for proofs of more abstract properties of the operating system. The correctness of the implementation is accomplished by showing that the state of the target machine running the operating system always corresponds to the abstract state.

The simplicity of the operating system is a major factor in our ability to mechanically check its correctness. Despite its simplicity, the gap between the specification and implementation is still quite large. This is due largely to the programming language gap between abstract Boyer-Moore functions and machine-code level operations. The gap could be reduced if we could write much of the operating system in a high level source language and verify it at that level. Of course, the high level language would require a verified compiler. This suggests that Micro-Gypsy might become a base for operating systems proofs in the future.

As mentioned earlier in this paper, the next verified operating system will be written for the target machine which is the successor to the FM8501. This will be done to forge one of the links in the vertically verified system architecture. Other work desirable in the future is to verify an operating system with more flexible characteristics such as process creation and deletion, channel creation and deletion, and dynamic memory management.

5. The Hardware

The lowest level component in our system is the formally specified and mechanically verified FM8501 microprocessor [7, 8]. The FM8501 demonstrates the use of formal logic as a representation format for combinational and sequential digital logic. Mathematical operations are defined using the Boyer-Moore logic; these operations are then shown equivalent to some graph of hardware gates. The mathematical operations are characterized by commonly used functions, such as addition, subtraction, and shifting. The implementation (gate graph) is characterized by Boolean functions applied to components of bit-vectors.

The FM8501 effort shows that a microprocessor or like device is better specified with several pages of formulae than with hundreds of pages of text. The compact formal specification is better suited to the task of description: it takes less time to read, it is unambiguous, and it provides a precise basis for our vertically verified computing system. The verification of the FM8501 demonstrates the logical perfection of one implementation.

5.1 The FM8501 Abstract Specification

The FM8501 is a complete, stand-alone microprocessor with a symmetrically organized instruction set. Its features include:

- a 16-bit general purpose processor, with 16-bit instructions;
- word addressing yielding a 64K word (128K byte) memory size;
- eight general purpose registers (one also being the program counter) and four condition code flags;
- a two-address instruction format which allows register-register, register-memory, or memory-memory operation for all instructions;
- register, register indirect, register indirect with post-increment, or register indirect with pre-decrement addressing modes are individually supported for both operands for all instructions;
- a general-purpose conditional-move instruction;
- a separate ALU for effective address generation.

The abstract specification function for the FM8501, **`soft-int`**, is an interpreter defined at the instruction level. Each time the abstract interpreter "ticks" an instruction is fetched and executed. The function **`soft-int`** takes an abstract state and a clock as arguments. The abstract state is the programmer-visible state of the processor. This is a subset of the processor's full internal state. The abstract state is composed of a 2^{16} element by 16-bit RAM, an eight element by 16-bit register file, and four one bit condition codes: carry, overflow, zero, and negative. The list **`list`** replaces the integer clock argument *n* in our general interpreter equivalence theorem (*). Each time **`soft-int`** recurs, it checks to see if there is remaining time with the test **`(not (listp list))`**. If so, **`soft-int`** calls itself, modifying the programmer visible state. The new value of **`prog-state`** is computed by the function **`next-macro-state`** taking the current **`prog-state`** as its argument.

```
(soft-int prog-state list)
=
(if (not (listp list))
    prog-state
    (soft-int (next-macro-state prog-state)
              (rest list)))

(next-macro-state prog-state)
=
(list (next-memory prog-state)
      (next-register-file prog-state)
      (next-carry-flag prog-state)
      (next-overflow-flag prog-state)
      (next-zero-flag prog-state)
      (next-negative-flag prog-state))
```

Each component of the programmer-visible state is computed in isolation.

- **`next-memory`** returns the next memory state conditionally updated with the result of the ALU computation;
- **`next-register-file`** determines the value of register file. This function is a composition of six updates as follows: PC increment, operand-a pre-decrement, operand-b pre-decrement, store ALU results, operand-a post-increment, and operand-b post-increment. Except for the PC increment, all updates are conditional upon the current instruction;
- **`next-carry-flag`** sets the carry condition code flag. The other flags are are assigned values in a manner similar to the carry flag.

The clock parameter **`list`** plays no role in the specification of the FM8501 except to make the function **`soft-int`** total.

5.2 The FM8501 Hardware Interpreter

The FM8501 hardware implementation is modeled as an interpreter. It corresponds to the concrete interpreter Int_C in (*). This interpreter works at the fundamental clock frequency of the microprocessor and defines the internal working of the FM8501. The FM8501 concrete interpreter represents three different aspects of the microprocessor: the timed nature of this device, the state-holding devices, and the combinational logic that transforms the state in each time unit.

An internal block diagram for the FM8501 is shown in **Figure 4**. Data enters the microprocessor at the top left-hand corner of the figure and leaves at the top right-hand corner. An input datum may flow into the instruction register or into one of the ALU input latches. Output from the ALU can be stored into the register file or memory after being latched in the data output register. The instruction register is located in the middle of the figure and, along with the microcode, controls the execution of instructions. Shown in the lower left-hand corner are an incrementer and a decrementer. These are used to increment the program counter and for pre-decrement and post-increment address computation.

The low-level characterization of the FM8501 is concerned only with bistate logic and logic devices typically used when building digital computers; these devices include *and*, *or*, *not*, *nand*, *nor*, *equv*, and *xor* gates, latches, register files, RAMs, and ROMs. To model time-sequenced computing devices a register-transfer style language is used [9]. Register-transfer languages describe digital hardware as combinational logic separated by clocked registers.

Combinational logic is described by recursively defined functions that map bit-vector inputs to a bit-vector output. We often define such hardware functions to work on arbitrarily sized data. For example, instead of defining the combinational logic for a 16-bit wide adder, we define a recursive function for adding *n*-bit wide vectors. However, in their eventual use, hardware functions are applied to symbolic expressions denoting bit-vectors of fixed size. Such applications can be expanded by symbolic evaluation to an equivalent expression involving only the seven Boolean gate functions: not, and, or, nand, nor, xor, and equv, and "glue" for creating bit-vectors.

To model sequencing we define recursive functions which take a clock argument plus other arguments representing state holding devices such as registers, memory, etc. Such hardware interpreter functions have a stylized definitional form: they are called recursively once each clock tick. The arguments to the recursive call specify how the state is modified each cycle as a function of the previous state. The final state is returned, when the clock is exhausted.

The FM8501 hardware interpreter **`big-machine-int`** takes a concrete state and a clock as arguments. The state argument **`concrete-state`** is composed of two parts: the internal state (the instruction register,

Figure 4: An Internal Block Diagram of the FM8501

data sequencing latches, the microaddress register, etc.) and the programmer-visible state (the abstract state). **`concrete-state`** is the union of the internal state and the programmer-visible state. The clock argument is denoted by the variable **`clk`**.

```
(big-machine-int concrete-state clk)
=
(if (not (listp clk))
    concrete-state
    (big-machine-int (next-micro-state concrete-state
                                       (first clk))
                     (rest clk)))
```

The function **`next-micro-state`** takes two arguments: the current state and the first element of clock. Current values for the reset and data-acknowledgment inputs are found in **`(first clk)`**. These values are unconstrained and are used to mimic (possibly random) external events which play a part in the operation of the FM8501.

The function **`next-micro-state`** is similar to the function **`next-macro-state`**. Each new component of the state is computed by a hardware function which takes **`concrete-state`** as argument. These hardware functions model combinational logic and are eventually expanded into Boolean functions composed of gates.

5.3 The Correctness of the FM8501

We establish the correctness of the FM8501 by proving a correspondence between the abstract interpreter **`soft-int`** and the concrete interpreter **`big-machine-int`**. The correctness theorem states, roughly, that any final state computed by **`soft-int`** can be computed by **`big-machine-int`** by running it long enough on the appropriate initial state. More precisely, the interpreter equivalence theorem tells us that if we start in an appropriate concrete state, then there exists a suitable clock for making **`big-machine-int`** compute **`soft-int`**.

```
Theorem:  Correctness-of-the-FM8501

  (implies (standard-hyps-int
                     (init (make-concrete-state prog-state)))
           (equal (soft-int prog-state list)
                  (abs-prog-state
                      (big-machine-int
                          (init (make-concrete-state prog-state))
                          (suitable-clock list)))))
```

The predicate **`standard-hyps-int`** recognizes a legal concrete state; it requires fixed register and RAM sizes. This function serves the role of *Abstract-State-OK* in (*), as the abstract state is a subset of the concrete state. In addition to being a legal state, the initial concrete state must have the FM8501 poised to execute instructions, which is accomplished by the function **`init`**. The function **`make-concrete-state`** creates a concrete state from any abstract state; this is the function *Map* in (*).

Recall that **`big-machine-int`** returns a final state which contains both the programmer-visible state and the internal state. On the other hand, **`soft-int`** returns only the programmer-visible state. To state the equivalence of the two functions we project out of the **`big-machine-int`** state only the programmer visible state with the function **`abs-prog-state`**, which is analogous to *Map*$^{-1}$ in (*).

As in the verification of the operating system, the abstract and concrete interpreters for the hardware run at different speeds: **`big-machine-int`** has a "faster" clock than **`soft-int`**. Many ticks of the **`big-machine-int`** clock are required to implement one tick of the **`soft-int`** clock. The function **`suitable-clock`** computes the clock value required by **`big-machine-int`** to compute the final state reached by **`soft-int`**.

5.4 Summary

The accurate specification of computing system operation begins with the hardware itself. Several aspects of the FM8501 formalization are:

- descriptions of arbitrary sized combinational logic,
- characterization of hardware devices within a formal theory,
- mathematically constructed data types for bit-vectors, natural numbers, and integers,
- formal representation of microcode,
- mathematical characterization of individual microcode states, transitions, and timing considerations,
- mathematical specification of von Neumann like devices.

The verification of the FM8501 is interesting as all proofs were performed by a heuristically guided mechanical theorem prover. The theorem prover allowed a very large number of proofs to be reliably performed, certainly many more than could have been done by hand.

The specification and verification of the FM8501 provides upper-level software (the operating system and the compiler) with a formal basis. The FM8501 specification interpreter provides the concrete interpreter for programs being executed by the FM8501. The original FM8501 work was completed in 1985, and we anticipate constructing the FM8501 in the near future.

6. Achieving A Vertically Verified System

The verification of the FM8501 is complete. The verification of the compiler and operating system are in progress. When these proofs are finished, our system will still be segregated as shown in **Figure 5**. The compiler is targeted to the FM8501, which in turn is implemented at the level of gates. The operating system is targeted to a machine TM similar to FM8501, but containing the architectural support required for multiple tasks and asynchronous I/O.

The problem of integrating this work to achieve **Figure 2** is not trivial. The definitions of TM and the FM8501 must be merged to integrate the operating system with the processor. The compiler must be re-targeted from the FM8501 to a virtual task defined by the operating system. We expand on these points below.

The FM8501 must be extended to include supervisor/user states, memory management capabilities, and device interrupts. These are the features we need to support a multi-tasking operating system. We will call this machine the REVISED-FM. It will also supersede TM. A new verified operating system will be targeted to the REVISED-FM. This will complete **Figure 2** up to the level of tasks.

The compiler is currently independent of the operating system. The semantics of Micro-Gypsy can be implemented completely by FM8501, and does not require any software-implemented primitives. The integration step will give us an opportunity to add to Micro-Gypsy the primitives like message passing which are implemented by the operating system. The Micro-Gypsy compiler will be targeted to a virtual task defined

Figure 5: State of system after first round of verification

by the operating system rather than directly to REVISED-FM. This is feasible because a task is nothing more than a single address space of the REVISED-FM together with a processor that interprets both the primitive instruction set of REVISED-FM plus the services defined by the operating system. Therefore, lifting the proof of the compiler above the level of the operating system will not significantly change the target, except to make available additional primitives provided by the operating system.

7. Conclusion

The goal of this research is to build a vertically verified computing system. This system will provide a verified run-time environment consisting of a multi-tasking operating system and a processor, and a verified compiler for programs written in the high-level language Micro-Gypsy. Each component of our system is specified in the Boyer-Moore logic, and its correctness proof is mechanically checked by the Boyer-Moore theorem prover. The correctness theorem for each component is an interpreter equivalence theorem, where the concrete interpreter for each layer (except the lowest) is precisely the abstract interpreter for the next lower layer. Transitivity of the interpreter equivalence theorems guarantees that the lowest layer interpreter implements the highest layer interpreter. The choice of Micro-Gypsy as the source language used in this project opens the possibility of producing fully verified applications, using the existing verification tools for Micro-Gypsy.

Acknowledgments: The paper benefitted from the comments of Matt Kaufmann, Mike Smith, and the referees.

References

1. Department of Defense, DoD 5200.28-STD, *Trusted Computer Systems Evaluation Criteria,* December, 1985.

2. D.I. Good, R.L. Akers, L.M. Smith, ‘‘Report on Gypsy 2.05’’, Tech. report 48, Institute for Computing Science, The University of Texas at Austin, February 1986.

3. D.I. Good, B.L. Divito, M.K. Smith, ‘‘Using The Gypsy Methodology’’, Tech. report, Institute for Computing Science, University of Texas at Austin, June 1984.

4. Robert S. Boyer, J Strother Moore, *A Computational Logic,* Academic Press, New York, 1979.

5. Robert S. Boyer and J Strother Moore, *Metafunctions: Proving them Correct and Using them Efficiently as New Proof Procedures,* Academic Press, Inc., 1981.

6. Richard Cohen, ‘‘Proving Gypsy Programs’’, Tech. report, Institute for Computing Science, University of Texas at Austin, May 1986.

7. Warren A. Hunt, Jr., ‘‘FM8501: A Verified Microprocessor’’, Tech. report, Institute for Computing Science, University of Texas at Austin, December 1985.

8. Warren A. Hunt, Jr., ‘‘The Mechanical Verification of a Microprocessor Design’’, *Proc. of the IFIP Intl. Working Conference: From HDL Descriptions to Guaranteed Correct Circuit Design,* Grenoble, France, September 1986.

9. Shlomo Waser and Michael J. Flynn, *Introduction to Arithmetic for Digital Systems Designers,* Holt, Rinehart and Winston, 1982.

Table of Contents

List of Figures