Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs

Patrice Godefroid, Doron Peled and Mark Staskauskas

IEEE Transactions on Software Engineering, volume 22, number 7, pages 496-507, July 1996.

Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs

Patrice Godefroid, Doron Peled and Mark Staskauskas

Patrice Godefroid and Mark Staskauskas are with Bell Laboratories, Lucent Technologies Inc., 1000 E. Warrenville Rd., Naperville, IL 60566. E-mail: {god,markstas}@bell-labs.com. Doron Peled is with Bell Laboratories, Lucent Technologies Inc., 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: doron@bell-labs.com.

August 29, 1996

Abstract

Formal validation is a powerful technique for automatically checking that a collection of communicating processes is free from concurrency-related errors. Although validation tools invariably find subtle errors that were missed during thorough simulation and testing, the brute-force search they perform can result in excessive memory usage and extremely long running times. Recently, a number of researchers have been investigating techniques known as *partial-order methods* that can significantly reduce the computational resources needed for formal validation by avoiding redundant exploration of execution scenarios. This paper investigates the behavior of partial-order methods in an industrial setting. We describe the design of a partial-order algorithm for a formal validation tool that has been used on several projects that are developing software for the Lucent Technologies 5ESS® telephone switching system. We demonstrate the effectiveness of the algorithm by presenting the results of experiments with actual industrial examples drawn from a variety of 5ESS application domains.

Keywords

Formal methods, automatic verification, validation, partial-order methods, concurrent programs, reachability analysis.

I. Introduction

Formal validation enables the automatic detection of errors such as deadlock and livelock in networks of communicating processes. By exhaustively exercising possible execution scenarios of the processes, formal validation can detect subtle errors that are missed during simulation and testing, where only a small fraction of the scenarios can be checked for correctness.

We have designed a tool that validates networks of communicating processes implemented in the Virtual Finite State Machine (VFSM) notation ([18], [2]). Our goal was to make formal validation a routine part of the VFSM design methodology, which is used on a variety of software projects for the 5ESS® (No. 5 Electronic Switching System), a product of Lucent Technologies Inc. (formerly the systems and technology unit of AT&T). To date, over twenty-five 5ESS developers have used the validator to search for errors in their VFSM designs. Our experience represents one of the first instances we know of in which validation technology has achieved widespread use in an industrial software development organization (see [1] for more details about our experiences with technology transfer).

Despite its many successful applications, the validator can be a difficult tool to use: the running time of validation typically grows exponentially with the size of the VFSM(s) being validated. The first few runs of the validator usually find errors quickly; however, as errors are detected and corrected, one soon reaches the point where the validator must explore millions of states before the next error is found. Often, only one or two validation runs per day can be completed, so obtaining an error-free validation run requires a great deal of patience and persistence on the part of the user.

We therefore became interested in techniques that might decrease the running time of validation. One such family of techniques, known as *partial-order methods*, has achieved impressive reductions in running time on many examples ([4], [13], [16]). The goal of validation is to check all possible execution scenarios of a collection of communicating processes for errors, where a scenario consists of an interleaving of execution steps of the processes. It is often the case that many steps in a scenario are independent, i.e., the order in which they appear in a scenario does not affect the presence or absence of an error. For example, if two processes both increment an integer variable in successive steps, the end result is the same regardless of the order in which these steps occur. Partial-order methods exploit the fact that, if two scenarios differ only in the relative ordering of independent execution steps, it is sufficient to check only one of the scenarios for errors. In validation examples where there is a great deal of independence among process execution steps, this can result in an enormous reduction in the memory and CPU resources needed for validation.

Our initial investigation of the usefulness of partial-order methods in the context of the VFSM validator revealed a number of potential difficulties. VFSMs communicate by asynchronous message-passing, and an execution step of a VFSM can involve receiving a message from its queue and sending messages to several other VFSMs. The large granularity of these execution steps limits the amount of independence among them: any two send operations on the same queue are dependent, since the order in which the messages appear in the queue is significant.

Another problem is that implementation of partial-order methods requires knowledge of the operations on shared data structures that a concurrent process will perform in a given execution step. In the case of VFSM, it is not easy to determine this information *a priori*. One approach would be to conservatively estimate the possible operations of a VFSM with a compile-time static analysis of its structure. Although relatively inexpensive to obtain, this information may overestimate the operations a VFSM can perform in a given step, and therefore *underestimate* the amount of independence among the steps, preventing partial-order methods from obtaining the maximum possible reduction in computational resources. Better information could be obtained by simply executing the VFSM and recording the operations it performs. However, execution of a VFSM for purely informational purposes would add a great deal of overhead that might well negate the reduction obtained by partial-order methods. Thus, there is a tradeoff between the accuracy of the information needed for partial-order methods and the cost of obtaining it.

We present in this paper a partial-order algorithm for the validation of networks of VFSMs. We also discuss how we dealt with the problems mentioned above in the course of developing the algorithm. Our algorithm is unique in that it rests on a model of concurrency that incorporates all the subtleties of a real-life operating-system environment, including timers and process priorities. This is in contrast with other existing algorithms of this family, which were designed to analyze systems modeled in simpler, more abstract formalisms, such as Petri nets and communicating finite-state automata. To investigate the extent of the reduction in validation effort obtained with partial-order methods, we present the results of validation runs with and without the partial-order algorithm on four VFSM examples that represent actual 5ESS applications. For each example, we perform three experiments with the partial-order algorithm. Each experiment uses a different approach to compute the information about the operations a VFSM performs in an execution step; the approaches differ in how they resolve the cost-accuracy tradeoff mentioned above. We show that the partial-order algorithm leads to reductions in the number of execution steps explored that range from 13% to more than a factor of five.

The remainder of the paper is organized as follows. We begin in Section II with a high-level overview of the VFSM methodology and toolset. Section III contains a precise mathematical description of the execution of a collection of communicating VFSMs. We then present in Section IV our adaptation of Holzmann's supertrace algorithm [8] to the validation of a network of VFSMs. Section V presents a partial-order algorithm for VFSM validation and its correctness proof. Section VI details the three approaches we used to determine the operations a VFSM performs in an execution step. In Section VII, we briefly describe the four 5ESS VFSM application examples used to test the effectiveness of the partial-order algorithm, and present the results of our experiments, including a discussion of how the structure of the examples influences the reductions obtained. We conclude in Section VIII with a discussion and our plans for future research.

II. VFSM Overview

The VFSM methodology [18] 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 allows the designer to execute a VFSM specification interactively.

The VFSM methodology has been used on over 75 5ESS software projects, and has been taught to several hundred 5ESS developers. The VFSM specification notation is easy to learn, since it is a slight extension to the familiar concept of a finite state automaton. Because part of the implementation is generated automatically from a VFSM specification, developers feel that the effort spent in writing the specification is repaid by less work during the coding phase, in contrast to other methodologies that result in only a piece of documentation.

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.

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

Fig. 1. Example state of a VFSM specification

Each input received by a VFSM 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.

Fig. 1 shows an example specification of one state of a VFSM. The example illustrates a simple handshake protocol. Thb 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 exit-action (`X:`) section lists the outputs that are to be produced when there is a transition out of this state; in this example, the timer started upon entry to the state is stopped. The input-action (`IA:`) section specifies that, if the desired acknowledgement to the message is received, which is represented by the presence of virtual input `I_ACK` in the VIR, 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`).

Fig. 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

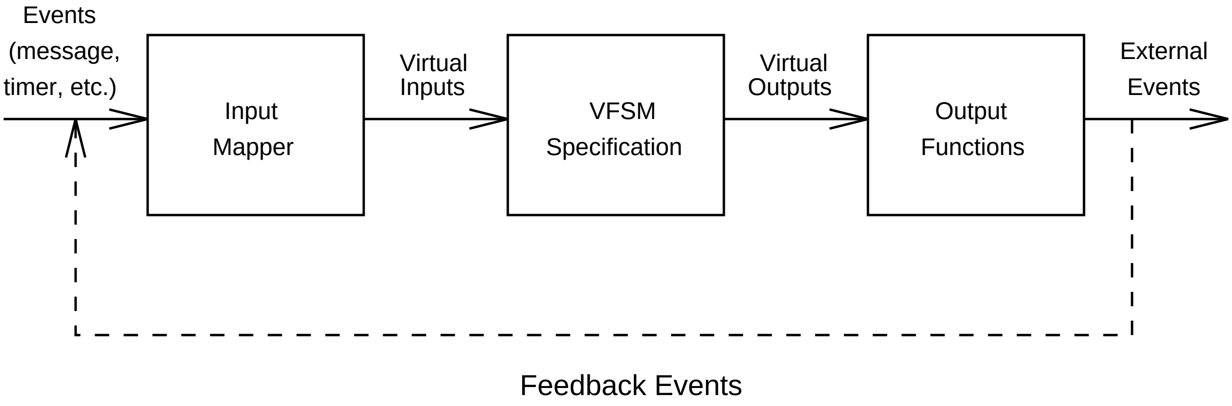

Fig. 2. Structure of a VFSM implementation

of the system. As shown to the left of Fig. 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, 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 Fig. 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.

In a VFSM implementation, inter-process communication occurs via system calls to send and receive messages that appear in the output functions and input mapper. In order to keep the VFSM model used for validation as small as possible, we do not allow the inclusion of the input mapper and output functions in the model. Instead, we 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.

For the input mapper, mapping abstractions specify the possible combinations of virtual inputs that might be inserted into, or deleted from, the VIR for each received event; for the output functions, the user can specify all combinations of timer operations, feedback events, and events sent to other VFSMs for each virtual output.

To use the validator, the user first defines mapping abstractions for each received event and virtual output of the VFSM(s) to be validated. It may also be necessary to construct "environment VFSMs" that model the external environment of the validated VFSMs; for example, when validating a telephone-call setup protocol, one would construct a VFSM that models the possible behavior of the caller at each step in the protocol. Given a network of communicating VFSMs, the validator generates possible execution scenarios in a manner described in more detail below; it checks for errors in inter-VFSM communication such as deadlock, livelock, unexpected inputs, and message buffer overflows. In future versions of the validator, we plan to allow the user to specify application-specific properties in temporal logic that will be checked during validation.

III. A Formal Model of VFSM Execution

In order to define a formal validation algorithm for a network of VFSMs, it is necessary to construct a precise mathematical model of the VFSM execution environment provided by the operating system under whose control the 5ESS software executes. A *VFSM system* consists of a collection of *processors*, which are intended to represent entities that can execute concurrently. Each processor contains one or more *processes*, at most one of which can execute at a time; thus, all processes in a processor execute under control of the same scheduler. Each process contains one or more VFSMs. There exists a fixed *priority* mechanism between the processes of the same processor. That is, there is a total order '>' such that if $P_1 > P_2$, and both processes can proceed, P_2 would have to wait for P_1 to go first (see Fig. 3). All communication among VFSMs is by asynchronous message passing. Each process has two bounded-length queues, one for intra-process communication among the VFSMs in that process, and another on which messages from other processes (on the same or a different processor) are received. It is assumed that the queues are large enough to contain the maximum number of messages that might be sent during execution of the system, so that queue overflow never occurs. Thus, an attempt to send a message to a queue that is full is a design error that is repaired by enlarging the queue or changing the logic of the VFSMs in the system.

In addition, each VFSM may have one or more timers that it can start and stop. Once a timer is started, it can expire at any time (it is not possible to associate a time duration with timers in our model). Expiration of a timer results in the sending of a message to the VFSM that started it. Since timer expiration occurs independently of the execution of VFSMs, each timer can be thought of as a separate processor.

An *execution* of a VFSM system is a (possibly infinite) sequence of *transitions*, where a transition is either the execution of a VFSM or the expiration of a running timer. A transition of a VFSM, in turn, consists of a finite sequence of *operations*, where an operation is the sending or receiving of a message, the starting or stopping of a timer, or an internal execution step that changes the state and/or VIR of the VFSM. At any point in an execution, at most one VFSM on each processor is enabled, and only the enabled VFSM can execute next. Below, we give a precise description of operations and transitions and of how the enabled VFSM on each processor is determined.

A. Global States and Executions

We model each process P as a seventuple $\langle M_P, V_P, H_P, L_P, S_P, I_P, T_P\rangle$, where M_P is the alphabet of possible *input messages*, and V_P is the set of VFSMs in that process. Each message received by a process is addressed to one of its VFSMs; thus, the incoming messages of process P are of the form $v(m)$, where $v \in V_P$ and $m \in M_P$. H_P and L_P are two queues, called the *high-priority queue* and *low-priority queue*, respectively. A message sent from one VFSM to another VFSM *in the same process* would get to the end of the high-priority queue of that process, while an inter-process communication would be through the low-priority queue. A process would prefer reading messages from its high-priority queue; thus, it would read a message from its low-priority queue only if the high-priority queue is empty. S_P is a finite set of *process states*. Each state has several components, including a component for each queue $s(H_P)$ and $s(L_P)$, and a separate component $s(v)$ for each VFSM $v \in V_P$ that contains its VFSM state and virtual input register. $I_P \subseteq S_P$ is the set of *initial process states*, and T_P is a finite set of operations, to be described below. A *processor state* is a collection of all the local states of the processes that belong to a

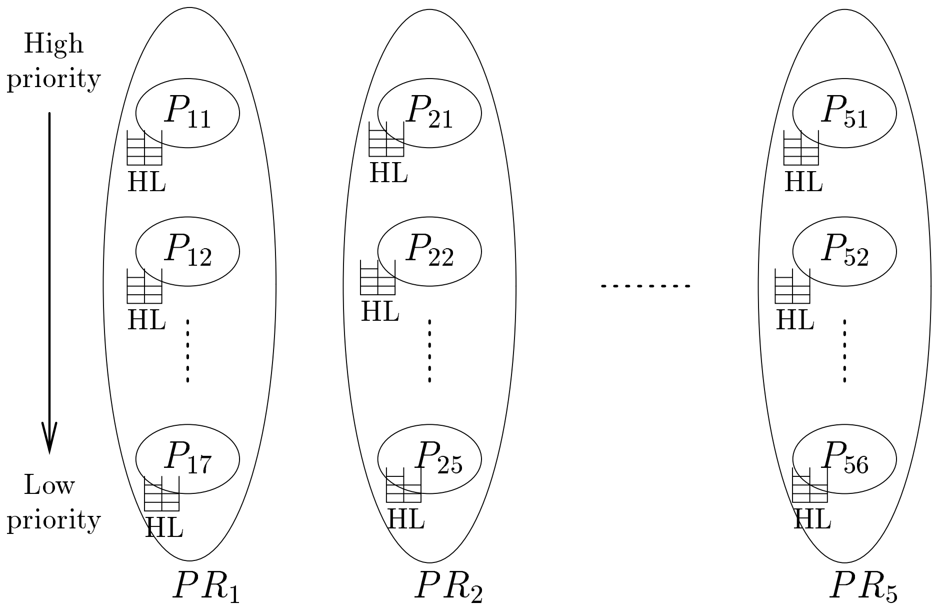

Fig. 3. Process hierarchy in VFSM

processor. A *global state* includes the state of all the processors of the system (note that each timer, as described below, is a separate processor). An *initial global state* is a global state in which every process P of the system is in one of its initial states I_P, and all of the message queues are empty.

Each operation τ of process P consists of the following elements:

- An *entry* process state $s \in S_P$. The operation τ can be executed only if P is in process state s.
- An *exit* state $s' \in S_P$. After executing τ, P is in a process state s'.
- One of the following:

1. A VFSM $v \in V_P$ and a message $m \in M_P$. Then τ is a *receive* operation of the VFSM v, i.e., would be executable only if v receives $v(m)$. Such an operation is denoted as $s \stackrel{\tau:?v(m)}{\longrightarrow} s'$, where s and s' must satisfy one of the following:
   (a) $v(m)$ appears as the first message in $s(H_P)$, i.e., $s(H_P) = v(m).R$. Then $s'(H_P) = R$, i.e., the first message is removed from the head of the high-priority queue of P.
   (b) $s(H_P)$ is empty and $v(m)$ appears as the first message of $s(L_P)$, i.e., $s(L_P) =$

$v(m).R$. Then $s'(L_P) = R$, i.e., the first message is removed from the head of the low-priority queue of P.

2. A triple $\langle Q, v, m\rangle$, where Q is a process, $v \in V_Q$ is a VFSM of Q, and $m \in M_Q$ is a message of Q. Then τ is a *send* operation, and it will send $v(m)$ to process Q. Such an operation is denoted as $s \xrightarrow{\tau:Q!v(m)} s'$. Denote the states of process Q before and after the operation by $\tilde{s}$ and $\tilde{s}'$, respectively. If $P = Q$, then $\tilde{s}'(H_Q) = \tilde{s}(H_Q).v(m)$, i.e., $v(m)$ is appended to the end of H_Q. If $P \neq Q$, then $\tilde{s}'(L_Q) = \tilde{s}(L_Q).v(m)$, i.e., $v(m)$ is appended to the end of L_Q.
3. None of the above. Then, τ is a *local* operation, responsible only for changing the process state. Such an operation is denoted as $s \xrightarrow{\tau:\epsilon} s'$.

A transition is a sequence of operations that can be executed only by the *enabled* VFSM, if any, on a processor. The enabled VFSM is determined by scanning the list of processes in priority order (i.e., from highest to lowest) until a process P is found that satisfies one of the following conditions:

1. Its high-priority queue, H_P, is non-empty; the enabled VFSM is the recipient of the message at the head of H_P.
2. H_P is empty, but its low-priority queue, L_P, is nonempty; the enabled VFSM is the recipient of the message at the head of L_P.
3. H_P and L_P are empty, but P contains a *free* VFSM v that is enabled. Normally, a 5ESS process is event-driven, executing only when it receives a message. A free VFSM is one that can execute even if it does not receive a message. Free VFSMs are typically used to model spontaneous behavior by the environment of a VFSM being validated; for example, the caller in a call-setup protocol can at any time choose to hang up the phone. The enabledness of a free VFSM is a function of its local state, and there can be at most one free VFSM per process.

A transition of an enabled VFSM consists of a receive operation (except in case (3) above), followed by a sequence of zero or more local, send, and timer operations. Roughly speaking, a transition therefore comprises all of the execution steps needed for a VFSM to completely process a single received message. A transition is atomic: no timer expirations or operations of VFSMs on other processors can be interleaved with the operations in a transition.

The execution model we have described differs from most other formal models of concurrent programs in that the granularity of transitions in our model is much larger. In traditional models, e.g., that used in [4], a transition contains at most one access to a shared data structure, and is therefore comparable to an operation in our model. We have chosen to represent execution steps as sequences of operations rather than individual operations because doing so greatly reduces the number of execution sequences that need to be explored during validation. The large granularity of transitions in our model is justified because it is an accurate reflection of the scheduling policy of the 5ESS operating system, which allows a VFSM to perform the processing of a received message as a non-preemptible unit of computation.

B. Timers

The VFSM model also includes *timers*. Each timer is associated with a particular VFSM of a process. A timer that is started will expire after some delay, sending a message to the process containing the VFSM that started it. Each timer W can be modeled as a triple $\langle S_W, T_W, v_W\rangle$ such that there are exactly two states $S_W = \{stopped_W, running_W\}$, three operations $T_W = \{start_W, stop_W, expire_W\}$, and v_W is the VFSM that "owns" the timer, i.e., it is the only VFSM that can start and stop it. The initial state of each timer W is $stopped_W$.

Each VFSM that uses a timer W has also two additional types of operation. The operation $s \overset{\tau:start(W)}{\longrightarrow} s'$ sets the timer W to be *running*. This operation can execute only *jointly* with the operation $start_W$ of timer W. Similarly, the operation $s \overset{\tau:stop(W)}{\longrightarrow} s'$ executes jointly with the $stop_W$ operation of W and places W in state $stopped_W$.

The operation $expire_W$ is enabled when the timer W is in its $running_W$ state. The result is that a message $v(W)$ is appended to the low-priority queue of the process that contains v_W. The timer W itself then returns into its $stopped_W$ state. During an execution of a VFSM system, a running timer can expire at any time. For example, if a VFSM starts a timer during a transition, that timer could expire immediately upon completion of that transition, or other VFSM transitions and/or timer expirations might occur first.

C. Examples

We present in this section some examples to provide concrete illustrations of the abstract model presented above. The first example shows how a VFSM transition is composed of a sequence of operations. Consider the example VFSM in Fig. 1, which is waiting for an acknowledgment message in the VFSM state shown. Suppose that the VFSM receives this message, and that the entry-action section of the state it enters next (`S_SEND_NEXT_MSG`) also sends a message and starts a timer. The sequence of operations the VFSM performs upon receiving the acknowledgment message is as follows:

1. A receive operation to remove the acknowledgment message from its high- or low-priority queue.
2. A local operation in which the virtual input `I_ACK` is inserted into its VIR.
3. A timer operation to stop the running timer, performed upon exiting the state `S_SEND_REQ`.
4. A local operation that models the change in VFSM state that occurs when the VFSM transitions from `S_SEND_REQ` to `S_SEND_NEXT_MSG`.
5. Upon entry to state `S_SEND_NEXT_MSG`, a send operation to send the next message in the handshake-protocol sequence.
6. Also upon entry to state `S_SEND_NEXT_MSG`, a timer operation to await the reply to the next message.

Our next example illustrates the composition of a typical collection of VFSMs in a VFSM system (this example is one of the four on which we perform the validation experiments that will be described later in the paper). The *voice storage system* is a 5ESS feature that allows mobile-phone users to access their voice mail; it does so by issuing commands to the *voice storage equipment* (VSE) hardware that identify the mailbox desired and the type of operation (play, record, or erase) that the user wishes to perform. The system consists of four VFSMs: a "main" VFSM, which implements the functionality of the voice storage system, and three "environment" VFSMs that model the communication partners of the main VFSM. Note that the environment VFSMs were constructed only for validation and do not represent complete 5ESS software modules. Each of the four VFSMs resides on a separate processor; since each process therefore contains only one VFSM, there is no communication on the high-priority queues.

The VFSMs in the system are as follows:

- The *voice storage terminal process* (VTP) VFSM is the "main" VFSM in the system; it acquires a trunk connection from the 5ESS switch to the VSE and coordinates the interaction between them.
- The VSE VFSM models the behavior of the voice storage equipment hardware. It receives messages from the VTP VFSM and generates possible responses.
- The *originating terminal process* (OTP) VFSM models the process that controls the caller end of a telephone call. The VTP reports back to the OTP on the progress of its attempt to establish a connection with the VSE.
- The *terminal maintenance subsystem* (TMS) VFSM models the interaction between the VTP and the fault-detection software in the 5ESS switch. If the trunk connection to the VSE becomes faulty, TMS issues an interrupt message to the VTP that requires it to release control of the trunk so that diagnostics can be performed.

IV. Verification by State-Space Exploration

The global state space A_G of a system can be explored by performing a search of all the states that are reachable from the initial state s_0. This can be done by recursively exploring all successor states of all states encountered during the search, starting from the initial state, by executing all enabled transitions in each state. During the search, visited states are stored in memory in order to avoid redundant explorations of parts of the state space.

The main limit of state-space exploration verification techniques is the often excessive size of the state space. Owing to simple combinatorics, the size can be exponential in the size of the description of the system being analyzed. This exponential growth is known as the *state-explosion problem*.

Several techniques have been proposed to tackle this problem. Among them, bit-state hashing [8] is a simple technique that has proved to be very useful for exploring large state spaces. The principle of this technique is the following. When a new state is visited during the search, its representation is hashed into an address in an array of bits stored in the randomly accessed memory available in the computer on which the state-space exploration is performed. If the bit of the corresponding location in the array is on, the algorithm considers that the state has already been visited. If the bit is off, it is set to on, and the algorithm continues the search from the current state. Since there is no collision detection, it follows that the above search is partial: there is always a possibility that reachable states will be missed. On the other hand, since only one bit of memory is needed to represent the visit of one state, instead of the full state description (which may require several hundreds of bytes of memory), this technique makes it possible to store more states in memory, and hence to explore larger state spaces while still avoiding redundant exploration of parts of the state space. This bit-state hashing technique is used by the VFSM validator.

Another family of techniques that have been developed to cope with the state-explosion problem are *partial-order methods* [14], [3], [16], [6], [12], [17], [19], [13], [4]. The aim of these methods is to avoid the part of the state explosion due to the exploration of all possible interleavings of concurrent transitions. Given a property, partial-order reduction methods explore only a *reduced part* of the global state space that is provably sufficient for verifying the given property. The difference between the reduced and the global state spaces is that not all interleavings of concurrent transitions are systematically represented in the reduced one. Which interleavings need to be preserved may depend on the property to be checked.

The intuition behind partial-order methods is that concurrent executions are really partial orders where concurrent "independent" transitions should be left unordered. Intuitively, transitions are independent when the order of their occurrence is irrelevant.

This notion of independency between transitions and its complementary notion, the notion of dependency, can be formalized by the following definition (adapted from [9]), where $s \xrightarrow{t} s'$ means that the transition t leads from the state s to the state s', while $s \overset{w}{\Rightarrow} s'$ means that the finite sequence of transitions w leads from s to s'.

Definition 1: Let $\mathcal{T}$ be the set of transitions (transitions of VFSMs and expirations of timers) in a VFSM system, and let S be the set of possible global states for this system. The relation $D \subseteq \mathcal{T} \times \mathcal{T} \times S$ is a *valid conditional dependency relation* for the system iff for all $t_1, t_2 \in \mathcal{T}$ and $s \in S$, $(t_1, t_2, s) \notin D$ (t_1 and t_2 are independent in s) implies that $(t_2, t_1, s) \notin D$ (D is symmetric w.r.t. $\mathcal{T} \times \mathcal{T}$) and that the two following properties hold in state s:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other); and
2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \stackrel{t_1t_2}{\Rightarrow} s'$ and $s \stackrel{t_2t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

■

All partial-order methods follow the same basic pattern: they operate as classical state-space searches except that, at each state s reached during the search, they compute a subset T of the set of enabled transitions in s, and explore only the transitions in T; the other enabled transitions are not explored. Such a search is referred to as a *selective search*. Partial-order methods presented in the references cited above differ by the way sets T are computed, and by the type of properties they can verify (see [4] for an extended survey).

Among all these algorithms, *persistent sets* were shown in [4] to provide an abstract characterization of a whole family of existing algorithms [11], [17], [5] for computing such sets T. The notion of persistent set is very similar to the notion of *faithful decomposition* introduced (independently) in [10] and to the notion of ample set [12]. We will use persistent sets in what follows. Intuitively, a subset T of the set of transitions enabled in a state s of A_G is called *persistent in* s if all transitions not in T that are enabled in s, or in a state reachable from s through transitions not in T, are independent with all transitions in T. In other words, whatever one does from s, while remaining outside of T, does not interact with or affect T. Following the definition of [5], we have:

Definition 2: A set T of transitions enabled in a state s is *persistent in* s iff, for all nonempty sequences of transitions

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

from s in A_G and including only transitions $t_i \notin T$, for $1 \leq i \leq n$, t_n is independent in s_n with all transitions in T. ■

Note that the set of all enabled transitions in a state s is trivially persistent since nothing is reachable from s by transitions that are not in this set.

```
Initialize:Stack is empty; H is empty;
           push (s_0) onto Stack;
Loop: while Stack ≠ ∅ do {
           pop (s) from Stack;
           if s is NOT already in H then {
              enter s in H;
              T = Persistent_Set(s);
              for all t in T do {
                  s' = succ(s) after t; /* t is executed */
                  push (s') onto Stack;
                  }
              }
           }
```

Fig. 4. Persistent-set selective search

Let a *persistent-set selective search* be a selective search through A_G which, in each state s that it reaches, explores only a set T of enabled transitions that is persistent in s, and that is nonempty if there exist transitions enabled in s. Such an algorithm is illustrated in Fig. 4. It can be shown that such a search reaches all deadlock states of A_G (e.g., see [4]).

V. Computing Persistent Sets for VFSM Systems

The key element required for the implementation of a persistent-set selective search is an algorithm for computing persistent sets. The algorithms that have been proposed for this purpose [11], [10], [17], [5], [12] infer the persistent sets from the static structure (code) of the program being verified. They differ from each other by the type of information about the program that they use. The aim of these algorithms is to obtain the smallest possible persistent sets. Usually, the more information about the program the algorithm uses, the smaller the persistent set it produces can be, albeit at the cost of a higher computational complexity. Note that exploring the smallest number of enabled transitions at each step of the search is only a heuristic: it does not necessarily lead to the exploration of the smallest number of states.

An algorithm for computing persistent sets in the context of VFSM systems is presented

---

1. Take one transition t that is enabled in s. Let $\mathcal{P}_i$ be the processor of t, and let $P = \{\mathcal{P}_i\}$.

2. For all processors $\mathcal{P}_i$ in P, add to P all processors $\mathcal{P}_j$ such that

(a) $\mathcal{P}_j$ contains a transition t' that is *potentially reachable* from the current state s, and that can perform a *Send* operation on a queue in $Sensitive(\mathcal{P}_i, s)$; or

(b) $\mathcal{P}_j$ is in $may\_be\_affected\_by(enabled(\mathcal{P}_i, s))$; or

(c) if $\mathcal{P}_i$ is a processor of type "timer" and is running, $\mathcal{P}_j$ contains a transition t' that is *potentially reachable* from the current state s, and that can stop this timer; or

(d) if $\mathcal{P}_i$ is a processor of type "timer" and is stopped, $\mathcal{P}_j$ contains a transition t' that is *potentially reachable* from the current state s, and that can start this timer.

3. Repeat step 2 until no more processors need be added. Then, return all transitions t that are in processors in P and that are enabled in s.

Fig. 5. Computing persistent sets for VFSM systems

in Fig. 5. This algorithm starts by considering a processor or a timer that contains a transition enabled in state s. (In what follows, timers are considered as processors of a special type.) This processor or timer is introduced in set P (step 1). Then, all processors or timers that contain at least one transition that "might interfere with" the initial enabled transition are included in set P (step 2). This relation "might interfere with" is further discussed below. Step 2 is repeated until no more processors need be added to P (step 3). Finally, all enabled transitions contained in a processor or timer in set P are returned.

The purpose of step 2 of the algorithm is to ensure that all processors not in set P do not contain transitions that are or could become *dependent* with the enabled transitions of the processors in set P (cf. Definition 2 above). Let us consider first the case where there are no timers in the system to be analyzed. In that case, steps 2.c and 2.d can be ignored.

Step 2.a of the algorithm makes use of the function $Sensitive(\mathcal{P}_i, s)$, which is defined as follows. Let $enabled(\mathcal{P}_i, s)$ denote the transition of processor $\mathcal{P}_i$ that is enabled in s, if there exists one. Given a processor $\mathcal{P}_i$ and a global state s, the function $Sensitive(\mathcal{P}_i, s)$ returns the set of low-priority queues of processor $\mathcal{P}_i$ such that sending a message to one of these queues *might change the value of* $enabled(\mathcal{P}_i, s)$, or *might change the effect of* the execution of the transition in $enabled(\mathcal{P}_i, s)$, if there is one. (A transition that sends a message to a queue in $Sensitive(\mathcal{P}_i, s)$ might be dependent with the transition in $enabled(\mathcal{P}_i, s)$.)

We now describe how to compute $Sensitive(\mathcal{P}_i, s)$. Let $sends\_on\_queues\_by(t)$ denote the set of queues on which transition t can perform a $Send$ operation, and let $next\_active\_process(\mathcal{P}_i, s)$ denote the process in processor $\mathcal{P}_i$ that contains the transition $enabled(\mathcal{P}_i, s)$ (if $enabled(\mathcal{P}_i, s)$ is empty, $next\_active\_process(\mathcal{P}_i, s)$ is empty as well). Then, let $empty\_ordered\_queues(\mathcal{P}_i, s)$ be the set of queues that are associated with processes in $\mathcal{P}_i$ that are of higher priority than $next\_active\_process(\mathcal{P}_i, s)$ (if $next\_active\_process(\mathcal{P}_i, s)$ is empty, $empty\_ordered\_queues(\mathcal{P}_i, s)$ is the set of all the queues in processor $\mathcal{P}_i$). If $enabled(\mathcal{P}_i, s)$ is a transition of a free VFSM and if both queues of $next\_active\_process(\mathcal{P}_i, s)$ are empty in state s, then $empty\_ordered\_queues(\mathcal{P}_i, s)$ also contains the two queues associated with $next\_active\_process(\mathcal{P}_i, s)$. Then, $Sensitive(\mathcal{P}_i, s)$ is the union of $empty\_ordered\_queues(\mathcal{P}_i, s)$ and of $sends\_on\_queues\_by(enabled(\mathcal{P}_i, s))$.

Step 2.b of the algorithm uses the function $may\_be\_affected\_by(enabled(\mathcal{P}_i, s))$, which returns the set of processors that contain a transition whose execution may be affected by the execution of the transition in $enabled(\mathcal{P}_i, s)$. Precisely, $may\_be\_affected\_by(enabled(\mathcal{P}_i, s))$ contains the processors $\mathcal{P}_j$ that have a queue q in $sends\_on\_queues\_by(enabled(\mathcal{P}_i, s))$ and such that either q is not the queue of lowest priority in $\mathcal{P}_j$ or the process of lowest priority in $\mathcal{P}_j$ contains a free VFSM.

Let us now consider the general case where there are timers in the system. In that case, interactions between processors of type "timer" and the other processors must be taken into account as well. This is done in steps 2.c and 2.d of the algorithm of Fig. 5. Moreover, if $\mathcal{P}_i$ is of type "timer," then $Sensitive(\mathcal{P}_i, s)$ is defined as follows: if $\mathcal{P}_i$ is running, $Sensitive(\mathcal{P}_i, s) = \{L_{(timer)}\}$, where $L_{(timer)}$ denotes the low-priority queue of the process that has started the timer; otherwise, we have $Sensitive(\mathcal{P}_i, s) = \emptyset$. Finally, in the presence of timers, $may\_be\_affected\_by(enabled(\mathcal{P}_i, s))$ also contains all the processors of type "timer" that are running and that can be stopped by $enabled(\mathcal{P}_i, s)$.

By step 2 of the algorithm of Fig. 5, sets of transitions t' that are *potentially reachable* from the current global state s and that can perform either a $Send$ operation (step 2.a) or specific operations on timers (step 2.c and 2.d) need to be determined. Obviously, an exact determination of these sets of transitions is as difficult as exploring the whole state space of the system. However, supersets, i.e., conservative approximations, of these sets of transitions t' can be obtained by statically analyzing the state-transition graphs of each VFSM individually: for each VFSM, the set of all operations of a given type that are statically reachable in the state-transition graph of this VFSM from any given *local* state of the VFSM can be determined at compile-time, before the state-space exploration is performed. This information can then be used during state-space exploration to approximate the sets of transitions t' reachable from the current state that can perform operations of specific types. Two algorithms for computing such approximations are discussed in the next section.

We now prove the correctness of the algorithm of Fig. 5.

Theorem 1: Any set of transitions that is returned by the algorithm of Fig. 5 is a persistent set in the current state s.

Proof

Let T be a set of transitions that is returned by the algorithm of Fig. 5, and let P denote the final set of processors computed by step 2 of the algorithm during this run.

The proof is by contradiction. Suppose that T is not persistent in s. Then, by Definition 2, there exists in A_G a sequence $s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ of transitions $t_1, t_2, \ldots, t_n \notin T$ such that t_n is dependent in s_n with some transition $t \in T$. Consider the shortest such sequence. For this sequence, not only is t_n dependent in s_n with some transition $t \in T$, but also, for all $1 \leq i < n$, t_i is independent in s_i with all transitions in T. Let us show that such a sequence cannot exist.

Let $\mathcal{P}$ denote the processor that contains transition t, and let $\mathcal{P}_n$ be the processor that contains transition t_n. Since for all transitions t_i, $1 \leq i < n$, t_i is independent in s_i with all transitions in T, t and t_n are both enabled in s_n. Since t and t_n are dependent in s_n, three cases are possible.

1. After the execution of t_n in s_n, t becomes disabled in s_{n+1}. If $\mathcal{P}$ is not of type "timer," this can happen only if t_n sends a message to a queue in $empty\_ordered\_queues(\mathcal{P}, s_n)$. Since $empty\_ordered\_queues(\mathcal{P}, s_n) = empty\_ordered\_queues(\mathcal{P}, s)$ and since $empty\_ordered\_queues(\mathcal{P}, s) \subseteq Sensitive(\mathcal{P}, s)$, t_n is a transition "t'" satisfying the conditions stated in step 2.a of the algorithm, and $\mathcal{P}_n$ is included in set P. If $\mathcal{P}$ is of type "timer," then it is running in state s_n, as well as in state s, and is stopped by t_n. By

step 2.c, $\mathcal{P}_n$ is included in set P.

2. After the execution of t in s_n, t_n becomes disabled in the state s' such that $s_n \xrightarrow{t} s'$. If $\mathcal{P}_n$ is not of type "timer," this can happen only if t sends a message to a queue in *empty_ordered_queues*$(\mathcal{P}_n, s_n)$. Note that, since t_n is enabled in s_n, *empty_ordered_queues*$(\mathcal{P}_n, s_n)$ does not contain the queue of lowest priority in processor $\mathcal{P}_n$ if the process of lowest priority of $\mathcal{P}_n$ does not contain a free VFSM. Therefore, $\mathcal{P}_n$ is in *may_be_affected_by*(*enabled*$(\mathcal{P}), s)$, and is included in set P by step 2.b of the algorithm. If $\mathcal{P}_n$ is of type "timer," then it is running in s_n, and is stopped by t. Consequently, $\mathcal{P}_n$ is in *may_be_affected_by*(*enabled*$(\mathcal{P}), s)$, and is included in set P by step 2.b of the algorithm.

3. The two sequences of transitions tt_n and t_nt are both executable from s_n, but their executions do not lead to the same global state. This means that both t and t_n send at least one message to the same queue, i.e., a queue in *sends_on_queues_by*(*enabled*$(\mathcal{P}, s_n))$. Since *sends_on_queues_by*(*enabled*$(\mathcal{P}, s_n))$ = *sends_on_queues_by*(*enabled*$(\mathcal{P}, s))$, and since *sends_on_queues_by*(*enabled*$(\mathcal{P}, s)) \subseteq$ *Sensitive*$(\mathcal{P}, s)$, $\mathcal{P}_n$ is included in set P by step 2.a of the algorithm.

In all cases, we conclude that $\mathcal{P}_n$ is in set P. Let t_k be the first transition in the sequence $t_1t_2\ldots t_n$ that is contained in processor $\mathcal{P}_n$. If t_k is enabled in s, then t_k is in the set T returned by the algorithm, which contradicts the assumption that $t_i \notin T$, for $1 \leq i \leq n$. Therefore, t_k is disabled in s.

Since t_k is disabled in s and enabled in s_k, let t_l be the first transition in the sequence $t_1t_2\ldots t_{k-1}$ such that t_k is disabled in s_l and enabled in s_{l+1}. (In other words, t_l and t_k are dependent in s_l.) By construction, we know that t_l is in a processor other than $\mathcal{P}_n$.

If $\mathcal{P}_n$ is not a processor of type "timer," then t_l can make t_k enabled in s_{l+1} only by sending a message to a queue in *empty_ordered_queues*$(\mathcal{P}_n, s_l)$. Since no transitions of $\mathcal{P}_n$ are executed from s to s_l, we have *empty_ordered_queues*$(\mathcal{P}_n, s_l) \subseteq$ *empty_ordered_queues*$(\mathcal{P}_n, s)$. Since *empty_ordered_queues*$(\mathcal{P}_n, s) \subseteq$ *Sensitive*$(\mathcal{P}_n, s)$, t_l is a transition "t'" satisfying the conditions stated in step 2.a of the algorithm, and the processor of t_l is included in set P.

If $\mathcal{P}_n$ is a processor of type "timer," then it is stopped in state s_l, as well as in state s, and t_l starts it. By step 2.d of the algorithm, the processor of t_l is included in set P.

In summary, in both cases, there exists a transition t_l, with $1 \leq l < k$, such that the processor of t_l is in P. If t_l is enabled in s, it is returned by the algorithm and is thus in T, which contradicts the assumption that $t_l \notin T$. Therefore, t_l is disabled in s.

By repeating the same reasoning, one comes to the conclusion that the processor of t_1 is in P. Since t_1 is enabled in s, this means that $t_1 \in T$, which contradicts the assumption that $t_1, \ldots, t_n \notin T$.

■

We have implemented a persistent-set selective search using the algorithm of Fig. 5. Whenever a new state s is visited during the search, the smallest persistent set that can be computed by the algorithm of Fig. 5 is computed. This is done by executing the algorithm of Fig. 5 with every transition enabled in s as the initial transition t taken in step 1 of the algorithm, and then selecting the smallest returned set.[1] Then, only the transitions in the smallest persistent set are explored.

VI. Determining VFSM Queue and Timer Operations

The algorithm in Fig. 5 requires knowledge of the queue and timer operations performed by each VFSM in the system. The knowledge used by the algorithm is of two types:

- "Long-term" knowledge about the operations that each VFSM might perform at some point in an execution sequence beginning at the current global state. This knowledge is required for the VFSM transitions in Fig. 5 that are "potentially reachable" from the current global state.
- "Short-term" knowledge about the operations that the enabled VFSM, if any, of a processor will perform in its next transition only. This knowledge is needed for calculation of the set $Sensitive(\mathcal{P}_i, s)$ in Fig. 5.

In order to explore the tradeoff between the accuracy of this knowledge and the cost of obtaining it, we used three different approaches to its calculation. The first two approaches, described in Sections VI.A and VI.B, employ static analysis to compute both the long- and short-term knowledge just mentioned. The third approach, described in Section VI.C,

[1] Optimizations like those described in [14], [15], [4] for avoiding redundant work during successive executions of the algorithm of Fig. 5 when searching for a minimal persistent set have also been implemented.

uses a more accurate, but more costly, means of obtaining the short-term knowledge that involves executing the enabled VFSMs in the current global state to determine the operations they will perform.

A. Per-VFSM Information

The first approach is to determine all the operations a VFSM might possibly perform in its lifetime. Fig. 6 shows the state-transition graph of a VFSM; each node of the graph is a VFSM state, defined by a textual representation like that in Fig. 1, and there is an edge of the graph for each next-state (`NS:`) transition. By performing a static analysis of the textual representation of each state, we can determine the operations the VFSM might perform in that state; these are denoted by the first (bold-faced) labels next to each state in Fig. 6. Taking the union of the sets labeling each state, we see that the VFSM in this example can perform send operations on queues 1, 2 and 3, and can start and stop timer 1. We refer to this information as *per-VFSM* information, since there is one set of data computed for each VFSM.

B. Per-State Information

The information computed on a per-VFSM basis clearly overestimates the operations a VFSM might perform in its lifetime. For example, if the VFSM in Fig. 6 is in state $S2$, then it can never send on queue 1 or start timer 1 again: these operations are performed only in state $S0$, which is not reachable from $S2$. Our second approach to calculating the information needed for the partial-order algorithm is to determine, for each VFSM state, the operations that can be performed in that state or in states reachable from it. We refer to this as *per-state* information, since a set of operations is associated with each VFSM state. In Fig. 6, the per-state information is represented by the second set of labels next to each state. Since all other states are reachable from $S0$, the per-VFSM and per-state information are the same in this state; in states $S1$, $S2$, and $S3$, however, the per-state information is strictly "better" than the per-VFSM information, so the use of per-state information may result in the computation of smaller persistent sets. Note that the effectiveness of per-state information depends on the structure of the state-transition graph. If, as in Fig. 6, the graph is acyclic, then per-state information can yield much

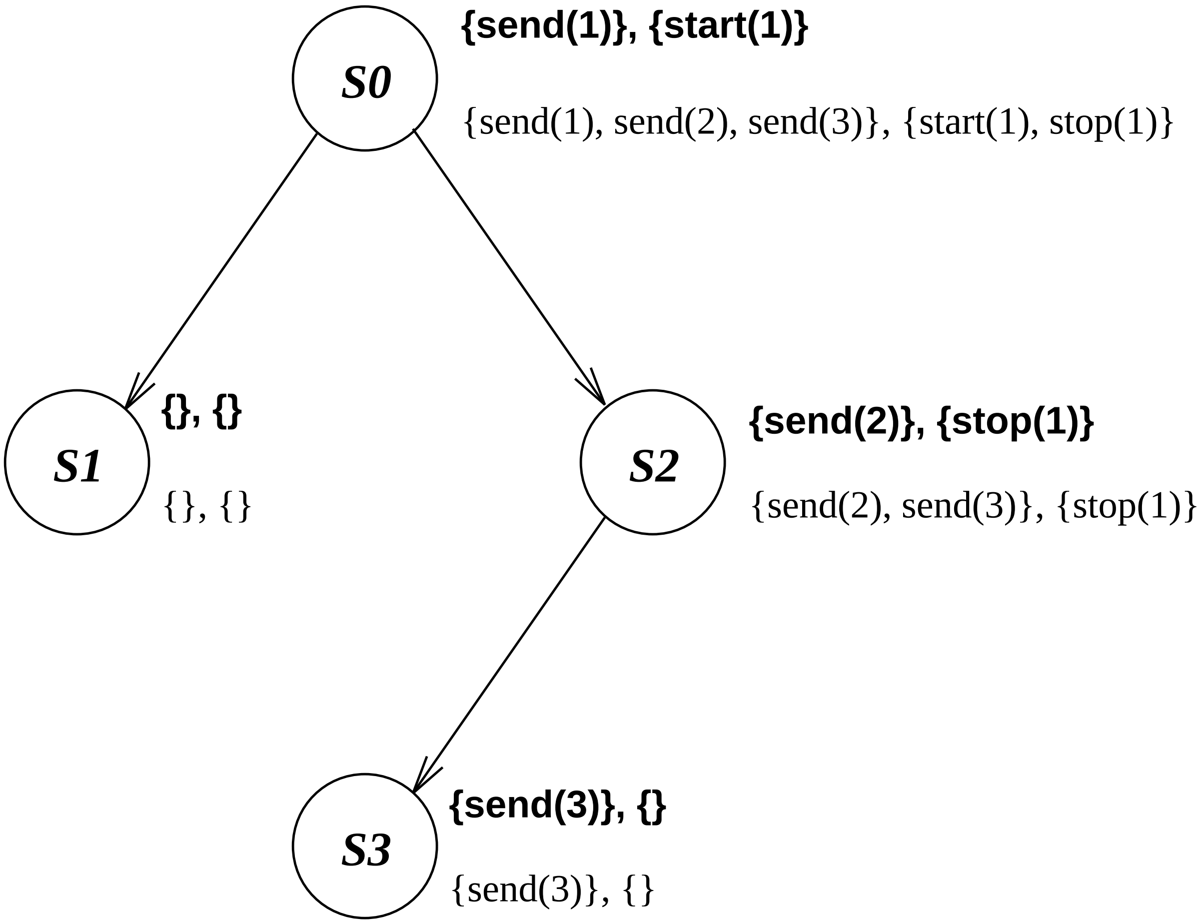

Fig. 6. Example VFSM state transition graph

benefit; however, if the graph is strongly connected, so that each state is reachable from every other state, then the per-VFSM and per-state information would be identical in all states.

C. Run-time Information

When considering the enabled VFSM of a processor, both the per-state and per-VFSM information define the operations it *might* perform in its next transition, not which operations it actually *will* perform, which depends on the message it will receive next. In the example in Fig. 6, the VFSM may start in $S0$ and end in state $S1$ when processing a given message, but both types of information described so far would erroneously assume that the VFSM sends messages on queues 2 and 3 and stops timer 1. Use of per-state or per-VFSM information to compute the short-term knowledge needed by the algorithm in

Fig. 5 can therefore result in a value for the set $Sensitive(\mathcal{P}_i, s)$ that is too large, limiting the reduction obtained by the partial-order algorithm.

Unfortunately, it is difficult to predict, using only static analysis, the operations an enabled VFSM will perform in response to a given message. However, it is possible to execute an instrumented version of the VFSM and record the operations it performs in the course of processing a received message, and then use this information in the persistent-set algorithm. We refer to this third type of information as *run-time* information, because it makes use of knowledge obtained during the execution of validation, as opposed to the static, *a priori* information used by the per-state and per-VFSM approaches.

We implemented a special "shadow" mode of VFSM execution for the purpose of collecting run-time information. When a VFSM is executed in normal mode during validation, it performs local, timer and queue operations on the current global state, resulting in a new global state (see step 9 in the validation algorithm in Fig. 4). Execution in "shadow" mode is exactly the same, except that no changes to the current global state are made: the VFSM simply records information about each queue and timer operation it encounters during the "shadow" execution, but does not actually perform the operations. Upon completion of the "shadow"-mode execution, we therefore know exactly which operations the VFSM will perform when executed in the current global state, and we can use this information to compute a more accurate value for the set $Sensitive(\mathcal{P}_i, s)$.

To summarize, validation with the partial-order algorithm using run-time information proceeds from a particular global state as follows:

1. Each enabled VFSM is executed in "shadow" mode to determine the queue and timer operations it will perform.
2. The information collected from the "shadow"-mode executions is employed in the calculation of $Sensitive(\mathcal{P}_i, s)$ by the algorithm in Fig. 5.
3. Each enabled transition in the persistent set that results is executed (in normal mode) to produce a new global state (steps 8-11 in the algorithm in Fig. 4).

The use of "shadow"-mode executions would seem to defeat the purpose of the partial-order algorithm, which is to compute a persistent set that is strictly smaller than the set of all enabled transitions and thereby avoid executing all of them. Note from the above discussion that each VFSM transition in the persistent set is in fact executed twice, once in "shadow" mode and once in normal mode. However, if the persistent sets that result are smaller than those computed with per-VFSM or per-state information, then the number of global states explored might also be smaller. Since the persistent-set calculation is carried out once for each global state encountered, the net reduction that results from exploring fewer states may be sufficient to compensate for the added overhead of performing "shadow"-mode executions in each global state.

Run-time information can be computed and used only for the enabled transition, if any, of each processor when considering the other transitions in the system that might be dependent with it; it cannot be used to calculate the long-term knowledge needed by the algorithm in Fig. 5. We have therefore used per-state information for the long-term knowledge about the operations that the VFSMs might perform in the "potentially reachable" transitions in Fig. 5, along with run-time information for the short-term knowledge about the enabled VFSM needed to calculate $Sensitive(\mathcal{P}_i, s)$; for simplicity, we refer to this combination as "run-time information" below.

VII. Experimental Results

To determine the reductions that can be obtained with the partial-order algorithm, we selected four VFSM examples that represent a cross-section of the application domains in which VFSM has been used. The *Voice Storage* example, which was described in Section III, implements a feature that allows mobile-phone users to access their voice mail. The *MFC signalling* example is a protocol for transmitting the digits of a telephone number over an interoffice trunk line. This example consists of three VFSMs, each of which is on a separate processor. The *CD Remove/Restore* example is a part of the 5ESS hardware maintenance software; it is responsible for removing and restoring groups of trunk lines. It consists of four VFSMs, three of which are in the same process (the fourth is on a separate processor). The *Speech Handler* example is a protocol for setting up a connection and transferring speech data between the 5ESS switch and a cellular relay site. It consists of six VFSMs distributed among three processors.

Table I presents the results of validating the four examples described above[2]. For each

[2]The results in Table I differ significantly from those presented in the initial version of this paper [7] because

example, we provide four sets of numbers: the first is for a normal validation run, and the last three are for runs with the partial-order algorithm enabled using either per-VFSM, per-state or run-time information about the operations performed by each VFSM. "States" gives the number of unique global states explored. "Matches" gives the number of times the validator determined, using the bitstate-hashing technique, that a global state it produced had been explored previously. "Transitions" is the number of state transitions explored, which is a good indicator of the total effort expended during validation; it is equal to the sum of "States" and "Matches". "Time" is the running time of each validation run in seconds ("user" time obtained with the Unix time command; all runs were performed on a SparcStation 20 with 190 MB of main memory). For all examples, a bitstate hash array of 128 MB, or over one billion bits, was used. The ratio of bits in the hash array to global states ranges from 60 to over 10,000, so the likelihood of hash collisions is reasonably small.

As the table shows, the partial-order algorithm achieves a reduction in the number of transitions explored in all cases; the reductions range from 13% for the CD Remove/Restore example to more than a factor of 5 for the Speech Handler example. The algorithm obtains the smallest reductions on the CD Remove/Restore example. This is due to its small amount of concurrency: there is little message communication between the two processors in the system; most interaction occurs among the three VFSMs in the same process, at most one of which is enabled at any given time. One of these VFSMs can start and stop timers, resulting in dependency whenever a timer is running and this VFSM is enabled. Hence, the partial-order algorithm is seldom able to find a persistent set that is smaller than the set of all enabled transitions.

The Voice Storage and MFC Signalling examples are both part of the 5ESS call-processing software, and they have a similar structure: each consists of a "main" VFSM and two or more "environment VFSMs" that exchange messages with it, but not with each other. All of the VFSMs in both examples have state-transition graphs that are acyclic, which explains the significant reductions obtained with per-state information relative to per-VFSM information. Run-time information yields additional reduction in both cases.

For the Speech Handler example, a substantial reduction is obtained only by using run-

condition 2.b in the algorithm in Fig. 5 was incorrectly omitted from the algorithm of [7].

time information. The reason is that there is quite a bit of interaction among the six VFSMs in this example; because four of the VFSMs have state-transition graphs that are strongly connected, both the per-VFSM and per-state information severely overestimate the queue and timer operations these VFSMs actually perform in a given transition.

The running times given in Table I provide information about the computational overhead introduced by partial-order methods. Comparing the running times for the Speech Handler example for the normal validation run and the run with the partial-order algorithm using per-VFSM information, we see that the latter running time is much greater, even though slightly fewer transitions are explored; this is due to the overhead added by the calculation of persistent sets. Comparison of the per-state and run-time experiments for the MFC Signalling example illustrates the overhead introduced by execution of all enabled transitions: the run-time information reduces the number of transitions explored substantially relative to per-state information, but the running times are nearly identical. This suggests that the use of run-time information must reduce the number of transitions executed by about 40%–50% relative to per-state information before any reduction in execution time will be obtained.

VIII. Conclusions

We have presented a partial-order algorithm for a validation tool that models the concurrent execution of a collection of VFSMs executing under the control of the 5ESS operating system. The success of the algorithm on a given example is very much a function of the structure of its VFSMs. For example, the acyclic structure of the VFSMs in the MFC Signalling and Voice Storage examples led to great reductions when per-state information was used. Both examples are processes that are created to handle the set-up of a telephone call and terminated when the call ends; termination is represented by "end states" in their VFSMs that have no successors. This structure is somewhat surprising: concurrent programs are sometimes distinguished from their sequential counterparts by the fact that the latter are terminating, while the former are non-terminating. We have found that *terminating* programs like these, which exhibit all the characteristics of a concurrent program during their lifetimes, are common in our application domain.

Partial-order methods improve the effectiveness of validation in two ways. The first is

TABLE I

Results of Partial-Order Algorithm

Example	Algorithm	States	Matches	Transitions	Time
Speech Handler	normal	8,650,617	16,126,440	24,777,057	1,729s
	PO (per-VFSM)	8,624,565	15,581,858	24,206,423	2,662s
	PO (per-state)	8,264,211	14,275,789	22,540,000	2,524s
	PO (run-time)	2,399,136	1,920,942	4,320,078	721s
Voice Storage	normal	485,666	672,370	1,158,036	100s
	PO (per-VFSM)	455,535	567,534	1,023,069	121s
	PO (per-state)	205,108	127,140	332,248	39s
	PO (run-time)	153,453	79,150	232,603	54s
CD Rmv/Rst	normal	626,613	2,873,113	3,499,726	398s
	PO (per-VFSM)	590,406	2,729,769	3,320,175	414s
	PO (per-state)	505,511	2,650,478	3,155,989	403s
	PO (run-time)	505,119	2,541,350	3,046,469	646s
MFC signalling	normal	17,587,557	44,106,215	61,693,772	3,413s
	PO (per-VFSM)	17,478,778	43,152,269	60,631,047	4,304s
	PO (per-state)	11,911,066	24,704,227	36,615,293	2,735s
	PO (run-time)	7,566,598	12,265,716	19,832,314	2,698s

that they decrease the amount of time needed to complete a validation run, allowing more runs to be completed in a given time period. The second is that, because the number of global states explored is usually much less with partial-order methods, the likelihood of a collision in the bitstate hash array is also less, increasing one's confidence that no errors have been missed.

Comparing the three approaches for determining the queue and timer operations a VFSM can perform, we can conclude that per-state information is a clear winner relative to per-VFSM information: the former requires only slightly more memory and processor overhead than the latter, but can yield significantly greater reductions. Run-time information often results in substantial reduction in the number of global states and transitions but may increase the overall running time.

Use of the partial-order algorithm described in this paper results in the exploration of a reduced state space that contains all the deadlocks of the complete state space; however, the reduced state space may not contain other kinds of errors, such as unexpected inputs and livelocks. We plan to modify our tool so that it checks for violations of application-specific correctness conditions specified in temporal logic. A slight modification to the partial-order algorithm will guarantee that the reduced state space contains all violations of safety and liveness properties.

References

[1] A. R. Flora-Holmquist and M. G. Staskauskas, "Formal validation of virtual finite state machines," *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT95)*, Boca Raton, FL, pp. 122-129, April 1995.

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

[3] P. Godefroid, "Using partial orders to improve automatic verification methods," *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, Rutgers, June 1990. Springer-Verlag. Extended version in ACM/AMS DIMACS Series, volume 3, pages 321–340, 1991.

[4] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*, Springer-Verlag, January 1996.

[5] P. Godefroid and D. Pirottin, "Refining dependencies improves partial-order verification methods," *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449, Elounda, June 1993. Springer-Verlag.

[6] P. Godefroid and P. Wolper, "Using partial orders for the efficient verification of deadlock freedom and safety properties," *Formal Methods in System Design*, 2(2):149–164, April 1993.

[7] P. Godefroid, D. Peled, and M. Staskauskas, "Using partial-order methods in the formal validation of industrial concurrent programs," *Proc. Intl. Symposium on Software Testing and Analysis*, San Diego, CA, January 1996, pages 261-269.

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

[9] S. Katz and D. Peled, "Defining conditional independence using collapses," *Theoretical Computer Science*, 101:337–359, 1992.

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

[11] W. T. Overman, *Verification of Concurrent Systems: Function and Timing*, PhD thesis, University of California Los Angeles, 1981.

[12] D. Peled, "All from one, one for all: on model checking using representatives," *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, Elounda, June 1993. Springer-Verlag.

[13] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol. 8, pp. 39-64, 1996.

[14] A. Valmari, "Error detection by reduced reachability graph generation," *Proc. 9th International Conference on Application and Theory of Petri Nets*, pages 95–112, Venice, 1988.

[15] A. Valmari, "Heuristics for lazy state generation speeds up analysis of concurrent systems," *Proc. of the Finnish Artificial Intelligence Symposium STeP-88*, volume 2, pages 640–650, Helsinki, 1988.

[16] A. Valmari, "A stubborn attack on state explosion," *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165, Rutgers, June 1990. Springer-Verlag.

[17] A. Valmari, "On-the-fly verification with stubborn sets," *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408, Elounda, June 1993. Springer-Verlag.

[18] F. Wagner, "VFSM executable specification," *Proc. IEEE International Conference on Computer System and Software Engineering*, pages 226-231, The Hague, 1992.

[19] P. Wolper and P. Godefroid, "Partial-order methods for temporal verification (invited paper)," *Proc. CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, Hildesheim, August 1993. Springer-Verlag.

Patrice Godefroid received the B.S. degree in electrical engineering (computer science elective) in 1989, and the Ph.D. degree in computer science in 1994, both from the University of Liège, Belgium. Since 1994, he has been a member of the technical staff in the Software Production Research Department at Bell Laboratories, Naperville, IL. His research interests include verification and testing of communication protocols, software analysis tools, and design methodologies for concurrent reactive systems.

Doron Peled received his B.Sc., M.Sc. and D.Sc. degrees in computer science from the Technion, Israel Institute of Technology, in 1984, 1987 and 1991, respectively. Between the years 1987 and 1991 he also did his military service. In 1991–1992, he was visiting the University of Warwick, England for a post-doctoral year. Since 1992 he has been a member of technical staff at Bell Laboratories, Murray Hill, New Jersey. Dr. Peled is interested in specification and verification of concurrent systems, formal semantics of programming languages, automata theory, and mathematical logic.

Mark Staskauskas received the B.S. degree in computer science from Columbia University in 1979, the M.S. degree in computer science from the University of California, Los Angeles, in 1981, and the Ph.D. degree in computer science from the University of Texas at Austin in 1992. From 1981 to 1984, he was with M/A-COM Linkabit, Inc., San Diego, CA, where he participated in a number of network protocol implementations. While a student at the University of Texas, he was employed by Bull HN Information Systems, Inc., Phoenix, AZ, and MCC, Austin, TX, in a variety of projects involving the application of formal methods to real-world hardware and software design problems. He is presently a member of the technical staff in the Software Production Research Department at Bell Laboratories, Naperville, IL. His research interests include formal methods for concurrent programs, software testing, and software engineering. Dr. Staskauskas is a member of Tau Beta Pi and Upsilon Pi Epsilon.