Tales from the Front: Industrial Experience with Formal Validation (Preliminary Version)

Mark Staskauskas
Software Specification and Analysis Research Dept.
AT&T Bell Laboratories
1000 E. Warrenville Rd.
Naperville, IL 60566
markstas@research.att.com

Abstract

We have developed a formal validation tool that has been used in several projects that are developing software for AT&T's 5ESS™ telephone switching system. The tool validates networks of communicating processes specified in the Virtual Finite State Machine (VFSM) notation, which is used during the low-level-design phase of software development to specify the control behavior of an implementation. The VFSM validator performs a reachability analysis, using Holzmann's bitstate-hashing technique [Ho88], to check for errors in VFSM communication such as deadlock, livelock and unexpected inputs. In this paper, we present an overview of the VFSM methodology and validator, report on the present status of the VFSM validation effort, and discuss some of the lessons we have learned about the proper role of formal validation in the software development process.

1 Introduction

Techniques for the automatic verification of concurrent programs have been the subject of a great deal of research over the past two decades. Recently, there have been many reports of successful applications of these techniques to industrial-strength hardware and software design problems. Motivated by these successes, we have been working to introduce a formal validation tool into a large AT&T software development organization that produces software for the 5ESS™ telephone switch. The input language of the tool is the Virtual Finite State Machine (VFSM) notation ([Wa92], [FS95b]), which was already in common use in the 5ESS™ environment prior to the development of our tool. Our goal was to make formal validation a routine part of the VFSM design process. The VFSM validator has been available to 5ESS™ developers for about two years, and over 20 of them have used it to find errors in their VFSM designs prior to testing; this is one of the first instances we know of in which validation technology has achieved widespread use in an industrial setting.

Our experience with validation differs from other examples reported in the literature in that we perform validation much later in the software-design process. In the NewCoRe project [Ho94a], for example, validation was performed on SDL specifications by highly trained "validation engineers" during the requirements and specification phases. The VFSM validator, by contrast, is used by software developers during the low-level-design phase, just prior to coding. The use of validation at this point in software design has its advantages and disadvantages. The VFSMs we validate are quite close to the implementation, which allows us to find errors that would be missed if a more abstract model was validated; however, the amount of detail that typically appears in our VFSMs can make the state explosion problem more severe.

The purpose of this paper is to share our experiences with others who are familiar with formal validation and are interested in its practical application. Much of our success with the VFSM validator is attributable to the fact that validation has been carefully integrated with the other parts of the VFSM design methodology, such as interactive simulation and automatic code generation. It is our belief that validation is not a stand-alone activity, but merely one part of a design methodology for concurrent programs. Despite our successes, we have also found that using present validation tools is a difficult process that demands a great deal of persistence and patience on the part of the user, and that the more routine and widespread use of validation will require significant technical advances that allow it to be performed with much less user intervention.

The remainder of this paper is organized as follows. We begin in Section 2 with a brief overview of the VFSM methodology and toolset. Section 3 describes how we developed a validator for the VFSM notation. In Section 4, we outline the present status of the VFSM validator, including some planned enhancements. Section 5 discusses some of the lessons we have learned in the course of introducing the VFSM validator into the 5ESS™ software development environment. Section 6 contains our conclusions.

2 VFSM Overview

The VFSM methodology [Wa92] consists of a *design paradigm*, in which the control behavior of a software module is specified as a finite-state machine; and an *implementation paradigm*, which consists of a design structure that defines the interface between the control specification and the rest of the implementation. The VFSM toolset translates the VFSM specification into executable form and produces templates for the modules that interface with the control portion of the implementation. The toolset also includes a simulator that enables the designer to execute a VFSM specification interactively, providing it with inputs and verifying that the outputs and state transitions produced in response accord with his expectations.

A VFSM specification is written in terms of states, virtual inputs and virtual outputs. The term "virtual" means that VFSM inputs and outputs are abstract names local to the VFSM: virtual inputs represent conditions in the environment that influence the control behavior of the specified system, and virtual outputs stand for actions to be taken by the system at various points during its execution. The exact binding between these abstract inputs and outputs and their concrete realizations in the implementation is specified by the VFSM implementation paradigm.

A major difference between a VFSM and a traditional FSM is in how inputs are handled. When a VFSM receives an input, it is stored in a set called the Virtual Input Register (VIR), and remains there until it is explicitly removed. VFSM state transitions and the production of virtual outputs can be conditioned on the presence of particular subsets of inputs in the VIR. VFSM is therefore an *extended* FSM model: the "state" of a VFSM at any point is given by its VFSM

```
S_SEND_REQ {
     E: O_SEND_MSG1, O_START_TIMER;
    IA: I_ACK     ? O_MSG_OK;
        I_TIMEOUT ? O_REPORT_ERROR;
    NS: I_ACK     > S_SEND_NEXT_MSG;
        I_TIMEOUT > S_ERROR;
}
```

Figure 1: Example State of a VFSM Specification

state and the contents of its VIR. The addition of the VIR adds considerable expressive power to the model, yet still assures that specifications are written at a high level of abstraction.

Figure 1 shows an example specification of one state of a VFSM. The example illustrates a simple handshake protocol. The entry-action (`E:`) section specifies that upon entry to this state, the VFSM will produce virtual outputs representing the sending of a message (`O_SEND_MSG1`) and the starting of a timer (`O_START_TIMER`). The input-action (`IA:`) section specifies that, if the desired reply to the message is received (`I_ACK`), appropriate action will be taken (`O_MSG_OK`); however, if the timer expires, indicating that the message or its reply has been lost, then error processing will take place (`O_REPORT_ERROR`). The next-state transition (`NS:`) section defines the VFSM state that will be entered next: either the subsequent step in the handshake protocol (`S_SEND_NEXT_MSG`) or an error-handling state (`S_ERROR`).

Figure 2 shows the structure of a VFSM implementation. The input mapper and output functions provide a "firewall" that enforces the separation of the top-level control behavior defined by the VFSM specification from the low-level data manipulations and functions of the system. As shown to the left of Figure 2, the input mapper receives *events*, such as messages, interrupts and timer expirations, from the environment of the system. Based on the event received and the values of local data structures, the input mapper determines which virtual inputs must be inserted into, or deleted from, the VIR.

When the input mapper completes, the VFSM specification is executed. The VFSM may change state several times and produce several virtual outputs, terminating execution when a state is reached from which no state transition is possible given the current VIR contents. The user associates with each virtual output the name of an *output function*; whenever that output is produced during VFSM execution,

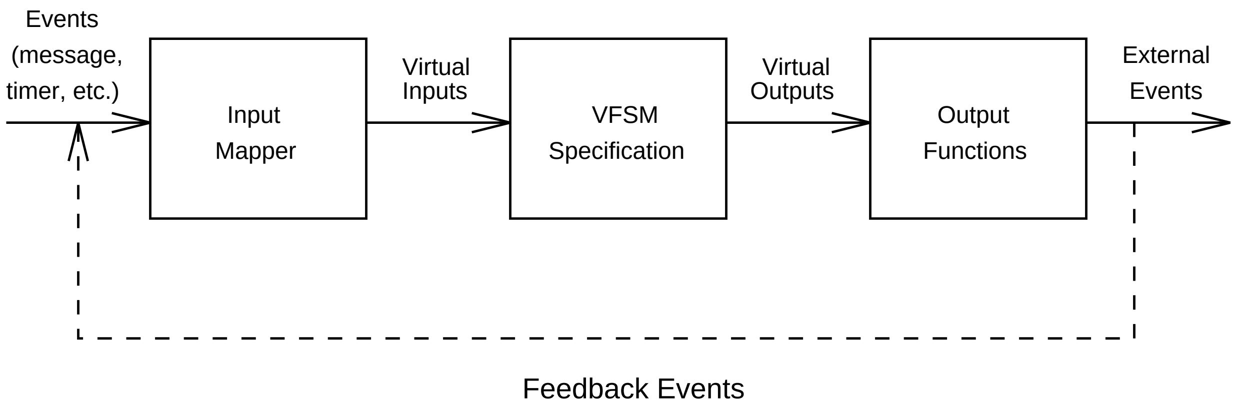

Figure 2: Structure of a VFSM Implementation

its output function is invoked. The output function performs whatever processing and data manipulation are necessary to realize the abstract behavior represented by the virtual output. Note that an output function may also invoke the input mapper with a *feedback event*, as suggested in Figure 2; thus, the VIR can change while the VFSM is executing. Feedback events are typically used when an output function detects an error condition that must be dealt with immediately by the VFSM.

3 The VFSM Validator

We now discuss how we adapted Holzmann's bitstate hashing technique to the validation of networks of VFSMs. To ameliorate the effects of the state explosion problem, an effective validation tool must keep the size of the global state as small as possible. To this end, one approach would be to include only the VFSM state and VIR as part of the global state, ignoring the input mapper and output functions. However, the purpose of formal validation is to detect errors in interprocess communication, and it is clear from Figure 2 that the input mapper and output functions contain information that is crucial to the communication behavior of the system, including feedback events and the sending and receiving of messages. Thus, we also considered allowing input mapper and output function C code to be part of the validation model. However, this C code also contains much information that is *not* relevant to inter-process communication, so including the input mapper and output functions would make the global state larger than necessary.

3.1 Mapping Abstractions

The approach we selected is a compromise between the above extremes: we do not allow user-defined C code and data in the validation model, but instead provide a feature known as *mapping abstractions* that allows the user to specify those aspects of the input mapper and output functions that have an impact on inter-process communication. In the validator, VFSMs communicate by sending each other events, and each VFSM has an event queue. When a VFSM receives an event, mapping abstractions representing the input mapper allow the user to define all possible combinations of virtual inputs that might be inserted into, or deleted from, the VIR. If more than one combination is possible, the validator will explore a separate successor state for each combination; thus, the mapping abstractions permit nondeterminism. Similarly, the mapping abstraction for each virtual output allows the specification of all possible combinations of inter-VFSM or feedback events that might be generated by its output function when that virtual output is produced.

Figure 3 presents two examples that illustrate the use of mapping abstractions; each example contains a fragment of C code that might appear in an input mapper or output function, along with a mapping abstraction intended to model its behavior. The first example shows an event representing a telephone call setup request. When the event is received, the input mapper does a database lookup to determine whether or not the caller has paid his bill, and inserts into the VIR either input `I_BILL_PAID` or `I_BILL_NOT_PAID` based on the outcome of the lookup. The mapping abstraction for this event specifies that either input is possible. The second example illustrates a mapping abstraction for an output function. The function invokes the system primitive `OSSENDMSG` to send the message `RESP_MSG` to the process `OTP`. However, `OSSENDMSG` may return an error code indicating that the attempt to send the message failed; if this occurs, the output function generates the feedback event `ERROR_EVENT` by calling the input mapper. The mapping abstraction captures both possible outcomes of the call to `OSSENDMSG`. These examples show how nondeterminism can be used to model the behavior of parts of the system that cannot be modeled completely in the validator.

3.2 Using the Validator

To validate a VFSM application, the user first constructs any necessary environment VFSMs that are needed to close the system. For each VFSM, the user then supplies mapping abstractions for each event and virtual output, and specifies its initial state and possible initial VIR configurations. The VFSM translation tools are then run to translate each VFSM specification into C code. The validator uses this C code and code implementing the validation algorithm to produce an executable validation program for the VFSM network. This program generates global states of the network using the tool-generated C code to produce successors corresponding to each VFSM. As the global states are generated, the validator checks for the following types of errors:

Deadlock: A deadlock is reported if a global state is reached in which all VFSMs are blocked waiting for events, all the event queues are empty, and at least one VFSM is not in a terminating state.

Unexpected Inputs: In each VFSM state, the expected virtual inputs are those appearing in its input-action and state-transition conditions. If an input is inserted into the VIR that does not appear in these conditions, the validator reports an unexpected input.

Queue Overflow: The inter-VFSM event queues have a maximum size selectable by the user. If an event is appended to a full queue, the validator reports a queue overflow.

Coverage: While the validation algorithm is executing, it records which states, input actions, and next-state transitions are executed for each VFSM. At the end of the validation run, the validator prints out a report listing the portions of each VFSM that were never executed, which may represent dead code.

Livelock: A livelock, or non-progress cycle, is a cycle of global states in which some VFSM fails to pass through a state designated by the user as a "progress state".

When the validator finds one of the above errors, it prints out an error trace, in the form of a message sequence diagram, that illustrates the complete execution scenario that resulted in the error. The level of detail in the error trace is selectable by the user, and can include inter-VFSM and feedback events, VFSM state transitions, production of virtual outputs, and insertion and deletion of virtual inputs.

4 Present Status and Plans

4.1 Data Modeling and Temporal Logic

Although VFSM attempts a complete separation of control and data, we have found that, for large designs, control flow can depend in subtle ways on the values of data structures. We are planning to add a simple data-modeling language to the mapping-abstraction notation described above that will allow users to accurately represent the impact of data structures on control behavior.

The VFSM validator presently checks for the satisfaction of "healthiness conditions" such as the absence of deadlock and unexpected inputs. However, the absence of these kinds of errors obviously does not insure that a VFSM design is correct; one would like to be able to specify the application-specific requirements to be met by the VFSM and have these checked by the validator. Temporal logic is a specification language that is well-suited to precisely stating the requirements that VFSM applications must satisfy. We plan to add the ability to specify and check temporal-logic properties as part of the VFSM validation process.

4.2 Partial-Order Methods

Partial-order methods ([God94], [KP92b]) are techniques that exploit the independence of process operations to avoid the redundant exploration of execution scenarios during validation. We recently developed a partial-order algorithm for the VFSM validator and tested it on four 5ESS™ VFSM applications. We found that it resulted in reductions in the size of the global state space of as much as an order of magnitude. This work is reported in [GPS96].

Modeling the input mapper:

in C code:

```
case E_CALL_SETUP:
    if (cust_data->bill_status == PAID)
        VIR_insert(I_BILL_PAID);
    else VIR_insert(I_BILL_NOT_PAID);
```

in VFSM validator:

```
$name E_CALL_SETUP
$inlist I_BILL_PAID || I_BILL_NOT_PAID
```

Modeling the output functions:

in C code:

```
case O_RESP_TO_OTP:
   return_code = OSSENDMSG(RESP_MSG, OTP);
   if (return_code == FAIL)
       input_mapper(ERROR_EVENT);
```

in VFSM validator:

```
$name O_RESP_TO_OTP
$evlist OTP.RESP_MSG || ERROR_EVENT
```

Figure 3: VFSM Validator Mapping Abstractions

4.3 Test-Sequence Generation

There has been a great deal of research on techniques for automatically generating minimal sequences of inputs that collectively exercise all state transitions of a finite state machine [LY95]. We have been exploring the use of such techniques in the context of VFSM. Using a modified version of the VFSM validator, our goal is to obtain a set of sequences of events (see Figure 2) that causes complete coverage of a VFSM, i.e. the execution of every input action and next-state transition (see Figure 1) at least once. Such a set of test sequences could be helpful to users of the VFSM simulator, as well as to testers responsible for testing a VFSM implementation when all coding is complete.

5 Lessons

5.1 The Importance of Interactive Simulation

After writing a VFSM specification, a developers typically uses the VFSM simulator to exercise selected scenarios and find obvious errors, and then uses the validator, which invariably finds many more subtle errors. Many proponents of validation believe that interactive simulation is a form of testing that would be rendered obsolete in the presence of a validation capability. We have found, to the contrary, that simulation and validation are complementary, rather than conflicting, techniques for finding errors in VFSM designs. The ability provided by simulation to look in detail at selected execution scenarios greatly enhances the developer's understanding of a VFSM design.

It is interesting to note that developers have found the VFSM simulator much easier to use than the validator. This is probably because the simulator is similar to various debuggers that they are already familiar with, and can be used without a detailed knowledge of its internal operation. Perhaps for this reason, our tool usage reports show that invocations of the VFSM simulator outnumber those of the validator by an order of magnitude.

In order to exploit the popularity of the simulator, we recently unified the simulator and validator so that they share a common interface, which has resulted in greatly increased usage of the validator. Prior to the unification, many users of the validator perceived the need to construct environment VFSMs and mapping abstractions as a burden. However, several users remarked that environment VFSMs would be useful during simulation. Previously, each simulation scenario required the laborious construction of a command script containing each external message to be sent to the simulated VFSM in that scenario. Since the environment machines embody all possible behaviors of the environment, their presence during simulation would eliminate the need for hand-constructed input scripts. We unified the VFSM simulator and validator so that all information needed for validation would be accessible in simulation sessions. To simulate a given scenario, the user need only make a series of menu selections to cause the desired environment behavior; this makes use of the simulator much more effective, as the number of simulation scenarios that can be completed in a given time period is greatly increased. An added benefit of the unification is that validator usage has been made simpler: the task of constructing environment machines no longer is perceived as a burden, since it makes simulation so much easier.

5.2 Generating Code From the Validation Model

One of the most popular features of the VFSM design methodology is the ability to derive a portion of the implementation automatically from the VFSM specification. To exploit this feature, we have designed the validator so that it uses the C code produced by the VFSM translator to generate successor states during validation. This gives developers a high degree of confidence that they are validating the actual implementation and not merely an abstract model of it. Unfortunately, we have no way at present of guaranteeing that the mapping abstractions described in Section 3 are consistent with the input-mapper and output-function C code. We are investigating the design of single language from which both the mapping abstractions and the implementation C code could be derived.

With most validation tools, the usual way of dealing with the state explosion problem is to abstract away portions of the validation model until one of tractable size is obtained. In our environment, however, this would result in the validation of a reduced VFSM that is different from the VFSM that will actually execute on the 5ESS™ switch. We are therefore investigating various forms of "partial" validation that will allow us to perform a reasonably thorough state-space search of VFSM examples that are too large for normal validation to run to completion in a reasonable amount of time. Note that, although the use of bitstate hashing constitutes a form of partial validation, the depth-first search strategy leads to validation runs in which large portions of the VFSMs to be validated are never executed. Our goal is a search strategy in which some depth-first paths are pruned in such a way that, at a minimum, complete VFSM coverage is obtained.

5.3 Accurate Modeling of the Execution Environment

The purpose of validation is to detect errors in the interactions among a collection of concurrent processes. It therefore follows that the input language of a validation tool must allow the user to model the type of inter-process communication supported by the run-time environment in which the implementation will ultimately execute. Early versions of the VFSM validator were unable to accurately model certain features of the target 5ESS operating system, such as timers and process scheduling and priorities, that had an impact on inter-VFSM behavior; because of this, users had to make special versions of their VFSMs equipped with logic that prevented some scenarios that were impossible in the 5ESS environment. We recently upgraded the validator to model these features of the operating system, creating an execution environment that mimics that present on the 5ESS switch and making the validator much more convenient to use.

Our experience suggests that there is no single validator input language that is suitable in all contexts. We note that bitstate hashing has been implemented for at least three languages (Promela, SDL and VFSM), and there have been numerous translators built between input languages for different validation tools. It would therefore seem useful to package validation algorithms in the form of libraries so that validators for new languages could be built quickly; at present, validators for new languages must be painstakingly hand-crafted.

5.4 Education Issues

A major premise of our work has been that the validator should be usable by software developers as one of many design tools in their arsenal, without requiring a great deal of expertise. We have prepared a half-day validator training class, which is part of a four-day VFSM class, to provide developers with enough information to use the validator effectively. The validator class includes hands-on experience in the form of simple validation exercises.

When users began to apply the validator to 5ESS™ design examples, many of them ran into significant difficulty. In responding to their questions, it became clear that effective use of the validator requires an in-depth understanding not only of *what* the validator is attempting to do, but also of *how* the validator does its job. The necessary information includes a basic understanding of the interleaving model of concurrency and how all interleavings are explored by a depth-first generation of the global-state space. In response to the many questions raised by users, we have modified the validator training class to include a rudimentary overview of validator operation. In spite of this, we have found that, although some users (particularly those with a strong mathematical background) quickly become proficient in validation, many others require a great deal of consultation before they can use the validator successfully. We now feel that a half-day's worth of training is not sufficient, and are looking into a longer class.

6 Conclusions

Our experience shows that validation can be profitably used, albeit with some difficulty, by software developers with relatively little training. We have also demonstrated that the low-level-design phase of software development is a promising, though often overlooked, niche for the introduction of validation.

Holzmann [Ho94b] has noted that Promela and SPIN were designed in accord with the "Unix philosophy," i.e. to do one thing (validation) and do it well. While the Unix philosophy is certainly pertinent to the design of file-manipulation utilities like `diff` and `grep`, we feel that a tool for the design of concurrent programs must perform *many* functions, and validation must be a cleanly integrated part of such a tool. In addition to VFSM, there have recently been many other examples of design methodologies based on extended finite state machines that are supported by powerful toolsets; examples include Statecharts and the Statemate toolset [Ha90], [Ha92]; SDL and the GEODE [VE95] and SDT [Te95] toolsets; and the Real-Time Object-Oriented Modeling (ROOM) methodology and the ObjecTime toolset [SGG94]. All of these toolsets have been used extensively in industry, and though most of them have a limited validation capability, they also provide a number of valuable features not available in typical validation tools, e.g. graphical editing and browsing, automatic generation of test cases, interactive simulation, code generation, configuration control, documentation support, and requirements traceability. Together, these toolsets constitute compelling evidence that formal methods can lead to substantial improvements in software quality. An easy-to-use validation function would greatly enhance the error-detection capability of the toolsets, and would allow automatic checking that a design model satisfies its requirements.

At present, formal validation is still an experimental technology. Perhaps the best evidence of this fact is that a carefully worked application of validation to a real-world design problem remains a publishable achievement. We have argued that present validation tools are difficult to use because they require the user to combat the state-explosion problem and have a detailed understanding of state-space search. We hope that, in the future, validation tools will be able to do their jobs with far less involvement on the part of the user.

References

[FS95a] A. R. Flora-Holmquist and M. G. Staskauskas. Formal validation of virtual finite state machines. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT95)*, Boca Raton, FL, April 1995.

[FS95b] A. R. Flora-Holmquist, J. D. O'Grady and M. G. Staskauskas. Telecommunications software design using virtual finite state machines. In *Proc. Intl. Switching Symposium (ISS'95)*, Berlin, Germany, April 1995.

[FS96] A. R. Flora-Holmquist and M. G. Staskauskas. Moving formal methods into practice: The VFSM experience. Submitted for publication.

[God94] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem.* PhD thesis, University of Liège, Computer Science Department, November 1994. (Also available by anonymous ftp from ftp.montefiore.ulg.ac.be in the pub/po-package directory, file thesis.ps.Z).

[GPS96] P. Godefroid, D. Peled and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In *Proc. Intl. Symposium on Software Testing and Analysis (ISSTA'96)*, San Diego, CA, January 1996 (to appear).

[Ha90] D. Harel et al. Statemate: A working environment for the development of complex reactive systems. In *IEEE Transactions on Software Engineering*, vol. 16, no. 4, April 1990, pp. 403-414.

[Ha92] D. Harel. Biting the silver bullet: Toward a brighter future for system development. *IEEE Computer*, January 1992, pp. 8-20.

[Ho88] G. J. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18(2):137–161, 1988.

[Ho94a] G. J. Holzmann. The theory and practice of a formal method: NewCoRe. In *Proc. IFIP World Congress*, August 1994.

[Ho94b] G. J. Holzmann. Proving the value of formal methods. In *Proc. 7th Intl. Conf. on Formal Description Techniques (FORTE'94)*, October 1994.

[KP92b] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.

[LY95] D. Lee and M. Yannakakis. Principles of testing finite state machines—A survey. *Proceedings of the IEEE*, to appear.

[SGG94] B. Selic, G. Gullekson and P. Ward. *Real-Time Object Oriented Modeling*. J. Wiley & Sons, 1994.

[Te95] Telelogic AB. SDT—The SDL Design Tool. WWW page, URL `http://www.telelogic.se`.

[VE95] VERILOG. *Object*GEODE. WWW page, URL `http://www.tdr.dk/public/SDL/verilog`.

[Wa92] F. Wagner. VFSM executable specification. *Proc. CompEuro*, 1992.