# ♦ The Virtual Finite-State Machine Design and Implementation Paradigm

*Alan R. Flora-Holmquist, Edward Morton, James D. O'Grady, and Mark G. Staskauskas*

*This paper discusses the virtual finite-state machine (VFSM) design and implementation paradigm and our experience in introducing VFSM on software development projects for several Lucent Technologies products. VFSM, which allows software developers to specify the control behavior of a module as a finite-state machine, is supported by a toolset that automates many tasks associated with producing an implementation, including aspects of code generation, documentation, and testing. VFSM has been used in the design of more than 75 software modules, and its application has resulted in shorter development intervals and the elimination of defects prior to testing. In this paper, we present an overview of the VFSM design and implementation paradigms and the capabilities provided by the VFSM toolset. We also discuss the technical and nontechnical issues that have had an impact on the successful introduction of VFSM.*

## Introduction

Software engineering is a relatively new and immature discipline. One consequence of this immaturity is that software engineers, unlike their counterparts in the older branches of engineering, are unable to assure the correctness of software designs before they have been constructed. In contrast, civil engineers can construct mathematical models of bridges and buildings before they are built, and they can analyze the models to determine that the proposed structures have the desired properties. Any design errors detected during this analysis are much cheaper to fix than those that escape detection until after the concrete is poured.

Many design methodologies and computer-aided software engineering (CASE) tools have been used in attempts to provide a similar modeling and analysis capability for software engineers. However, the informality of many of these software models limits their effectiveness. Often, there is no well-defined notion of what it means for a computer program to be a correct implementation of a model, and it is often impossible to execute or analyze the model to determine that its behavior conforms with the designer's expectations. In the absence of such a capability, the correctness of a software design cannot be ascertained until after the code that implements it has been written. As a result, computer programs usually contain many more defects after construction than the products of other engineering disciplines.

The virtual finite-state machine (VFSM) design and implementation paradigm[1,2] is an approach to software modeling that addresses the limitations just described. VFSM allows the software developer to construct a *formal*, *executable* model of the control behavior of a software module. Because the finite-state machine model is formal, the control portion of the implementation can be generated automatically from the VFSM specification by means of a translation tool. The executable nature of a VFSM model enables a substantial amount of error checking to be performed on it early in the design process. The VFSM toolset includes both an interactive simulator that

**Panel 1. Abbreviations, Acronyms, and Terms**

ATM—automated teller machine
CASE—computer-aided software engineering
INAP—Intelligent Network Application Protocol
NCSL—noncommentary source lines
OFA—output function array
OSDS—Operating System for Distributed Switching
PIN—personal identification number
PSU—packet switching unit
SDL—Specification and Description Language
VFSM—virtual finite-state machine
VIR—virtual input register

allows the developer to exercise execution scenarios of a network of communicating VFSMs and a validator that attempts a systematic non-interactive search for errors in all possible scenarios. These tools complement each other well in that the simulator allows the user to examine selected scenarios in detail and to detect obvious errors, while in most cases the exhaustive search performed by the validator finds more subtle errors.

VFSM is specifically intended for modeling the control behavior of a software module. Control behavior is of primary importance in the design of many Lucent Technologies software products. The structure of the software for setting up a telephone call, for example, centers around the stimuli it receives from its environment—on- and off-hook signals, dialed digits, messages from other telephone switches—and the actions it takes in response to these stimuli. At a more detailed level, these actions may involve data manipulations, such as accesses to a subscriber database and the recording of billing information. By allowing stimuli and actions to be represented in an abstract form, the use of VFSM results in an implementation in which the control structure is kept strictly separate from the code devoted to data manipulations. This separation of concerns provides a bird's-eye view of the structure of a software module that allows its control behavior to be understood and modified independently of its data-related aspects.

Over the past several years, we have been involved in the introduction of VFSM in a number of Lucent organizations, including those responsible for the software of the 5ESS® switching system. VFSM has been enthusiastically received by software developers in these organizations, and as we describe in more detail below, the use of VFSM has resulted in software modules with fewer defects than those produced using traditional methods. A major factor in the popularity of VFSM is that the effort required to construct a VFSM specification is more than repaid by the automation provided by the VFSM toolset—a single textual representation of a VFSM serves as the basis for automatic code and documentation generation, as well as simulation and validation. Although the introduction of VFSM has been successful, it has by no means been easy. Our experience has been that technology transfer is not merely the last milestone in the development of a new method but rather a continuous process whose success requires a sustained effort to eliminate barriers to the use of the new technology.

Following this introduction, the "VFSM Design Paradigm" section presents an overview of the VFSM design paradigm, including a comparison with related design approaches and an example that shows how VFSM is used to build abstract models of the events and conditions that influence a module's control behavior and the actions it takes in response to them. The "VFSM Implementation Paradigm" section describes how these abstract entities are bound to concrete realizations by the VFSM implementation paradigm. The "VFSM Toolset" section presents the capabilities for design verification and code and documentation generation that are provided by the VFSM toolset. The "Experience with VFSM in Lucent Technologies" section discusses our experience with VFSM, including the history of its introduction, some data regarding the results of its use, and some examples of application areas in which it has been effectively used. The paper concludes with a summary of this discussion and our plans for future enhancements to VFSM.

## VFSM Design Paradigm

The fundamental VFSM design principle is one of abstract modeling and specification. The paradigm promotes the specification of control rather than pro-

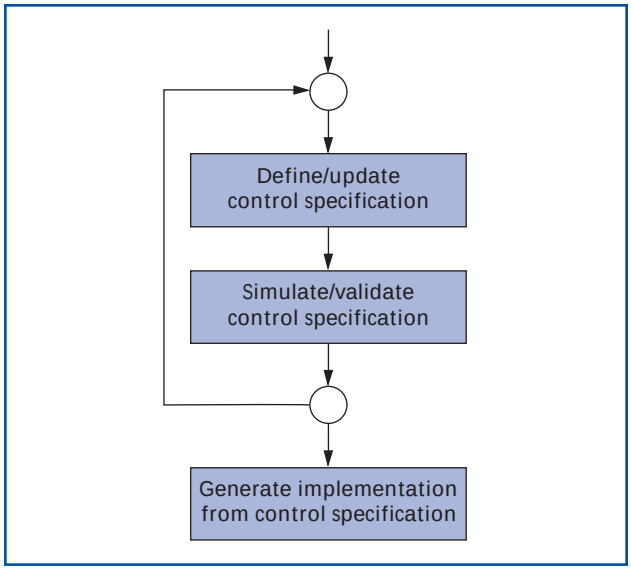


***Figure 1.***
***The VFSM design cycle.***

gramming, and it focuses the abstract behavioral model on *what* behaviors should be produced and *when* the behaviors should be produced, without ever mentioning *how* the behaviors should be implemented. The VFSM design cycle encourages an incremental, iterative approach to control specification and modeling. As **Figure 1** shows, a designer will define and then iteratively refine a control specification through simulation and validation.

After the control model is simulated and validated, the state machine implementation is generated from the control specification. The design paradigm assumes that the implementation paradigm will provide the necessary environmental interfaces and mappings to realize the abstract behavioral model, as **Figure 2** illustrates.

The VFSM specification language is specifically designed to reinforce the abstract behavioral model by maintaining a separation of control and data. Control refers to the rules governing behavior and data refers to the underlying implementation required to realize the behavioral model. A behavioral model for a given control domain is defined via the VFSM specification language. The specification is written in terms of an abstract name space (inputs, outputs, and states) that captures the implementation-independent characteristics of the control domain. The input space defines the set of abstract environmental conditions on which all control decisions are based. The output space defines the set of abstract behaviors the machine can exhibit and the state space defines a set of frames of reference for evaluating environmental conditions.

Supporting the specification are the input, output, and state dictionaries. Within the dictionaries, every input, output, and state name used in the specification is formally defined. A definition consists of both abstract and implementation information. The abstract information is sufficient to define the behavioral model, and it is also adequate for simulating and validating the behavioral model. The implementation information is used by the mapper generation tools to integrate and bind the abstractions to system-specific implementation constructs. As **Figure 3** illustrates, the control specification and dictionary sources define an abstract and implementation view of the behavioral model. The two views are input for design documentation, simulation, validation, and code generation.

Several unique features of the VFSM paradigm contribute to its expressive power and simplicity. The paradigm defines a virtual input register (VIR), which is used to store abstract inputs. The VIR provides a mechanism for implementing a separation between control and data. A condition represented by an abstract input is said to exist if that input is found in the VIR. Once it is present, a virtual input remains in the VIR until it is explicitly removed. During specification evaluation, Boolean expressions of virtual input names are used to test for the presence of subsets of inputs currently in the VIR. A VFSM state machine may generate output on state entry and exit, as well as in response to abstract conditions present in the VIR.

**Figure 4** illustrates three states of a VFSM specification for controlling an automated teller machine (ATM). The ATM VFSM waits for the customer to insert the appropriate card and then prompts for a personal identification number (PIN). If the customer does so within a certain amount of time and number of tries, a transaction may begin.

The ATM remains in its initial state, `SIDLE`, until the next state transition `NS:` becomes true. If the abstract condition `ICARDPRESENT` exists (the virtual input name `ICARDPRESENT` is contained in the VIR), the machine will transition to state `SGETPIN`. Note

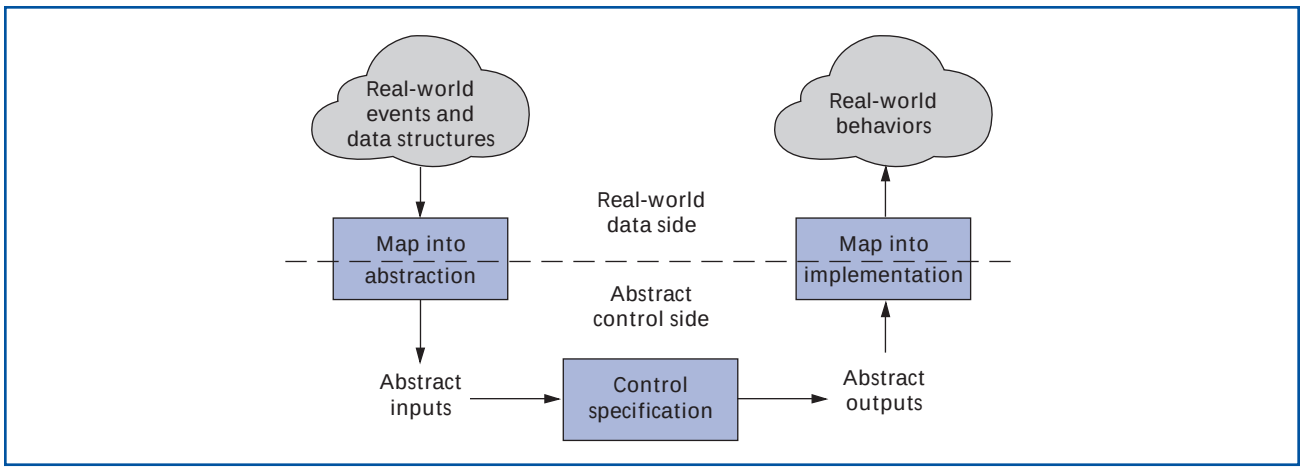


***Figure 2.***
***VFSM design paradigm overview.***

that the *physical* stimulus that denotes the presence of the ATM card, which may be a message, an interrupt, or a change in the value of a data structure, need not be specified at this level; only the abstract condition represented by the virtual input ICARDPRESENT is relevant to the control behavior. In general, a next state transition is specified as a Boolean combination of input names and then a state name. If the input expression is true given the current VIR contents, a transition to the given state occurs.

The SGETPIN state specifies that the input names IPINPRESENT and ICTIMEOUT should be cleared from the VIR (C:) on entry into the state. The state also specifies entry (E:) and exit (X:) actions, which are produced whenever there is a transition into or out of the state, respectively. The entry action (E: OPINPROMPT, OSTARTCTIMER) specifies that the customer is to be prompted for a PIN and a guard timer started for a reply. The exit action (X: OSTOPCTIMER) specifies that the guard timer is stopped when the machine transitions out of the SGETPIN state. If the IPINPRESENT input condition becomes true, a transition to state SCHKPIN will occur. If the ICTIMEOUT input condition becomes true, a transition to state SBYEBYE will occur (this state is not shown in Figure 4). Until one of these conditions becomes true, the machine remains in the SGETPIN state.

The SCHKPIN state specifies an entry action to validate the customer PIN (E: OPINVALREQ). Following entry action production, the machine should proceed (M: PROCEED) to evaluate the input action section (IA:). The input action section specifies that the output OBADPINREPLY should be produced if the input condition IPINNOTOK is contained in the VIR, in which case the customer will be notified. In general, input actions are specified as a Boolean combination of input names and then a list of outputs. If the input expression is true given the current VIR contents, the list of outputs is produced. The VFSM will enter state SBTRANS (not shown) if the customer entered a correct PIN, in which case the transaction can commence. If the PIN is incorrect but the customer has not exhausted the number of possible retries (IPINRETRYOK), the VFSM returns to state SCHKPIN and the customer will again be prompted to enter the PIN. Otherwise, the customer has made the maximum number of tries, so the VFSM enters the terminal state SBYEBYE.

As noted earlier, the name space dictionaries support the control specification. **Figure 5** illustrates segments of the input, output, and state dictionaries for the ATM VFSM. One or more attributes compose a dictionary record, and the attributes are given as keyword value pairs. The example dictionary records all begin with $name and $desc attributes. The input dictionary segment defines the input IPINRETRYOK, and the segment's $desc attribute gives a brief English description of its meaning. The virtual input class ($class) attribute denotes that this input

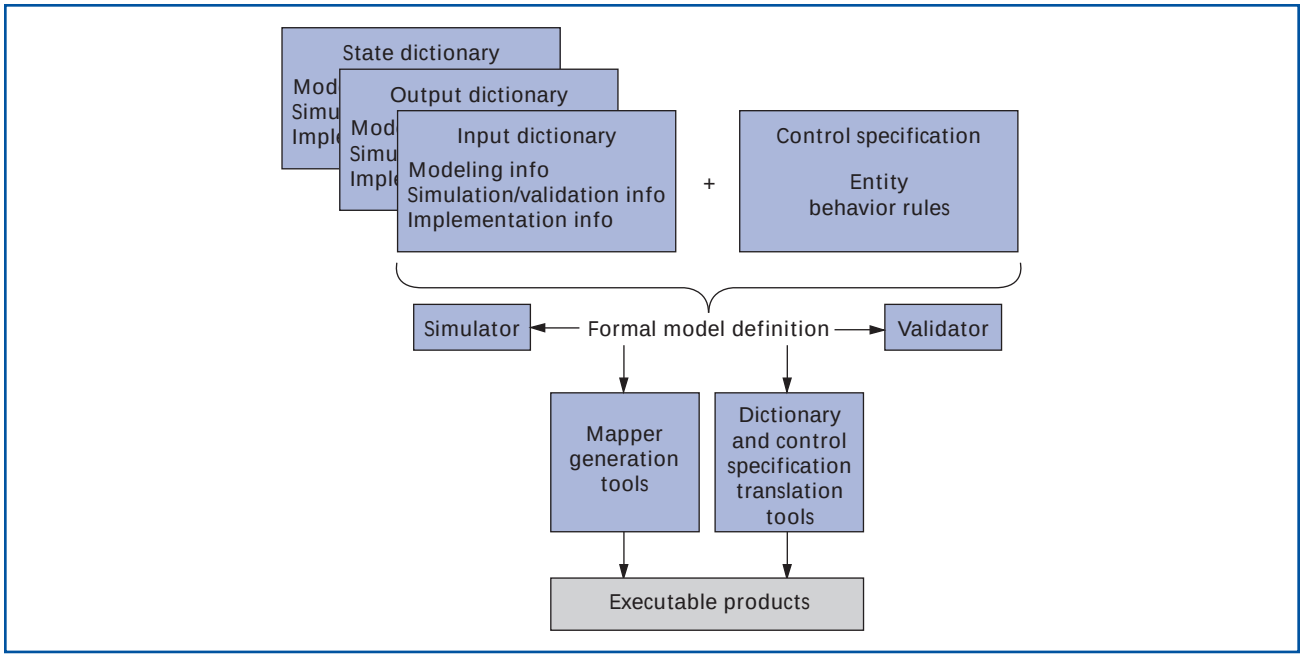


***Figure 3.***
***Dictionaries and specification in VFSM design.***

belongs to a set of related virtual inputs, one of which (at most) may be present in the VIR at a time. The virtual event (`$event`) attribute defines the external stimuli that may cause this input to be inserted into the VIR—in this case, the event `EPINVALFAIL`. The `$how` attribute contains a logical condition, expressed in C, that indicates when the input is to be inserted into the VIR when one of the events in its `$event` attribute is received. The `$vercond` attribute is an abstract version of the `$how` attribute that is used during simulation and validation. The `$vercond` attribute is described in more detail in the "VFSM Verification Tools" subsection.

In addition to `$name` and `$desc` attributes, the output dictionary entry contains the name of a function to be called to realize the given abstract output (`$fcn`) and the code segment to be executed for the given abstract output name (`$how`). The output dictionary record also defines a machine event (`$me`) attribute which, like the `$vercond` attribute in the input dictionary, is an abstract representation of the `$how` attribute that is used during simulation and validation.

## Related Work

VFSM is one of several tool-supported methodologies based on extended finite-state machines that recently have attained substantial industrial use. Other examples include Statecharts[3] and the Specification and Description Language (SDL)[4].

VFSM differs from these methodologies in two major ways. The first difference is that VFSM is tailored for use by software developers during low-level design while the other methodologies are used primarily by system architects earlier in the design process. The advantage of using VFSM so late in the design process is that the gap from low-level design to code is small enough to be bridged automatically by the VFSM translator. Subsequent changes to the control behavior of the module are made to the VFSM specification rather than to the implementation, and the translator assures that the two remain consistent. By contrast, CASE methodologies used earlier in design often result in an item of documentation that inevitably becomes out of date with respect to the implementation.

The second major difference is that Statecharts and SDL provide design notations that are far more expressive than VFSM. SDL, for example, offers powerful object-oriented data modeling capabilities. In contrast, VFSM extends the basic FSM model with lit-

```
SIDLE {
  NS: ICARDPRESENT > SGETPIN;
}

SGETPIN {
  C:   IPINPRESENT, ICTIMEOUT;
  E:   OPINPROMPT, OSTARTCTIMER;
  X:   OSTOPCTIMER;
  NS:  IPINPRESENT > SCHKPIN;
       ICTIMEOUT > SBYEBYE;
}

SCHKPIN {
  M:   PROCEED;
  E:   OPINVALREQ;
  IA:  IPINNOTOK ? OBADPINREPLY;
  NS:  IPINOK > SBTRANS;
       IPINRETRYOK > SGETPIN;
       TRUE > SBYEBYE;
}
```

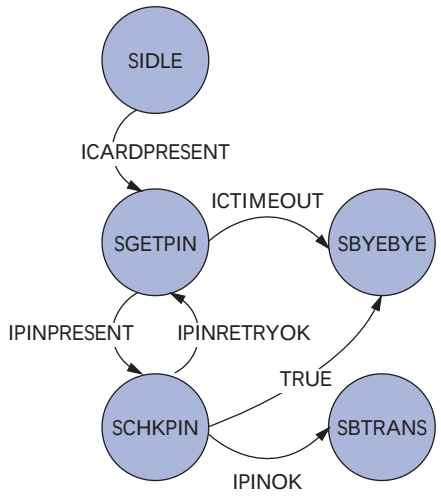


***Figure 4.***
***Example VFSM specification and state transition graph.***

tle more than Boolean variables. The danger of highly expressive design notations is that they can be incorrectly used as programming languages, resulting in an overly detailed design that is too close to an implementation and cannot easily be simulated or validated. The VFSM notation, by virtue of its simplicity, forces the developer to produce an abstract specification of the control behavior of a module, and VFSM has proven to be sufficiently expressive to capture the vast majority of applications we have encountered.

## VFSM Implementation Paradigm

This component of VFSM allows the abstract behavioral model defined during design to be integrated without modification into the target system environment. The implementation paradigm makes no assumptions regarding the run-time environment of the host system. Intermachine communication and timing services are assumed to be provided by the host environment. As **Figure 6** shows, a VFSM implementation comprises four implementation modules: interface function, input mapper, output mappers, and specification interpreter. The roles, responsibilities, and interfaces of each module are specified by the paradigm. The specification interpreter is provided as part of the system software. The interface function must be coded by the application developer, and the input and output mappers are generated from the input and output dictionary contents, respectively.

The role of the interface function is to shield the VFSM implementation from the real-world stimuli. The interface function is responsible for mapping real-world stimuli into the virtual event space, invoking the input mapper and interpreter. In the ATM example, the interface function would detect the physical stimulus (message, interrupt, or change in data structure value) denoting the presence of the customer's ATM card, conveying the stimulus via the input mapper to the VFSM specification. The input mapper role is that of VIR manager. It is responsible for mapping virtual events into one or more virtual inputs, which are stored in the VIR. It is the role of the output mapping functions to realize the state machine behaviors. Output mapping functions are responsible for mapping virtual outputs into real world behaviors. The specification interpreter executes a VFSM specification using the current state and VIR contents of the VFSM to be executed and the encoded version of the VFSM specification produced by the VFSM translator.

The operations below characterize a typical execution scenario of the functions shown in Figure 6.

```
Input Dictionary Segment

$name    IPINRETRYOK
$desc    It is valid for the customer to re-enter the PIN
$class   retrystat
$event   EPINVALFAIL
$how     COND ptr2app->num_tries < ptr2app->max_tries
$vercond num_tries < max_tries

Output Dictionary Segment

$name    OPINVALREQ
$desc    Report customer id validation request
$me      num_tries++;
         EPINVALSUCC | | EPINVALFAIL;
$fcn     atmomisc
$how     ptr2app->num_tries++;
         if (ptr2app->pin_entered == ptr2app->pin_required)
         {
              atmimap (EPINVALSUCC, ptr2instance);
         }
         else
         {
              atmimap (EPINVALFAIL, ptr2instance);
         {

State Dictionary Segment

$name    SBTRANS
$desc    Begin customer transaction selection
```

***Figure 5.***
***Dictionary example.***

- Execution begins when the interface function receives an external stimulus—for example, a message, timer expiration, or interrupt.
- The interface function maps the external stimulus into a virtual event within the event space of the given VFSM and invokes the input mapper with the virtual event as an argument. Based on the virtual event and the values of implementation data structures, the input mapper may either insert inputs into or delete them from the VIR.
- The interface function next invokes the specification interpreter. Given the current state and VIR contents, the interpreter evaluates the VFSM control specification, which may result in state changes and the production of virtual outputs.
- When a virtual output is produced, the interpreter invokes the C function specified for that output by the `$fcn` attribute in the output dictionary. This function performs the necessary operations to realize the behavior associated with the virtual output. As Figure 6 shows, an output function may produce "feedback" events by invoking the input mapper, which can result in an instantaneous change of the VIR while the VFSM is executing.
- Execution of the VFSM continues until a state is reached from which no next-state transitions are possible given the VIR contents. At this point, the specification interpreter returns control to the interface function. Because the VFSM will remain in this "quiescent" configuration until its VIR changes, there is no need to execute the VFSM again until the interface function receives another external stimulus.

## VFSM Toolset

A number of tools have been developed to support software designers who use the VFSM paradigm. Code generation tools translate the specifications for a design into code, which will be compiled into a product. Verification tools allow developers to exercise their VFSM designs in a desktop UNIX* environment. This procedure saves the costs associated with testing

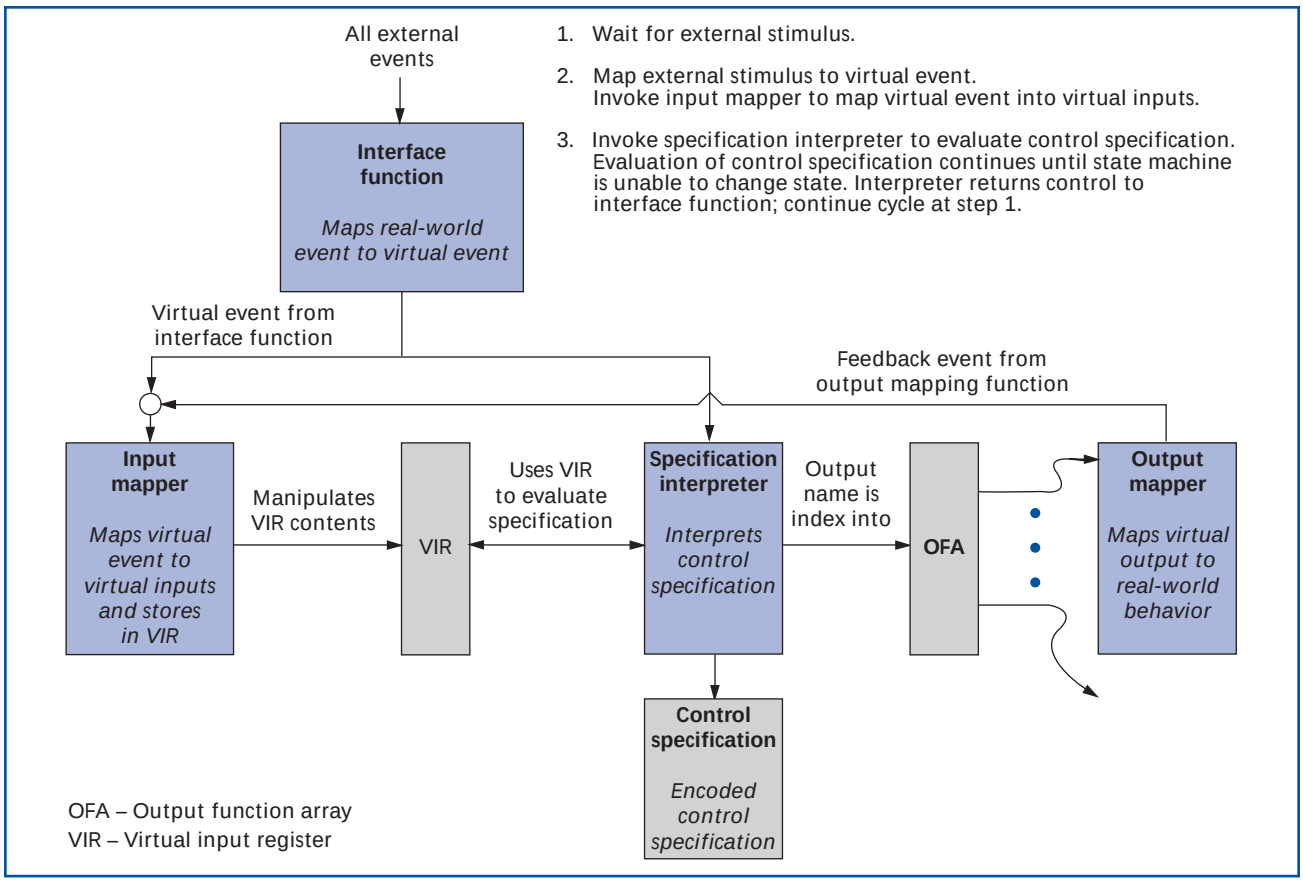


***Figure 6.***
***General VFSM implementation structure.***

fully realized implementations of flawed designs on actual switching hardware. Documentation generation tools create the required design documentation from the specification.

### Code Generation Tools

Code generators serve two purposes in the VFSM environment. First, they ensure that the design described in the specification and dictionaries is exactly and precisely implemented by the code. Second, they free users from performing repetitive coding and testing tasks, allowing them to concentrate on the essentials of the design, which are generally complex. VFSM has two major code generators: the translator, which encodes the specification into C, and the mapper generators, which produce the mapping code functions based on the dictionaries.

**VFSM translator.** This program translates information from the specifications and dictionaries into C code, which is then compiled into either the simulator/validator or the target product itself. The relationships between state transitions, output productions, and the contents of the VIR as defined by the specification are encoded for later interpretation by the VFSM specification interpreter. Pointers to each output function defined by the `$fcn` attributes in the output dictionary are stored in the output function array (OFA) structure so that they can be invoked by the VFSM specification interpreter during the execution of a VFSM.

To execute a VFSM specification, the specification interpreter will use the encoded data about the specification to determine what to do. Because the encoded data must be interpreted, it is not possible to get the same efficiencies from VFSM as you can from compiled languages. Several optimizations[5,6] have been applied to the encoding to make execution as efficient as possible, making VFSM acceptable for all but the most real-time critical applications. The projects that

have used VFSM to reengineer existing code have benefited from many savings because of the simple execution model and, thus, simpler interactions that seem to counterbalance the loss of performance.

The VFSM translator reports two kinds of errors and warnings. Of course, we must report problems that cause translation to fail and indications that the design is incomplete. In addition, we report a variety of semantic errors and warnings. These indicate that the design seems to contain violations of the VFSM paradigm. For instance, because only one input from a class can be in the VIR at a time, a logical AND condition can never be true if it contains two members of the same class.

**VFSM mapper code generation.** In VFSM, we use dictionaries to store both the abstract and real-world information associated with virtual names. The user must always supply the application-specific code that realizes specific abstract names. However, given the correct information stored on a per-virtual-name basis in the dictionaries, we can generate the code that controls the mapping between the abstract names and the real-world representations. Generating the control code in this way saves the user time and effort, and it avoids the possibility of user error while designing or coding this control.

The input mapper generation algorithm uses the input dictionary attributes `$name`, `$class`, `$event`, and `$how`. By definition, the `$how` attribute contains one of three reserved keywords for every virtual input. The keyword *DIRECT* indicates that the given input name is mapped unconditionally; only one member of a virtual input class may be defined as *DIRECT* for a given event. An input whose keyword is *COND* is realized by a logical condition in the data model. This condition is specified via a C expression in the `$how` attribute for the input. If the input has the keyword *DEFAULT*, it is mapped when none of the conditions in the `$how` attribute for other members of its class with attribute *COND* is true.

The output mapping functions are defined by the designer in the `$fcn` field of the output dictionary. Thus, many outputs may share the same output mapper (for example, for grouping-related behaviors). The output mapper generation algorithm uses the output dictionary attributes `$name`, `$fcn`, and `$how`. The `$how` attribute defines the code segment to be executed when the given abstract output action is to occur.

**Figure 7** provides examples of portions of the generated mappers for the ATM example in Figures 4 and 5. These examples illustrate how the mappers are assembled from the information in the dictionaries. Within each input and output mapper, a pointer provides the user with access to an application-specific data structure, which contains any data that can influence control behavior. The top of Figure 7 gives the output function for the virtual output `OPINVALREQ` in Figure 5. The name of the function is derived from the `$fcn` attribute of this output, and the function takes two arguments: a virtual-output identifier and a data pointer. The body of the output function is a C `switch` statement that contains a case for each virtual output that has `atmomisc` as its `$fcn` attribute. This case contains the code fragment from the `$how` attribute for the output that realizes its behavior.

The bottom of Figure 7 shows an input mapper segment for the ATM example. Like the output functions, the input mapper takes two arguments: a virtual-event identifier and a data pointer. The body of the input mapper is a `switch` statement that contains a case for each virtual event. This case contains a code fragment for each virtual input that has the event in its `$event` attribute. In Figure 7, we assume that the event `EPINVALFAIL` appears in the `$event` attribute of three virtual inputs: `IPINRETRYOK`, which is shown in Figure 5 and has the keyword *COND* in its `$how` attribute, and `IPINRETRYNOTOK` and `IPINNOTOK`, which have keywords of *DEFAULT* and *DIRECT*, respectively. `IPINRETRYOK` and `IPINRETRYNOTOK` are in the same virtual input class while `IPINNOTOK` is in a different class. The example input mapper illustrates the use of class and keyword information to generate mapping code. `IPINNOTOK` is unconditionally inserted into the VIR, `IPINRETRYOK` is inserted only if its logical condition holds, and `IPINRETRYNOTOK` is inserted by default if the condition of the only other member of its class, `IPINRETRYOK`, does not hold.

```
atmomisc(output, ptr2instance) /* OUTPUT FUNCTION */
ATMOUTPUTS output;
VFSMINSTENV *ptr2instance;
{
        /* Switch on output and realize the specified virtual output */
        switch  (  output  )  {

        .     .     .     .     .     .     .
        .     .     .     .     .     .     .

        case OPINVALREQ:
              { /* Start OPINVALREQ scope */
              ptr2app->num_tries++;
              if (ptr2app->pin_entered == ptr2app->pin_required)
              {
                    atmimap (EPINVALSUCC, ptr2instance);
              }
              else
              {
                    atmimap (EPINVALFAIL, PTR2INSTANCE);
              }
              }  /* End OPINVALREQ scope */
              break;  /* End case OPINVALREQ */

        .     .     .     .     .     .     .
        .     .     .     .     .     .     .

}  /* End of output function atmomisc */

atmimap (event, ptr2instance)     /* INPUT MAPPER */
ATMEVENTS event;
VFSMINSTENV *ptr2instance;
{
        /* Switch on event and derive virtual inputs */
        switch  (  event  )  {

        .     .     .     .     .     .     .
        .     .     .     .     .     .     .

        case EPINVALFAIL:
              map_array [map_index++] = IPINNOTOK;

              if  ( ptr2app->num_tries < ptr2app->max_tries  )
              {
                    map_array [map_index++] = IPINRETRYOK;
              }
              else
              {
                    map_array [map_index++] = IPINRETRYNOTOK;
              }
              break;  /* End of case for event EPINVALFAIL */

        .     .     .     .     .     .     .
        .     .     .     .     .     .     .

}  /* End of input mapper */
```

***Figure 7.***
***Output and input mapper segments.***

### VFSM Verification Tools

The VFSM toolset includes two tools for pre-implementation verification of the behavior of a state machine: the VFSM simulator and the VFSM validator[7]. The simulator is highly interactive, relying on the user to direct how the machines should execute, while the validator performs exhaustive traversal of various execution scenarios.

Both the simulator and validator support networks of communicating VFSMs, allowing the interaction among VFSMs in the network to be exercised along with the execution of the individual VFSMs. In a

```
Executing VFSM atm

First, we map the external event, which is waiting for the ATM machine.
There is only one choice, so the input IPINPRESENT can be directly mapped.

Mapping Event atm.ECUSTOMERPIN

instance atm VIR = {  IPINPRESENT(2)  }

The simulator is executing the ATM machine given the VIR contents above.
The input causes a transition to a new state. Because we are leaving
SGETPIN, we perform the two exit actions, one of which stops the ECTIMEOUT
timer.

<VFSM>  INSTANCE atm ENTERED SPECIFICATION EXECUTOR at TIME 3
<VFSM>  Machine state: atm'SGETPIN(1)
Exit:   atm'OSTOPCTIMER(6)  Stopping  Timer atm.ECTIMEOUT

In the new state, the $me for entry action OPINVALREQ is
$me     num_tries++;
        EPINVALSUCC | | EPINVALFAIL;
The user must choose one of these choices. The "s" choice below
would skip this choice entirely. The events to be chosen are both
feedback events and will be mapped immediately.

<VFSM>  Changing state to atm'SCHKPIN(2)
Entry:  atm'OPINVALREQ(2) Output->Event Choice for output atm.OPINVALREQ(2)
Choice #          What?
1:            EPINVALSUCC
2:            EPINVALFAIL

Enter Choice: [1-2,s] >>    2
Mapping Feedback Event atm.EPINVALFAIL

instance atm VIR = {  IPINPRESENT(2), IPINNOTOK(4)  }

instance atm VIR = {  IPINPRESENT(2), IPINNOTOK(4), IPINRETRYOK(5)  }

Proceeding to evaluate input actions
Output: atm'OBADPINREPLY(4)
Sending Event cust.EBADPINRPT

We transition to a new state, send the EPINPROMPT request to the cust
machine, which completes the processing on the ATM machine.
<VFSM>  Changing state to  atm'SGETPIN(1)

<VFSM>  Removing inputs:    IPINPRESENT(2)    ICTIMEOUT(7)
Entry:  atm'OPINPROMPT(1)
Sending Event cust.EPINPROMPT
atm'OSTARTCTIMER(5)
```

***Figure 8.***
***Example simulation session.***

VFSM-generated implementation, inter-VFSM communication takes place via system calls to send and receive messages that appear in the interface function and output functions. The simulator and validator, however, do not support the inclusion of C code, so we have provided a set of communication and timing primitives, known as the *verification environment*, that are based loosely on the Operating System for Distributed Switching (OSDS) used in the 5ESS switch. VFSMs communicate in the verification environment by sending each other virtual events. Each VFSM has an event queue on which it receives events from other VFSMs. When it executes a VFSM, the verification environment removes the event at the head of its event queue, performs input mapping for the event, and then invokes the specification interpreter for the VFSM.

The communication and timing primitives are invoked in the `$me` attribute in the output dictionary and the `$vercond` attribute in the input dictionary. These attributes are known as *mapping abstractions* because they allow an abstract specification of those aspects of the input mapper and output functions that can have an impact on control behavior. Examples of mapping abstractions appear in the dictionaries in Figure 5. The code fragment in the `$how` attribute in the output dictionary compares the customer's stored PIN to the one that was just entered to see if the latter is valid. Because the database of customer PINs is not included in the VFSM model, the

$me attribute that models this code fragment simply lists the possible outcomes of the PIN validation. Such a list of possibilities separated by a logical OR symbol (||) is referred to as a *choice point*. Note that the $me construct can also reference simple scalar variables (num_tries in this example), and it could also include timer start and stop operations and the sending of virtual events to other VFSMs.

VFSMs often communicate with other entities in their environment. We have found that use of the simulator and validator is far more effective if VFSMs are constructed that model the behavior of these external communication partners. In the ATM example, it is useful to define such an *environment VFSM*, known as "cust," that represents the behavior of the customer using the ATM. This VFSM begins the interaction by sending the ECARDPRESENT event to the ATM machine, and it can then optionally generate an event representing the PIN when prompted by the ATM VFSM.

**VFSM simulator.** This highly interactive tool allows the designer to execute a design model and closely observe the results. The simulator allows effective prototyping of specifications in various stages of development, from a completed specification to one that has not been well fleshed out. The specification of behaviors in the dictionaries is reflected in the verification environment. Thus, little additional effort is required to simulate a VFSM or network of VFSMs.

When the simulation session is at a quiescent point, the user can select an enabled VFSM (that is, one in which the event queue is not empty) to execute next. The user can also choose to cause the expiration of a running timer. When a choice point is encountered during execution of a VFSM, the simulator suspends its execution and prompts the user to make a choice using a simple menu. Once the choice is made, the simulator resumes execution of the VFSM, which continues until another choice point is encountered or the VFSM execution completes. Completion occurs when a VFSM state is reached from which no next-state transitions are possible. As the VFSM is executing, the simulator displays the VFSM states entered, changes in VIR contents, virtual outputs produced, timer operations performed, and

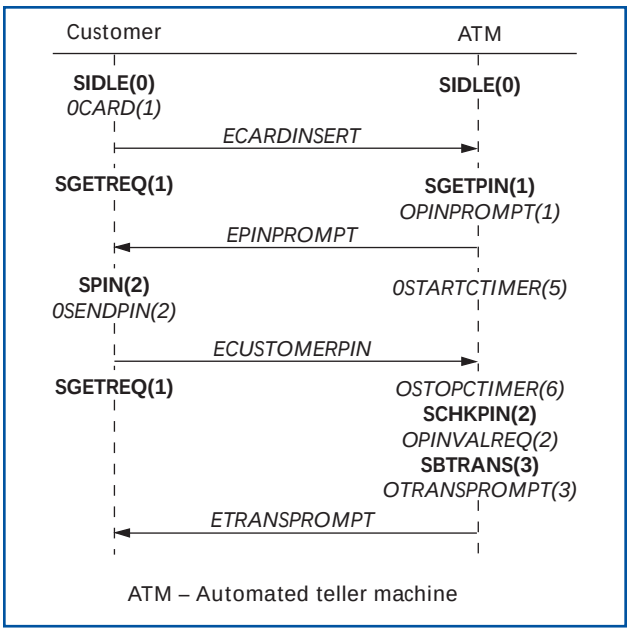


***Figure 9.***
***Seqflow message sequence chart output for the ATM example.***

virtual events that are fed back or sent to other VFSMs. Examination of this output allows the developer to determine that the VFSM design has the intended behavior.

**Figure 8** presents a sample simulation session for the ATM example. We have added the environment machine to simulate the actions of the customer at the ATM machine. The event ECUSTOMERPIN has been sent from the customer VFSM. In the figure, simulator output is shown as text `like this` while comments on the text are shown **`like this`**.

During execution, the simulator produces *message sequence charts*, optionally showing which states have been hit, which outputs have been produced, which inputs have been mapped, and which variables have been set. They also show the communications via pairs of instances. A local tool, seqflow, is used to display the results. In this way, a VFSM design can be used to document its own behavior accurately. In a non-VFSM design, such sequence charts are merely assertions about what the design should do.

Message sequence charts are a powerful tool for illustrating scenarios for a design. The seqflow tool was in common use in 5ESS switch development for many years before VFSM began generating seqflow input. By using this tool, we produced output our users were

familiar with and had experience generating themselves, and the message sequence charts we produce automatically during simulation can become part of the required design documents. **Figure 9** shows the output of seqflow for a simulation session of the ATM VFSM. The two columns show the customer environment VFSM and the ATM VFSM. Arrows between columns denote the sending of virtual events between the VFSMs. The labels in bold type in each column denote the VFSM states entered while the labels in italicized type represent the virtual outputs produced.

**VFSM validator.** This tool exhaustively exercises possible execution scenarios of a network of communicating VFSMs. It checks for errors in the concurrent interaction of the VFSMs, such as deadlock, in which some VFSM is waiting to receive a virtual event but all the event queues in the network are empty. Whenever such an error is found, the validator outputs a message sequence chart, such as that shown in Figure 9, illustrating the sequence of events that led to the error.

In the VFSM simulator, the user interactively selects which VFSM instance will execute next. In addition, the user can provide a selection whenever a choice point is encountered during the execution of a VFSM. The VFSM validator is different in that it operates in batch mode and with no user intervention. Whenever there is a choice of which VFSM to execute next or which choice selection is to be made, the validator systematically tries all possibilities. Therefore, one can visualize the execution of the validator as a tree that branches whenever either kind of choice must be made. A given simulation session explores only one path through this tree, while the validator attempts to explore all possible paths.

The VFSM validator searches for errors in a network of communicating VFSMs by constructing the *global state graph* of the network. The global state is a vector containing the state of each VFSM in the system that includes all information relevant to its communication behavior. The state of a VFSM includes the VFSM state, VIR, and event queue. Beginning from the initial global state, the validator generates possible successors of each global state by executing each VFSM in the network. Because the number of possible global states is finite, the generation of global states in this way eventually terminates. The global state completely determines the scenarios that can occur in the future, so it is not necessary to explore successors of a global state that was generated at an earlier point in the validation.

In most implementations of validation algorithms, a hash table is used to track which global states have already been generated. However, because the number of global states is an exponential function of the global state size, the hash table will overflow available memory when a large application problem is validated. The VFSM validator employs the *supertrace algorithm*,[8] which avoids explicit storage of global states in memory. Therefore, it can handle larger problems (the drawback of the algorithm is that it may not explore all global states so that some errors can be missed).

We developed a prototype version of the validator in 1993. To convince ourselves that the validator had enough features to be useful in real 5ESS switch applications, we validated three VFSM applications that had already completed testing in the lab. To our surprise, the validator found bugs in all three applications that escaped detection during simulation and testing. This clearly illustrates the power of formal validation as a technique for finding errors. The VFSM validator has since been used by more than 25 software developers, representing one of the first instances in which formal validation has been used on a wide scale in an industrial software development organization. We have found that the simplicity of the VFSM notation allows an extremely compact representation of control behavior that keeps the number of global states explored by the validator to an acceptable level for most applications.

### Documentation Generation Tools

The VFSM toolset includes tools that can create a design documentation package for a given VFSM specification and dictionaries. The package includes a state transition diagram, a descriptive form of the behavioral model, and formatted dictionaries. When combining the above output with the per-scenario output from simulation runs, the only documentation left for the user to write is a few paragraphs describing external interfaces, partitioning, and general strategy. In this

```
STATE:                  SGETPIN
DESCRIPTION:            Get the customer pin
PREDECESSORS:           SIDLE, SCHKPIN

STATE EXECUTION MODE:

          On entry to this state, DO NOT evaluate the Input Action Section.

ENTRY CLEAR:

          (IPINPRESENT) Customer has entered PIN {class: custact} {event(s): ECUSTOMERPIN}
     and  (ICTIMEOUT) Timeout occurred while waiting for cust reply {class: custtimer} {event(s): ECTIMEOUT}

ENTRY OUTPUT:

          (OPINPROMPT) Prompt for customer PIN {event(s): cust.EPINPROMPT}
     and  (OSTARTCTIMER) Start the cust reply watchdog timer {event(s): TSTART(ECTIMEOUT)}

EXIT OUTPUT:

          (OSTOPCTIMER) Stop the cust reply watchdog timer {event(s): TSTOP(ECTIMEOUT)}

NEXT STATE TRANSITION SECTION:

     if                   (IPINPRESENT) Customer has entered PIN {class: custact} {event(s): ECUSTOMERPIN}
     then go to           (SCHKPIN) Check the customer PIN

     if                   (ICTIMEOUT) Timeout occurred while waiting for cust reply {class: custtimer} {event(s): ECTIMEOUT}
     then go to           (SBYEBYE) End of transaction
```

***Figure 10.***
***Generated descriptive behavioral model segment.***

way, the VFSM toolset generates the majority of the required design documentation. The user is thereby relieved of much writing of documentation by hand and is assured that the documentation is consistent with the implementation because both are derived from the same VFSM specification and dictionary source files.

The descriptive form of the behavioral model consists of one or more pages for each VFSM state followed by tables of cross-reference information on virtual names. **Figure 10** shows the descriptive form for the state `SGETPIN` in the ATM example. Notice that in addition to the information that can be obtained just by looking at the raw state specification, the descriptive form includes:

- The name of the state(s) from which the current state can be entered (derived from the specification),
- The one-line description of each virtual name (from the `$desc` fields in the dictionaries),
- The events that may be generated for the given virtual outputs (from the `$me` fields in the output dictionary),
- The events that lead to the given inputs being mapped (from the `$event` fields in the input dictionary), and
- The classes of the given inputs (from the `$class` fields in the input dictionary).

## Experience with VFSM in Lucent Technologies

The first of the following two subsections briefly reviews the history of the development and introduction of VFSM and discusses the issues surrounding VFSM technology transfer.[9] The second subsection discusses several areas of application of VFSM technology.

### Technology Transfer

After developing a successful prototype of the VFSM toolset, we shared VFSM project results and vision with the user community. Our anticipation of a warm welcome for the new technology contrasted sharply with the chilly reception given our proposal by potential users who raised a number of objections to VFSM. Much of the present 5ESS switch development effort involves maintenance, reengineering, and interfacing with existing code. Therefore, many individuals thought introducing a new methodology like VFSM would be feasible only on projects involving development of completely new software. Some developers would not consider adopting VFSM without quantita-

tive evidence that its use would yield the claimed reductions in development intervals and defect rates.

The technology transfer ice breaker finally appeared in the form of a project whose schedule was considered unattainable given the available software development environment. Working with the lead engineer on a few preliminary estimates demonstrated that VFSM technology could provide the productivity required to meet production schedules. VFSM became part of the critical path of the user community's success. Our hope was that this success would validate VFSM technology.

The project was delivered on time, and its success was attributed to the use of several enabling technologies, including VFSM. This pilot project won over a small group of enthusiastic VFSM converts. However, it became clear that seamless integration of VFSM into the embedded software engineering infrastructure was required before it would be ready for a larger user community within the 5ESS switch development organization. To be suitable for use in development projects, VFSM had to become an accepted part of the official design process, which required precise documentation of the steps required in its application. It was necessary to allow VFSM specifications to be put under configuration control and for the VFSM translator to be integrated into the load-building process that compiles and links all 5ESS switch source code. An extensive four-day training course was developed that covered behavioral modeling concepts, a hands-on introduction to the VFSM simulation and validation environments, and the integration of a VFSM-based design into a C-based implementation.

These efforts, along with organizational support for the necessary process changes, led to extensive use of the VFSM methodology and toolset in the 5ESS switch environment. User experience with VFSM has now become the guiding influence as we endeavor to provide ever increasing technology leverage over the design, coding, and testing tasks. VFSM technology transfer continues today, some five years after the first VFSM application. Our overarching goal is to make application of the VFSM methodology and toolset as smooth and painless as possible. As part of the technology transfer process, one of the authors of this paper participates as a consultant, reviewer, and inspector for each and every VFSM-based design. Our measuring sticks for gauging technology transfer progress are repeat users and referrals. Furthermore, several experienced VFSM designers have transferred to other development organizations and persuaded their new colleagues to use VFSM.

We now have enough experience with VFSM that it is becoming possible to obtain quantitative evidence regarding its effectiveness. Data were gathered on the number of faults detected during testing in the first 16 VFSM-designed 5ESS switch modules. These data were compared against the number of faults predicted for these modules using a model based on data from previous (non-VFSM) modules. The number of defects in the VFSM modules was found to be as much as 50% less than predicted. An informal poll of VFSM users suggests that the implementation of VFSM leads to interval reductions of an estimated 15%. We are now collecting more precise data to confirm this estimate.

To date, 630 people have been trained in VFSM, and the distribution of VFSM use by area was determined to be:

- 46% hardware maintenance/fault recovery,
- 41% signaling/customer features,
- 7% peripheral hardware control, and
- 6% trunk and line maintenance.

### Example Application Areas

The applications that follow are some of those in which VFSM has been used. The descriptions were provided by the engineers who worked on the designs.

- *Path control.* When new types of ports were introduced in the 5ESS switch (that is, hardware with associated functionality), approximately 3,000 lines of code had to be written to facilitate setting up and tearing down the connections (paths) between them. We reengineered this domain so that high-level control was specified using a small VFSM. Now, when new ports are introduced, it takes fewer than 100 noncommentary source lines (NCSL) to facilitate path control. This reengineering was completed in less time than it would have taken to write the 3,000

NCSL that would have been required by the existing architecture.

- *Intelligent networks*. The European Telecommunication Standard of the Core Intelligent Network Application Protocol (INAP) was developed on the 5ESS switch using VFSM, which was extremely effective. Extensive simulation was done during the design phase of this project resulting in a very high-quality final product. In fact, according to project quality data, the projected faults per thousand NCSL for this development is significantly lower than the projected mean. Furthermore, updates and additions to the protocol have been done easily. This project—more than 10,000 lines of code in length—resulted in a very large state machine (the state machine contains nearly 200 states). Even though this was a large project, the VFSM designers believe the implementation is structured in a way that makes it easy to understand because it was designed and carried out using VFSM.
- *Packet switching channel control*. The 5ESS switch includes a local area network called the packet switching unit (PSU). Protocol handlers are plugged into the PSU to provide various features like ISDN and common channel signaling. For wide band paths, it was necessary to interconnect several PSUs with the resulting channel being controlled by a network of 16 VFSMs, which handle such complicated interactions as those involving alarms, craft requests, and protection switching. Comparable developments have taken much longer and had many more bugs. The consensus among members of the project team is that using VFSMs to partition the functionality, as well as using the simulator and validator to test before reaching the lab or writing C code, contributed significantly to the project's success. In addition, handing out the design of the VFSMs, virtual input mapping, and virtual output realization to individual developers contributed greatly to the smooth operation and integration of the entire team.

These experiences reinforce our belief that the use of VFSM leads to designs that have fewer faults, that are completed more rapidly, and that are easier to maintain.

## Conclusion

We presented the VFSM design and implementation paradigm, the toolset we constructed to support its use, and our experience in transferring VFSM technology to several Lucent software development organizations. VFSM effectively raises the level of abstraction available to the programmer by allowing a high-level specification of the control behavior of a software module from which the implementation and documentation can be generated automatically. The executable nature of the specification makes possible a thorough analysis with the VFSM simulator and validator. This analysis allows developers to detect and correct errors while working at their desks, which is far more effective than debugging code on the actual switch hardware where it is difficult to re-create error scenarios and where availability of the hardware testbed is limited. The automation provided by the VFSM toolset is perhaps the most important factor in the widespread acceptance and popularity of VFSM in the development communities in which it has been introduced. Our efforts to adapt VFSM technology to the constraints of the existing development environment were also crucial.

In the future, the use of multiple cooperating VFSMs within a process (a VFSM community) will become more common as VFSM technology is applied to larger design and implementation problems. A VFSM community architecture is being defined to address the needs of larger applications. The architecture defines a run-time environment with dynamic state machine creation and destruction, machine dispatching, intermachine communication, and timing. The architecture and implementation pattern are application and operating system independent. Moreover, the new dictionary language constructs are being introduced to increase the amount of implementation that can be generated from the abstract formal design model.

## Acknowledgments

The success of the VFSM project is attributable to many people who provided support in a variety of ways that allowed a fledgling technology to gather strength and grow. The original concept of VFSM was created by Dr. Ferdinand Wagner, an outside consultant who worked with the Lucent development team. Without the initial support of Lucent managers Norm Chaffee and Dave Smith, there never would have been a VFSM project.

Early Lucent adopters Al Sawyer, Mary Mui, E-Ling Lin, Fangli Chang, and Lily Wang kept the project alive by using VFSM in call processing development. Lee Stecher and Randy Duerr maintained support of VFSM as the technology continued to mature. Tom Mills provided numerous tool enhancements, integrated the VFSM translation tools with the 5ESS switch software development environment, and invented a VFSM encoding arrangement that provided a significant space and time optimization.

A second wave of enthusiastic Lucent VFSM users and part-time apostles, Paul Iverson, Susan Eick, Bob Perez, Bill Bielawski, Scott Gavin, Pat Polsley, and Costas Tsioras applied VFSM to larger design problems in the hardware maintenance and signaling areas.

Additionally, the authors thank the anonymous referees for their detailed comments on an earlier version of this paper.

## *Trademark

UNIX is a registered trademark of Novell.

## References

1. F. Wagner, "VFSM Executable Specification," *Proceedings of the IEEE International Conference on Computer System and Software Engineering*, The Hague, The Netherlands, 1992, pp. 226-231.
2. A. R. Flora-Holmquist, J. D. O'Grady, and M. G. Staskauskas, "Telecommunications Software Design Using Virtual Finite-State Machines," *Proceedings of the International Switching Symposium (ISS '95)*, Berlin, Germany, Apr. 1995, pp. 103-107.
3. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, Apr. 1990, pp. 403-414.
4. R. Braek and O. Haugen, *Engineering Real-Time Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
5. A. R. Flora-Holmquist and J. D. O'Grady, *Finite-State Machine with Minimized Vector Processing*, U. S. Patent 5,459,841, applied for Dec. 1993, issued Oct. 1995.
6. A. R. Flora-Holmquist and T. L. Mills, *Finite-State Machine with Minimized Memory Requirements*, U. S. Patent 5,473,531, applied for Dec. 1993, issued Dec. 1995.
7. A. R. Flora-Holmquist and M. G. Staskauskas, "Formal Validation of Virtual Finite-State Machines," *Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT '95)*, Boca Raton, Florida, Apr. 1995, pp. 122-129.
8. G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
9. A. R. Flora-Holmquist and M. G. Staskauskas, "Moving Formal Methods into Practice: The VFSM Experience," *Proceedings of the First Workshop on Formal Methods in Software Practice (FMSP '96)*, San Diego, California, Jan. 1996, pp. 49-59.

***(Manuscript approved March 1997)***

*ALAN R. FLORA-HOLMQUIST is a member of technical*

*staff in the Platform Planning, Access Control, and Test Department at Lucent Technologies in Naperville, Illinois. Currently, he is responsible for ICore system software integration. Previously, he led the virtual finite-state machine (VFSM) team and was responsible for training, consulting, and tool development. Mr. Flora-Holmquist has a B.A. degree in mathematics from Marist College in Poughkeepsie, New York, and an M.S. in computer science from New Mexico State University in Las Cruces.*

*EDWARD MORTON is a member of technical staff in the*

*Platform Planning, Access Control, and Test Department at Lucent Technologies in Naperville, Illinois. He is responsible for virtual finite-state machine (VFSM) training, consulting, and platform development. Mr. Morton has a B.S. degree in computing science from Glasgow University in Scotland.*

*JAMES D. O'GRADY is a member of technical staff in the*

*Platform Planning, Access Control, and Test Department at Lucent Technologies in Naperville, Illinois. He develops tools for the virtual finite-state machine (VFSM) project and works on peripheral control for the 5ESS® switching system. Mr. O'Grady has a B.S. degree from the University of Illinois at Urbana-Champaign, as well as an M.S. from Stanford University in California, both in computer science.*

*MARK G. STASKAUSKAS is a member of technical staff*

*in the Software Production Research Department at Bell Labs in Naperville, Illinois. He is responsible for research into formal methods for the design of concurrent programs and their application to Lucent Technologies software. Mr. Staskauskas has a B.S. degree from Columbia University in New York, an M.S. from the University of California at Los Angeles, and a Ph.D. from the University of Texas at Austin, all in computer science.* ◆