A Framework for Evaluating Specification Methods for Reactive Systems

Mark A. Ardis
John A. Chaves
Lalita Jategaonkar Jagadeesan
Peter Mataga
Carlos Puchol
Mark G. Staskauskas
James Von Olnhausen

June 1996

Appeared in the *IEEE Transactions on Software Engineering*, 22(6):378-389, June 1996.

A Framework for Evaluating Specification Methods for Reactive Systems

Mark A. Ardis, John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, James Von Olnhausen
Bell Laboratories
Naperville, IL 60566-7013 (USA)

Abstract

Numerous formal specification methods for reactive systems have been proposed in the literature. Because the significant differences between the methods are hard to determine, choosing the best method for a particular application can be difficult. We have applied several different methods, including Modechart, VFSM, Esterel, Basic LOTOS, Z, SDL and C, to an application problem encountered in the design of software for AT&T's 5ESS® telephone switching system. We have developed a set of criteria for evaluating and comparing the different specification methods. We argue that the evaluation of a method must take into account not only academic concerns, but also the maturity of the method, its compatibility with the existing software development process and system execution environment, and its suitability for the chosen application domain.

1 Introduction

A reactive system is one that must continually respond to stimuli from its environment. Because of the enormous number of possible sequences of stimuli that it must handle, the design of a reactive system is notoriously difficult: since testing can exercise only a small fraction of possible execution scenarios of the system, subtle errors often escape detection and can cause havoc in the implemented system.

Formal methods represent a promising approach for obtaining confidence in the correctness of a reactive system. A formal methodology for reactive-system design should include a *requirements language* for specifying the properties to be met by the system; a *modeling language* for building a high-level prototype of an abstract algorithm for the reactive system that the user can interactively simulate and then refine into an implementation; and a *verification method* that allows the user to formally prove that an abstract algorithm written in the modeling language satisfies properties specified in the requirements language.

Reactive systems are ubiquitous in the software for AT&T's 5ESS® telephone switching system [1]. For example, the software for establishing and tearing down telephone calls must correctly handle each of the possible sequences of events that can be entered by the caller and callee in a telephone call. Recently, we have been investigating various specification methods for reactive systems that might be useful in the 5ESS software development environment. In order for us to better understand the relative strengths and weaknesses of the methods, we have applied each of them to the design of a simple protocol drawn from an actual 5ESS application. The protocol maintains a redundant communication channel consisting of two lines, at most one of which can be selected for communication at any time. The problem statement consists of a set of requirements stated in English that define how operator commands and line-state changes affect line selection. For each of the specification methods studied, we attempted to specify the protocol requirements, construct an abstract algorithm in the method's modeling language, and then verify that the algorithm satisfies the requirements.

We selected for evaluation the specification methods Modechart[2], VFSM[3], Esterel[4], Basic LOTOS[5] , **Z**[6], and SDL[7], and the programming language C (the latter was chosen as a point of contrast, since it is the primary language in which the 5ESS software is implemented). We chose these par-

[1]M.A. Ardis, J.A. Chaves, L.J. Jagadeesan, P. Mataga, M.G. Staskauskas, and J. Von Olnhausen are with Bell Laboratories, 1000 E. Warrenville Rd., Naperville, IL 60566. Email: maa@bell-labs.com.

[2]C. Puchol is with the Department of Computer Sciences, the University of Texas at Austin, Austin, TX 78712. The work described here was performed while he was visiting Bell Laboratories. Email: cpg@cs.utexas.edu.

ticular methods because they represent a good cross-section of extant reactive-system specification techniques; each of the authors is proficient in at least one of them; and each method has seen significant usage in industry, and therefore possesses a level of maturity that warrants its consideration for use in large-scale software development.

We have developed a set of criteria to help us evaluate the specification methods and assess important similarities and differences between them. In the course of our work, we discovered that the compatibility of a method with the 5ESS software development environment is at least as important as the technical characteristics of the method itself. Thus, while our list of criteria includes those normally considered by specification researchers, such as the presence of a well-defined semantics, we also included several others that we deem essential to the successful integration of the method into an actual software development process. As an example, in legacy software systems like the 5ESS switch, most current software projects involve the design or re-engineering of modules that must interface with an existing code base; it is therefore crucial that any new methodology ultimately result in code that executes without modification in the existing operating-system environment.

In work related to ours, Gerhart et al. [8] have put forth a set of criteria for evaluating formal methods in general, and Lewerentz and Lindner [9] have collected several case studies of the application of formal methods for reactive systems to the specification of a common design problem. Our work differs from the above in that both our choice of application problem and evaluation criteria are specific to a particular system and software development environment (however, because the 5ESS switch software is a good representative of the large class of legacy software systems, the results and methodology of our study have wide applicability). We hope that this paper will be of interest to software developers and language designers interested in the application of formal methods in large-scale software development.

The remainder of this paper is organized as follows. In Section 2, we state the problem to which we have applied the specification methods. Section 3 contains an overview of each method and a brief description of how we used it to solve the given problem. In Section 4, we define the criteria upon which we base our evaluation of the methods, which is presented in Section 5. We conclude in Section 6 with a summary of what we have learned from our study.

2 APS Problem

In collaboration with a software development project we discovered a simple problem called "Automatic Protection Switching (APS)"[10] described in terms of finite state machines. The idea is to provide more than one line for each communication channel. If a line degrades or fails, a backup line, called the "protection line" is used instead. The project chose *1+1 unidirectional non-revertive* APS. In this strategy, a protection line is allotted for each working line (1+1), the decision to switch lines is only made by the receiving side (unidirectional), and a switch to the protection line remains in effect even after the working line clears to an equivalent condition (non-revertive).

A standard redundancy method is used to check the accuracy of transmission of messages. We can assume that the number of erroneous bits received on the working line is continuously recorded, and that correction of messages is not an issue. (Some other protocol will take care of repair or retransmission of faulty messages.)

A line signal is considered *degraded* when it has a bit error rate within a dangerous range, typically between 10^{-5} and 10^{-9}. A line signal is considered to have *failed* when the bit error rate exceeds the degraded range, or whenever other hard failures have occurred, such as a complete loss of signal. (Since the signal is continuously monitored, a complete loss of signal is first detected as a degraded condition, and then a failure.) Either a degraded or failed line may *clear* spontaneously. The expected response to a degraded or failed signal on the working line is to (automatically) switch to the protection line, if that line is in better condition.

Technicians are provided with a set of commands to change the configuration of the channel:

remove line: The line is taken out of service.

restore line: The line is placed in service.

forced switch: The specified line is selected for communication, as long as it is in service.

conditional switch: The specified line is selected for communication, as long as it is available and of at least the quality of the selected line.

A protocol is needed to maintain the highest quality communication available while responding to operator requests and signal degradation and failure.

One way to present a protocol is to list all the possible states and transitions of the system in a table. We produced such a table as a prototype specification while we were trying to understand the requirements. For a small problem this might suffice, but we doubt that this would scale up well.

Another method is to describe some general properties about APS that should be provable from any specification of a protocol that satisfies the standards.

1. The quality of a line may be expressed by exactly one of the following 4 states: normal, degraded, failed, out of service.

2. One (and only one) of the two lines is always selected.

3. When a "remove line W" event occurs,

   (a) If the working line is not out of service, it goes out of service.

   (b) Otherwise, the working line remains out of service.

   The behavior on a "remove line P" event is analogous.

4. When a "restore line W" event occurs,

   (a) If the working line is out of service, it becomes normal.

   (b) Otherwise, the state of the working line does not change.

   The behavior on a "restore line P" event is analogous.

5. When a "forced switch to line W" event occurs,

   (a) If the working line is not out of service, it becomes the selected line.

   (b) Otherwise, the selection of the lines remains unchanged.

   The behavior on a "forced switch to line P" event is analogous.

6. When a "conditional switch to line W" event occurs,

   (a) If the working line is not out of service and is not of poorer quality than the protection line, the working line becomes the selected line.

   (b) Otherwise, the selection of the lines remains unchanged.

   The behavior on a "conditional switch to line P" event is analogous.

7. When a "line W degraded" event occurs,

   (a) If the working line is normal, then it deteriorates to a degraded quality.

   (b) Otherwise, the state of the working line remains unchanged.

   The behavior on a "line P degraded" event is analogous.

8. When a "line W failed" event occurs,

   (a) If the working line is degraded, then it deteriorates to a failed quality.

   (b) Otherwise, the state of the working line remains unchanged.

   The behavior on a "line P failed" event is analogous.

9. When a "W cleared" event occurs,

   (a) If the working line is degraded or failed, then it clears to a normal quality.

   (b) Otherwise, the state of the working line remains unchanged.

   The behavior on a "line P cleared" event is analogous.

10. If a "remove line", "restore line", "line degraded", "line failed", or "line cleared" event occurs, and the currently unselected line becomes of a higher quality than the selected line, the selection will be switched.

11. Removing, restoring, deterioration, or clearing of a line does not affect the state of the other line.

12. Switching the selected line does not affect the state of either line.

13. It is forbidden to switch to a line that is out of service, except when both lines are out of service.

14. The selected line will only change as a result of one of the following:

    (a) The selection is changed with a switch command.

    (b) The currently selected line deteriorates to a quality worse than the other line, or the currently selected line goes out of service.

    (c) The currently unselected line clears or is restored to a quality better than the selected line.

3 Solutions

Each of the authors prepared a specification that satisfied the requirements of the APS problem in a different language. We also reviewed the solutions prepared by the project team as an aid in understanding the requirements. Although there were many similarities between solutions, there were also many differences.

3.1 Modechart

The Modechart specification language [2] is a synchronous language designed for the specification of real-time systems. Modechart borrows from Statecharts [11] the use of hierarchical graphical formalisms to extend conventional state-transition diagrams. Modechart adds absolute and relative timing semantics to Statecharts transitions. Its semantics is defined in terms of two equivalent semantics. The first one is an “axiomatic” semantics formalized in RTL (Real-Time Logic [12]), which is a logic especially amenable to reasoning about the absolute timing of events. The second is an “operational” semantics [13] which captures a more computational and intuitive approach to the language. A set of tools has been developed for the specification, analysis and implementation of real-time systems within this framework [14, 15].

Modechart specifications are made up of modes and transitions. Modes can be thought of as hierarchical partitions of the state space, and can be combined by using parallel or serial composition. The most basic modes are called atomic modes. A serial relationship among several modes indicates that the system operates in at most one of the modes at any time. Serial modes may designate one mode as their initial mode, which is instantaneously entered once the serial mode is entered. Modes can also be in parallel, in which case the system operates in all modes simultaneously.

Transitions allow the system to switch from one mode to another (possibly at different levels within the hierarchy) and can only take place between modes which have a serial relationship. When a transition occurs, the transition is said to be *taken*. Entry of a serial mode causes the system to enter one of its children. Entry of a parallel mode forces the system to enter all its children. A transition out of one mode requires exit of all of the modes in parallel to it, if any. A transition has an associated expression that, when “triggered,” can cause the transition to be taken. The transitions’ expressions may be disjunctions of *triggering expressions* and *timing expressions*. Triggering expressions are conjunctions of external and internal events of the system (they are triggered iff the conjunction is true). Timing expressions are of the form (lb, ub), where lb is the lower bound and ub is the upper bound (possibly infinite) of a time interval since its origin mode was entered in which the transition may be triggered. If a mode has at least one transition triggered, one must be taken (non-deterministically).

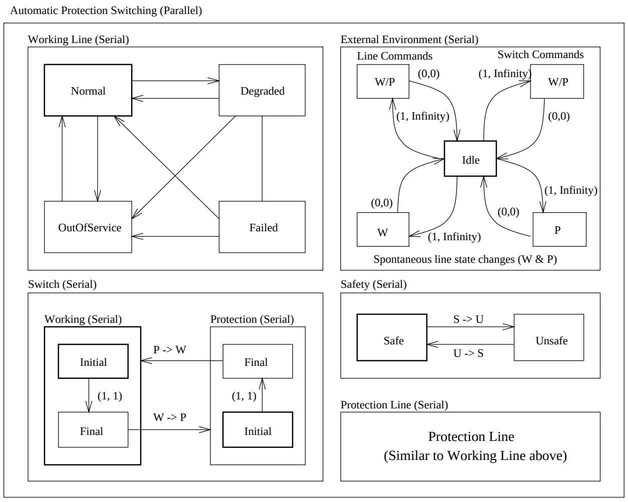

Figure 1: Modechart Specification for APS

Figure 1 sketches our Modechart specification for the APS problem. Modes are represented graphically as rectangles, which can enclose other modes. Initial modes are denoted by a thicker border in the figure. The specification is composed of the following modes:

- Working Line/Protection Line: serial modes for each of the two lines, representing their current state.
- External Environment: a serial mode modeling environmental behavior.
- Switch: a serial mode for the switch. It has two modes which denote the position of the switch, each with two child modes to avoid the possibility of instantaneous switching loops (by adding a single time unit delay).
- Safety: an auxiliary serial mode which is used to help prove system safety properties.

The Switch mode is the “main” mode in the specification. The conditions for the transitions in it (labeled $P \to W$ and $W \to P$) are derived from the description of the problem. The conditions for the transitions in the External Environment mode model the environment as specified, that is, it is in the Idle mode initially and, at any time, some operator command or line state change event may occur, after which the environment returns (instantaneously) to the Idle mode. The conditions to determine when the system enters an unsafe state ($S \rightarrow U$) have been obtained from the required properties in Section 2.

We proved the correctness of the specification (with respect to the properties stated in Section 2) automatically by using the verifier in the toolset. Some of the properties have been proved by showing that the system cannot be in the Unsafe state at any point in the computation graph for any amount of time, while other properties have been formally proved by using the Modechart verifier with predicates expressed in RTL, such as mode exclusion.

3.2 VFSM

The Virtual Finite State Machine (VFSM) methodology [3] 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 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.

```
S_CMD_CSW_WORK {

       # if working line is in same or
       # better state than protection
       # line, make working line the
       # currently selected line
   IA: I_WORK_NORM ||
       I_WORK_DEGR && I_PROT_DEGR ||
       I_WORK_DEGR && I_PROT_FAIL ||
       I_WORK_FAIL && I_PROT_FAIL
           ? O_SELECT_WORK;

   NS: always > S_MAIN;
}
```

Figure 2: Example State of VFSM Solution

In developing a VFSM specification of the APS problem, we represented the states of the two lines and the currently selected line using virtual inputs: for example, there is a virtual input for each of the four possible states of the working line, exactly one of which is in the VIR at any time. We associated one or more VFSM states with each of the fourteen operator commands and line-state changes. Figure 2 shows the VFSM state that handles the command to conditionally switch to the working line. In the input-action (IA) section, a virtual output that selects the working line is produced if it is not in a worse state than the protection line; the latter condition is expressed in terms of virtual inputs. The next-state (NS) section then specifies a return to the main state of the VFSM to await the next command or line-state change.

The VFSM toolset includes a validator that exhaustively checks for concurrency-related errors such as deadlock and livelock. The validator will soon be enhanced to enable the checking of user-specified properties of an application, similar to those for the APS problem listed in Section 2.

3.3 Esterel

Esterel [4] is a language, with a precisely defined mathematical semantics, for programming the class of input-driven deterministic reactive systems — those that wait for a set of possibly simultaneous inputs, react to the inputs by computing and producing outputs, and then quiesce, waiting for new inputs. Es-

TEREL is based on the "synchrony hypothesis," which states that every reaction of the system to a set of inputs is assumed to be instantaneous with respect to the environment. In practice, this amounts to requiring that the environment of the system be invariant during every reaction, or equivalently, that reactions are atomic. This requirement is satisfied by the 5ESS environment since its operating systems are non-preemptive.

The programming model in ESTEREL is the specification of components, or modules, that run in parallel and can have hierarchical structure. Modules communicate with each other and the outside world through *signals*, which are broadcast and may carry values of arbitrary types. Consistent with the synchrony hypothesis, the emission and reception of signals is considered to be instantaneous. Pre-emption operators and nesting permit coding behaviors such as interrupts and timeouts, and provide for precise and unambiguous descriptions of complex responses to input events. ESTEREL allows only deterministic behaviors to be specified: the inputs to every reaction (and the current values of variables) fully determine the outputs emitted in that reaction as well as the input-output behavior of the rest of the program.

Instantaneous broadcast communication, parallelism and pre-emption, present in most synchronous languages, are structuring tools for programming convenience and simplify reasoning about reactive systems. They preserve determinism and do not incur any run-time overhead —the compiler automatically performs the complex interleaving between parallel modules and all internal communication is compiled away, a task that can be error-prone if done by hand. In particular, ESTEREL strongly supports "logical concurrency": conceptually concurrent tasks can be described as parallel modules, while the implementation produced by the compiler is purely sequential. The ESTEREL compiler generates a single deterministic finite state machine that yields a predictable and efficient implementation.

The design of our ESTEREL implementation of APS is given in Figure 3. The "line selection" module continually emits signals to the outside world indicating the current line selection, the "P/W line status" modules continually emit signals to the outside world indicating the current state of the lines, and the "previous P/W line status" modules continually emit internal signals indicating the previous state of the lines. (The synchrony hypothesis renders it necessary to emit these signals corresponding to the previous state.) The "Process Input" module receives

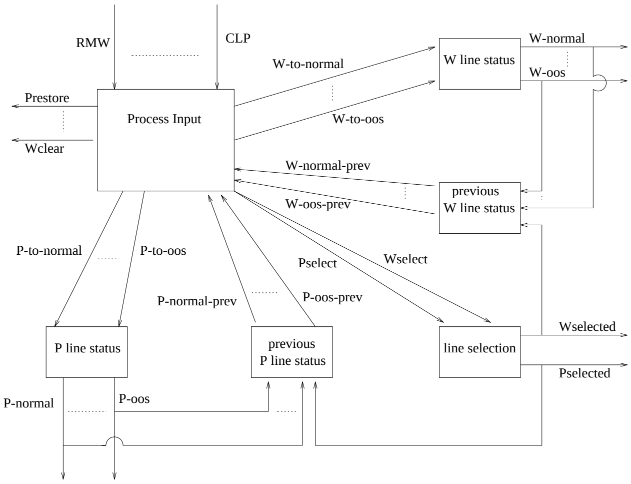

Figure 3: Design of ESTEREL Implementation of APS

inputs from outside world, corresponding to changes in the quality of lines and commands from technicians. Based on the signals from the "previous P/W line status" modules, the "Process Input" module determines how the status and selection of the lines should change, and emits corresponding internal signals to the "P/W line status" modules and the "line selection" module. Furthermore, it emits signals to the outside world indicating actions that should be taken on the lines, *e.g.* removal, restoral.

The AGEL [16] development environment for ESTEREL provides high-quality tools, including an editor, compiler, simulator, debugger, and verifier. The verifier provides a graphical representation of the generated state machine, and computes reduced state machines based on bisimulation equivalence [17]. The simulator supports interactive execution of ESTEREL programs, by allowing the user to provide input signals and corresponding values and observe the output signals and corresponding values that are produced in response. In conjunction, the execution of the ESTEREL source code and the generated state machine can be followed through the symbolic debugger. The editors support both textual representations of ESTEREL programs and graphical representations of state machines. We found this environment to be quite useful in writing and debugging our implementation.

Furthermore, we automatically formally verified the correctness of our ESTEREL implementation using the technique developed in [18]. In particular, this verification process involves three steps:

1. Automated translation from a class of temporal logic safety properties [19] to ESTEREL.

2. Compilation of the given program in parallel with the ESTEREL "model" of the properties, resulting in a finite state machine.
3. Analysis of this finite state machine for satisfaction/violation of the properties.

All three of these steps have been automated, and are supported by a toolset (described in more detail in [18]). Using this technique and toolset, we automatically formally verified that our ESTEREL implementation satisfies all the properties given in Section 2, expressed as safety properties in temporal logic.

3.4 LOTOS

LOTOS [5] was designed for protocol specification, especially within the telecommunication domain[20]. Basic LOTOS is that part of the language that describes interaction of processes in terms of shared events. A process algebra paradigm is used, where each process is described in terms of legal sequences of events experienced by that process.

The APS requirements table was first transcribed into a special finite-state machine dialect, called Primitive LOTOS[21]. A compiler then translated this form into an executable subset of Basic LOTOS. The result is a state-oriented style[22], but with explicit representation of states by processes. That is, each state of the table is represented by a LOTOS process, and each transition is represented by an event sequence that ends with an invocation of the appropriate process (state).

Figure 4 shows one process description in elided form (arguments to processes have been omitted). The figure describes the transitions possible from the state where both lines are normal, and the working line is selected. Each line shows a possible transition as a result of a single event. For example, if the `RMW` (remove working line) event occurs, then the system takes a transition to the state `PON` (protection line selected, working line out of service, and protection line normal). The alternation operator `[]` is used to join all the possible transitions from this state.

Primitive LOTOS is simpler than Basic LOTOS, and it has a tabular dialect that was easier for the domain experts to review. The observation that tabular representations are a useful alternative to other formalisms has also been made by Leveson[23]. Tables are certainly familiar to software engineers, and they are easy to inspect for omissions. Of course, tables quickly prove unwieldy when they grow in size. We were fortunate that the APS problem was simple enough to express in a small table.

```
PROCESS State_WNN [...]: NOEXIT :=
  RMW; State_PON [...]  []
  RMP; State_WNO [...]  []
  FCP; State_PNN [...]  []
  CSP; State_PNN [...]  []
  DGW; State_PDN [...]  []
  DGP; State_WND [...]
ENDPROC
```

Figure 4: fragment of LOTOS solution

Most of the properties were easy to check informally. We did not attempt to represent the properties explicitly, but most could have been described in separate LOTOS processes. By conjoining these separate processes and running a simulator we could have detected potential deadlocks, which would have demonstrated inconsistency between the two descriptions.

We found simulation to be useful in debugging the LOTOS specification. As part of simulation we generated traces of individual scenarios in Message Sequence Chart [24] form. These were useful in discussions with domain experts who reviewed the original project requirements. In fact, most of the properties were originally expressed as scenarios by these reviewers.

3.5 Z

The **Z** notation [6] combines abstract data modeling and a mathematical toolkit based on set theory and first-order predicate logic. In conjunction with **Z**'s modest structuring conventions, these may be used to specify system state and valid state changes for an instantaneously responding reactive system.

To construct a specification for APS, we adopted certain structuring and naming conventions in addition to the standard conventions of **Z**. A loosely object-oriented style of specification due to Hall [25], in which much of the necessary **Z** can be macro-generated, was followed. In addition, an event semantics suggested by the work of Zave and Jackson [26] was assumed. Though built up from simpler pieces (see Figure 5), each event was described by a separate schema, with preconditions guarding against illegal operations, and postconditions specifying the outcomes.

The structure provided by the conventions adopted allows a natural specification of the behavior. Indeed, in many cases the behavior specification was precisely a required property. Some properties had to be derived from the specification, which in this case was straightforward to do by hand. However, while some technology is available for reasoning with **Z**, such verification is not a routinely automatable process.

For development of reactive systems, simulation appears to us to be essential. In this case, because of the simplicity of the specification, we were able to carry out a mechanical translation into the SETL2 language [27]. The translation results in a nondeterministic finite state automaton in general, since the **Z** need not fully constrain the outcome of any event. The finiteness of the system is relied upon to provide a well-defined set of possible states. Thus, any **Z** schema corresponds to a SETL Boolean function over the combinations of pre- and post-states. The features of SETL allow straightforward programming constructs to compute the allowable initial states, and to determine the allowed transitions from any state that can be reached.

Given this translation, it is possible to perform some simple verification: find the reachable states; check for deadlocks; check for non-determinism (presumably a defect for this application); and generate a finite state machine that implements the specification efficiently, if not particularly elegantly.

Of course, this naive technique does not generalize well. More sophisticated development environments based on **Z** and related notations are now becoming available (for example, the B-Method [28]). These offer support for both analysis and implementation, but the limited industrial use of the technology for reactive systems makes it difficult to evaluate its utility for large-scale projects. It is clear, however, that any practical use of **Z** must add a significant amount of structure to the minimal semantics of the notation.

3.6 SDL

SDL is one of the most mature formal methods used in the telecommunications domain. It has recently evolved to include concepts from object-oriented languages to provide modularity. There are commercial toolsets available for SDL that facilitate design, debugging and maintenance. If the descriptions of actions are written in a C-like language, then the specification may be compiled into C. Our internal SDL tool set supports this formal SDL dialect, see Figure 6, but does not support the recent object-oriented extensions.

There are many ways to construct an SDL model to describe a given behavior. Process states are represented by an explicit state identifier and by other persistent data stored by the process. Similarly, events

SelectionSpontaneous

$$\begin{array}{|l}
\mathit{ChannelOp} \\
\hline
\mathit{selected}' = \text{not } \mathit{selected} \Leftrightarrow \\
\quad (\mathit{LineSet}'\,(\text{not } \mathit{selected})).\mathit{status} > \\
\qquad (\mathit{LineSet}'\,\mathit{selected}).\mathit{status}
\end{array}$$

Degrade_Event

$$\begin{array}{|l}
\mathit{ChannelOp} \\
\mathit{line}? : \mathit{LINEID} \\
\hline
\upharpoonright \mathit{LineDegrade} \\
\mathit{SelectionSpontaneous}
\end{array}$$

Figure 5: **Z** fragment describing spontaneous switch response to a degrade event on a specified line.

```
STATE SEL_WORKING;
  INPUT Clear(line);
    DECISION ((cond[line] == OOS)
        || (cond[WORK] == OOS));
    (TRUE):
      TASK 'badinput = 1';
      NEXTSTATE -;
    ENDDECISION;
    TASK 'cond[line] = NORMAL';
    DECISION ((line == PROT) &&
        (cond[WORK] > cond[PROT]));
    (TRUE):
      NEXTSTATE SEL_PROTECTION;
    ENDDECISION;
    NEXTSTATE -;
ENDSTATE SEL_WORKING;
```

Figure 6: Fragment of SDL model

```
CLAIM;
    ((APSENV FIRSTMSG Done)
      && (envmsg == CONDSWITCH)
      && (sel != psel))
    IMPLIES (cond[sel] <= pcond[psel])
      UNTIL (sel == psel);
ENDCLAIM;
```

Figure 7: Example of temporal logic for SDL models

have an identifier and optional data parameters. In each case, information can be built into either the identifier or the associated data. This flexibility can be used to produce a model that meets the formal goals as well as informal goals such as making the model more intuitively understable to the target community.

The SDL specification was written in a style meant to emphasize clarity of behavior and would be quite different with other constraints. The system uses two explicit states, one for each line that may be selected. Other information, such as the quality of each line, was kept as persistent data. Incoming events were given one parameter - the line id to which the event is targeted. Thus, the SDL model has two states with seven inputs per state. Within each state-event three activities are performed sequentially: check for illegal request, determine and set changed line quality, determine and set new selected line.

To validate the model, the original system requirements were written as temporal logic invariants in an SDL-like notation, see Figure 7. A separate SDL model was written to represent the environment in which the requirements must hold. In this case the system is fully defined and must operate in an environment that can generate any event at any time.

Our validator[29] performs the most effective state space search possible given memory and time constraints. Any scenario violating basic properties or user claims is presented in detail as a counter-example. Since the total state space for this specification is quite small, an exhaustive search was possible. The same technique has been applied successfully to SDL specifications of very high complexity.

3.7 C

Several different solutions to the problem were written in C. These programs were useful in demonstrating the properties of C that we evaluated, but they were not used in the industrial project. These programs fell into two major classes.

The first program class was a simple finite state machine that encoded the state transition diagram in the requirements document into a two-dimensional array based on current state and event; the control code was merely a loop that waited for a new event and then did a table look up to determine the new state. It was easy to code and simple to check because we had already produced a prototype table specification of the requirements. This solution would be easy to maintain as long as legal state transitions were few and could be represented as a table. Of course, the state transition table was a sparse matrix and thus wasted data space.

Since large industrial problems are typically difficult to describe in table format, the second class of programs did not use our prototype table specification. Based on observations of some industrial code, it seemed natural to implement a "switch" structure based on current state. In each state's "case" statement there was an additional "switch" structure based on event. This code accomplished the same functions as the first class of implementations, but was much larger and more difficult to maintain for this problem. Every state's "case" statement had its own custom logic on how to deal with events. Two basic concepts were repeated and thus were split out into their own functions. These concepts were "which facility is better now" and "should a switch to the better facility occur". This class of solution would make it easy to map behavior back to an individual requirement and would also avoid the wasted data space problem.

4 Criteria

We now present the criteria that we consider important for any specification language proposed for reactive systems. We have grouped these criteria into two categories; the order in which the criteria appear below within each group is arbitrary. The *fundamental selection criteria* are those that we believe are very important for a language to be successful, especially in 5ESS software development. The *important selection criteria* are those that we believe are generally very important, but for which we found less evidence in our setting.

The aim of these criteria is to help assess the suitability of languages for industrial software development, and they reflect concerns that arise through the various phases of the traditional waterfall model of software development. An informal assessment of the importance of each critieria during the phases of the software lifecycle is givin in Figure 8. While the nature of these criteria is necessarily subjective, we believe that they accurately reflect the important considerations in our setting.

4.1 Fundamental Selection Criteria

Applicability Can the technology describe real world problems and solutions in a natural and straightforward manner? If the technology makes assumptions about the environment, are they reasonable or benign? Is the technology compatible with the physical reality of the intended system? In particular,

Usability in Embedded Systems Can the specification be added as a component(s) to an existing (specification of the) system?

Hardware/software compatibility Are all the assumptions made by the language supported by the existing hardware/software in the system? (For example, if the specification assumes the existence of threads, or a certain type of communication mechanism – such as message-passing, shared memory, or semaphores – are these facilities provided by the operating system?)

Implementability: Reasonable path to code Can the specification be easily refined or translated into an implementation that is compatible with the rest of the system? (For example, if code is generated automatically, is it in the same language/dialect as the rest of the system?) How difficult is this translation? Is it automated? Is the generated code efficient? Furthermore, is there a clean well-defined interface between the machine-generated code and the manually generated portions of the implementation?

Testability/Simulation Can the specification be used to test the implementation? Is every statement in the specification testable by the implementation? Is it possible to execute a specification, typically by interactively providing it with input stimuli and observing that the behavior and output produced in response are as expected?

Checkability Can a domain expert check the specification for accuracy? How readable are specifications for domain experts (as opposed to experts in the technology)?

Maintainability Can the specification be used as the starting point for maintenance activities? Is it easy to make modifications to the specification?

Modularity Does the method possess simple composition operators that allow a large specification to be more easily written and understood by decomposing it into smaller parts? Is it possible to make conceptually small additions, changes, and generalizations to a specification without major rewriting?

Level of abstraction/expressibility How closely and expressively can the objects in the specification describe, from the user's point of view, the objects, equipment and the environment in the domain? How large is the ratio between the amount of interesting properties of a system that can be described versus the amount of extra information or detail included while doing so?

Soundness Do the language and tools provide support for detecting inconsistencies and ambiguities in specifications? Does the language have a precisely defined semantics?

Verifiability Is it possible to demonstrate formally that a specification or program satisfies properties stated at the same or a higher level of abstraction? Is this process automated, or can it be automated? How easy is it to use?

Run-time Safety If the language supports automated code generation, does the resulting implementation degrade gracefully under unexpected run-time conditions?

Tools Maturity How much development have the tools undergone in terms of available facilities, quality, amount and quality of training available, amount of support available (at development time, test time and run time), user base size, industrial use, momentum and impetus gained in industrial settings?

4.2 Important Selection Criteria

Looseness Does the language permit incompleteness or non-determinism in the specifications?

Learning Curve Can a completely new user quickly learn the concepts, techniques and heuristics to make useful application of the language?

Language Maturity How much development has the language undergone in terms of certification, amount and quality of training available, user base size, industrial use, momentum and impetus gained in industrial settings?

Phases						Criteria
R	D	I	T	M	O	
+		+				Applicability
		+		+		Implementability
+	+		+			Testability/Simulation
+			+	+		Checkability
				+		Maintainability
	+			+		Modularity
+	+					Level of abstraction/expressibility
+			+			Soundness
+	+	+	+	+		Verifiability
		+		+		Run-time Safety
		+	+	+		Tools Maturity
+						Looseness
					+	Learning Curve
					+	Language Maturity
	+					Data Modeling
+	+	+		+		Discipline

R = requirements phase
D = design phase
I = implementation phase
T = test phase
M = maintenance phase
O = Other (does not fit any particular phase)

+ = Criterion is important during phase

Figure 8: Importance of Criteria During Software Lifecycle Phases

Data Modeling Does the language provide facilities to describe data representation, relationships and/or abstractions? Does it provide them in a natural, integrated way?

Discipline Does the language force users to write reasonably well-behaved programs?

5 Our Evaluation of the Languages

We have evaluated Modechart, VFSM, ESTEREL, Lotos, **Z**, SDL, and C using the criteria in the previous section. As part of this assessment, a specification of the APS problem was written in each of these languages, using the available toolsets. The toolsets and our specifications were then evaluated to see how well they fit our criteria.

In earlier work, we had re-written larger portions of the 5ESS software in these languages. In particular, VFSM has been used in the design of many 5ESS software modules, including applications in call processing and signaling[30]. We have written an ESTEREL version of some alarms software in the 5ESS switch[31]; this feature was also written in Modechart. We have used LOTOS to specify various parts of telecommunications switching systems, including call processing and maintenance facilities[21]. Many real-time switching protocols in 5ESS signaling and call processing applications have been written in SDL[32]. We have written a **Z** specification of a set of 5ESS subscriber telephone features[33]. Finally, we have written in C many features of the 5ESS switch. We emphasize that our evaluation of the languages and tools reflects our APS specifications as well as our previous experiences.

Our findings appear in Figure 9. We have used the symbol + to indicate a particular strength of a language, – to indicate a clear weakness of the language, and 0 to indicate that the language meets the criterion in a way that is adequate, but less than ideal. NA indicates that the criterion is not applicable to the language; for example, languages that currently do not support code generation cannot be evaluated for run-time safety.

We note these ratings are highly dependent on our focus on reactive systems, and may differ for other types of applications.

Our evaluation indicates the following trends. Applicability is not an issue, and implementability and testability/simulation are by and large not much of a problem for the languages that we evaluated – with the exception of **Z**. We note however that **Z** is applicable to many systems that are not reactive in nature.

Verifiability – proving the correctness of a specification with respect to requirements – is a strength of all the languages we considered, with the obvious exception of C. In these approaches, requirements typically may be specified in temporal logic, and may include properties such as continual progress and absence of deadlock.

Contrary to traditional criticism and popular perception, we found that the learning curve was not a problem, at least for this group of researchers and developers. Similarly, the maturity of languages and tools has increased, and formal methods are scoring better than critics claim.

Perhaps surprisingly, our evaluation suggests that maintainability is not a strength of any of these languages. Since maintainability is one of the most serious concerns for embedded legacy systems, such as the 5ESS switch, we believe this issue must be addressed in order for formal methods to be adopted for large-scale industrial use. Criteria that are related to the maintainability of large-scale industrial software, such

	MODECHART	VFSM	ESTEREL	LOTOS	Z	SDL	C
FUNDAMENTAL CRITERIA							
Applicability	+	+	+	+	0	+	0
Implementability	0	+	+	0	–	+	+
Testability/Simulation	0	+	+	+	–	+	+
Checkability	+	0	0	0	0	+	–
Maintainability	0	0	0	–	0	0	0
Modularity	+	–	0	0	+	0	0
Level of abstraction/ expressibility	+	0	+	0	+	0	–
Soundness	+	0	+	+	0	0	0
Verifiability	+	+	+	+	+	+	–
Run–time Safety	NA	0	0	0	NA	0	–
Tools Maturity	0	+	+	0	0	+	+
IMPORTANT CRITERIA							
Looseness	0	0	0	0	+	0	0
Learning Curve	+	+	0	0	0	+	+
Language Maturity	0	0	0	+	+	+	+
Data Modeling	–	–	0	0	+	0	0
Discipline	0	+	0	0	0	0	–

+ = Strength, 0= Adequate, – = Weakness, NA = Not applicable

Figure 9: Our Evaluation of Languages by Criteria

as checkability, modularity, and discipline, also by and large did not fare well, and partly contribute to the lack of maintainability. Run-time safety, another related criterion, also did not score well; however, the scores in this category are hurt by the ability of the languages to interface with C – which is essential for applicability to the 5ESS switch.

With regard to data modeling, most of the languages we considered are aimed at abstractly representing the behavior of reactive systems, which can often be accomplished without sophisticated data modeling; in fact, data modeling did not arise in our application of Automatic Protection Switching. However, we do believe that it plays an important role in larger problems.

6 Conclusion

We have described specifications in seven different formal methodologies of a simple reactive system that is a part of the software for the 5ESS switching system. We proposed a set of criteria for evaluating the methodologies that takes into account specific characteristics of large-scale software development projects, and we then used the criteria to evaluate each of the methodologies. There is no clear "winner" among them, in the sense that no methodology dominates all the others in all criteria.

Our evaluation reveals several insights into the selected methods. **Z** and C are the least well-suited to the chosen specification task, and the ratings reflect this. **Z** is primarily a requirements language for abstract data types and does not offer a modeling language; it therefore scores poorly on criteria like implementability and testability/simulation. By contrast, C certainly provides a "modeling language" of great expressive power, but has no facility for specifying high-level requirements, and thus garners low marks for its level of abstraction and expressibility.

The remaining five methods offer modeling languages, with some support for specifying and verifying high-level requirements. These methods can be compared according to the amount of expressive power they provide. We have found that there is a tradeoff between expressive power and ease of understanding. The dialect of LOTOS we used is the simplest and least expressive of the methods, essentially equivalent to a finite-state automaton. Although it is easily comprehended by non-experts, the specification is quite monolithic and unstructured, and simple extensions to the problem, e.g. adding another communication line, would greatly increase the size of the specification.

Modechart, SDL and VFSM are also based on finite-state automata, but each adds a modest extension on top of the basic FSM model: Modechart adds serial and parallel modes, SDL adds simple data structures, and VFSM adds the virtual input register. These increases in expressive power result in better-structured solutions to the APS problem: with these methods, we were able to represent the states of the communication lines and the currently selected line as independent parts of the solution, instead of combining them into a single state variable. While these increases in expressive power extend the size of problem to which these methods can be comfortably applied, they are not sufficient to allow the modular structuring of very large reactive systems.

ESTEREL is clearly the most expressive of the five: it provides a powerful set of communication and abstraction mechanisms for structuring a reactive program as a collection of interacting components, and thus scales the best to large application problems. The many features and constructs present in ESTEREL suggest that, relative to the other methods, a larger amount of expertise is required to be able to understand and construct ESTEREL specifications.

7 Future Work

Although the APS case study was sufficient to make some evaluations of the technologies we used, it did not fully exercise all of the criteria of interest. For example, maintainability can best be judged by making modifications to the problem and observing how the different solutions can be adapted to accommodate the changes. We are pursuing a second phase of this case study that will elaborate on some of these findings and that will investigate some other criteria of interest.

7.1 New Requirements

As mentioned earlier, there are several types of protection switching. The unidirectional style is the simplest, since it allows autonomous action at each end of the communication channel. Our colleagues in development ultimately decided to use a more complicated style, called 1+1 bidirectional non-revertive APS.

Bidirectional switching requires that each end use the same line, working or protection, whenever this is possible. If one end needs to switch lines it first asks the other end to switch, and then switches itself after getting positive acknowledgment. A protocol for requesting and acknowledging these requests is described in the standards.

Here is how a successful transition takes place:

1. The side that wants to make a transition (side A) changes the signal that it sends to the other side.
2. The other side (side B) receives the new signal and responds by accepting the transition. At this point side B may make any necessary switch.
3. The originating side (A) receives the acceptance of the new state and changes its signal again (reacknowledgment). At this point side A may make any necessary switch.
4. Side B receives the new signal from side A, showing that the transition has been successfully finished.

There are a few potential conflicts that may occur:

- Both sides might request the same transition at approximately the same time.
- Each side might request a different transition at approximately the same time.
- One side might receive a higher-priority request during the transition.

The synchronization of the two ends of the channel introduces several new types of requirements:

- Each end of the channel must now keep track of both its own state and the state of the other end.
- Each end of the channel now receives stimuli from 3 sources, rather than 2. I.e., it may receive events from the other end, from the (human) maintainer of the communication link, and from changes in line quality detected by the hardware.
- Although it may not be necessary to consider requests from the two ends of the channel as occurring truly concurrently, it is certainly possible for the state of each end to change during the requesting protocol.
- One of the cases that must be handled in the requesting protocol is failure of the other end to acknowledge the request. This introduces a primitive timing requirement.

The addition of these new requirements significantly increases the state space of possible configurations of the system.

7.2 Preliminary Findings

We modified most of our solutions to the original problem to handle the new requirements. Although we have not finished our evaluations of the solutions, we have not found evidence to change our earlier assessments of the various formalisms. On the other hand, we have found that the increase in complexity supports some of our claims about maintainability. For example, the state-oriented style of specification used with LOTOS is awkward and difficult to review for the larger problem. Other formalisms fared better, but all of them were stressed by the large state space.

Unfortunately, the standards are ambiguous in their description of certain cases, notably those cases where a request cannot be granted. We believe that these ambiguities may lead implementers of APS to choose interpretations that conflict with one another.

To test the ambiguity problems we examined several cases where implementers could choose different interpretations, and we examined the consequences of these choices. In some cases it was possible to completely defeat the protection switching protocol, causing the communication link to fail, even though there was at least one working line in each direction. (We hasten to add that the implementation chosen by our development colleagues does not suffer from any of these flaws.) We hope that future authors of standards will consider using formal languages, so that ambiguity can be minimized.

8 Acknowledgements

It is a pleasure to acknowledge the help of our colleagues Bill Bielawski, Peter Fales, and Dave Furchtgott who introduced us to APS and Douglas Stuart, who helped with one of the formalisms. Joanna McCaffrey provided substantial SDL expertise and several valuable suggestions that improved the paper. We thank Mary Zajac, David Weiss, and Jan Sharpless for their support in conducting this study. We thank David Weiss and the anonymous referees for their help in improving the presentation of our work.

Carlos Puchol was partially supported at the University of Texas at Austin by a Fulbright fellowship from Spain's Ministry of Education and Sciences.

References

[1] K.E. Martersteck and A.E. Spencer, "Introduction to the 5ESS(TM) switching system", *AT&T Technical Journal*, vol. 64, no. 6 part 2, pp. 1305–1314, July-August 1985.

[2] F. Jahanian and A. Mok, "Modechart: A specification language for real-time systems", *IEEE Transactions in Software Engineering*, vol. 20, no. 12, pp. 933–947, December 1994.

[3] F. Wagner, "VFSM executable specification", in *CompEuro92*, 1992.

[4] G. Berry and G. Gonthier, "The ESTEREL synchronous programming language: design, semantics, implementation", *Science of Computer Programming*, vol. 19, pp. 87–152, 1992.

[5] ISO, *LOTOS—A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, ISO, 1989, International Standard ISO 8807.

[6] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall International, 2nd edition, 1992.

[7] ITU-T, "Specification and Description Language SDL", Recommendation Z.100, 1993.

[8] S. Gerhart, D. Craigen, and T. Ralston, "Experience with formal methods in critical systems", *IEEE Software*, vol. 11, no. 1, pp. 21–28, January 1994.

[9] C. Lewerentz and T. Lindner, "Case study 'production cell': A comparative study in formal specification and verification", Tech. Rep., Forschungszentrum Informatik, 1994.

[10] Bellcore, "Synchronous optical network (SONET) transport systems: Common generic criteria", Tech. Rep. TR-NWT-000253, Issue 2, Bellcore, 1991.

[11] D. Harel, "Statecharts: A visual formalism for complex systems", *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[12] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems", *IEEE Transactions in Software Engineering*, vol. 9, pp. 890–904, 1986.

[13] C. Puchol, D. Stuart, and A.K. Mok, "An operational semantics for Modechart specifications", Tech. Rep. UTCS-TR95-37, Department of Computer Sciences, The University of Texas at Austin, September 1995.

[14] P.C. Clements, C. L. Heitmeyer, B. G. Labau, and A. T. Rose, "MT: A toolset for specifying and analyzing real-time systems", *IEEE Real-Time Systems Symposium*, December 1993.

[15] C. Puchol, A.K. Mok, and D. Stuart, "Compiling Modechart specifications", in *IEEE Real-Time Systems Symposium*, December 1995, pp. 256–265.

[16] "AGEL workshop manual version 3.0", 1989, Produced by ILOG, Mountain View, CA, USA.

[17] R. Milner, *Communication and Concurrency*, Series in Computer Science. Prentice Hall, 1989.

[18] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen, "Safety property verification of ESTEREL programs and applications to telecommunications software", in *7th International Conference on Computer-Aided Verification, Volume 939 of the Lecture Notes in Computer Science*, July 1995, pp. 127–140.

[19] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1992.

[20] R. Boumezbeur and L. Logrippo, "Specifying telephone systems in LOTOS", *IEEE Communications*, pp. 38–45, August 1993.

[21] M. Ardis, "Lessons from using Basic LOTOS", in *16th International Conference on Software Engineering*, 1994, pp. 5–14.

[22] C.A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma, "Specification styles in distributed systems design and verification", *Theoretical Computer Science*, pp. 179–206, 1991.

[23] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements specification for process-control systems", *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–707, September 1994.

[24] ITU-T, "Message Sequence Chart", Recommendation Z.120, 1992.

[25] A. Hall, "Using Z as a specification calculus for object-oriented systems", in *VDM and Z – Formal Methods in Software Development*, D. Bjørner, C. A. R. Hoare, and H. Langmaack, Eds. 1990, pp. 290–318, Springer-Verlag.

[26] P. Zave and M. Jackson, "Where do operations come from? A multiparadigm specification technique", 1994, Draft available from the authors.

[27] W. Kirk Snyder, "The SETL2 programming language", Tech. Rep., Courant Institute of Mathematical Sciences, New York University, 1990.

[28] J.-R. Abrial, *The B-Book*, Cambridge University Press, 1995.

[29] G.J. Holzmann, "Practical methods for the formal validation of SDL specifications", *Computer Communications*, pp. 129–134, March 1992.

[30] 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 (ISS95)*, Berlin, Germany, April 1995.

[31] L.J. Jagadeesan, C. Puchol, and J.E. Von Olnhausen, "A formal approach to reactive systems software: A telecommunications application in ESTEREL", *Formal Methods in System Design*, vol. 8, no. 2, March 1996, Preliminary version appeared in the Proceedings of the IEEE Workshop on Industrial-Strength Formal Specification Techniques, April 1995.

[32] John A. Chaves, "Formal methods at AT&T – an industrial usage report", in *Formal Description Techniques IV*, Parker and Rose, Eds. 1992, pp. 83–90, North-Holland, Amsterdam.

[33] P. A. Mataga and P. Zave, "Formal specification of telephone features", in *Z User Workshop, Cambridge 1994*, J. P. Bowen and J. A. Hall, Eds. 1994, pp. 29–50, Springer-Verlag.