Developing and Validating Thousands of Executable Finite State Machines¹

Glenn Semmel
NASA, YA-E1-S3
Kennedy Space Center, FL 32899
321-861-2267
glenn.semmel-1@ksc.nasa.gov

Gwendolyn H. Walton
School of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL 32816
407-823-3276
gwalton@mail.ucf.edu

Abstract—In 1996, a project began at NASA's Kennedy Space Center to replace the legacy Launch Processing System with a state-of-the-art process control system called the Checkout and Launch Control System. Over 13 million lines of real time control software are being reengineered from a legacy language called Ground Operations Aerospace Language to a modern graphical programming language called ControlShell by Real Time Innovations. ControlShell combines a graphical language with a component-based design methodology. The ControlShell finite state machine component type is semantically similar to the statecharts of the Unified Modeling Language (UML).

Thousands of executable finite state machines are being developed for the new launch system. The new system must deliver exceptionally reliable software to maintain safe operations and a high level of confidence by the end users. This paper discusses some of the development challenges of this project, the design approach, and the use of automated model checking for validation.

TABLE OF CONTENTS

1. INTRODUCTION
2. COMPONENT-BASED ARCHITECTURE
3. MODEL CHECKING
4. CASE STUDY
5. CONCLUSIONS AND FUTURE WORK

1. INTRODUCTION

In 1996, a project began at NASA's Kennedy Space Center to replace the legacy Launch Processing System with a state-of-the-art process control system called the Checkout and Launch Control System. Over 13 million lines of real time control software are being reengineered from a legacy language called Ground Operations Aerospace Language.

This project has many inherent developmental and testing challenges. The new system must deliver exceptionally reliable software to maintain safe operations and a high level of confidence by the end users. In addition to modern requirements engineering, modeling, design, and implementation, the developmental effort will require

extensive testing and validation prior to the system becoming operational. It is estimated to require 30 person years just to develop and execute software validation procedures for the real time control applications software.

Functionality of Checkout and Launch Control System

The Checkout and Launch Control System must provide the facilities for system engineers and test conductors to command, control, and monitor space vehicle systems from the start of the Shuttle Interface Test through terminal countdown and launch or abort/safing and scrub turnaround. The system will continually monitor the Shuttle and its ground equipment, including its environmental controls and the equipment that loads propellants. Consoles with vehicle responsibilities communicate information directly to and from the Shuttle computer systems. Consoles with ground support equipment responsibility communicate information to and from the Hardware Interface Modules, which are connected to the numerous ground support systems. Each module is capable of interfacing to approximately 240 sensors or controls. Overall, some 40,000 temperatures, pressures, flow rates, liquid levels, turbine speeds, voltages, currents, valve positions, switch positions, and many other parameters must be monitored.

Real Time Processing System

As illustrated in Figure 1, the Real Time Processing System allows a system engineer to control flight equipment and ground support equipment under test. It is used for test and

Figure 1. Real Time Processing System

launch operations of the Space Shuttle and is comprised of hardware subsystems, system software, and applications

¹ U.S. Government work not protected by U.S. copyright

software. The system software and hardware provide a reliable platform for controlling the hardware interfaces to the hardware end items such as power supplies, valves, and tanks. The system software transfers commands and data between applications software and the end items. Applications software contains the reactive control logic that monitors and controls the end items under test.

2. COMPONENT-BASED ARCHITECTURE

The Checkout and Launch Control System is being developed using a component-based architecture. Within the system, End Item Manager (EIM) applications are user defined processes that accept command requests from console operators and communicate with other EIMs. An EIM manages multiple hardware components, such as power supplies and valves, referred to as End Item Components (EICs). See Figure 2. The command requests may be individual commands to the hardware EIC or may be commands that cause the EIM to execute a sequence of hardware commands to put a subsystem into a specific state while monitoring the measurement responses to assure safe

operation. For example, an EIM may accept a vehicle power up command, which would issue a sequence of commands to power on multiple subsystems while monitoring various EICs, such as power busses, and the status of the subsystems. In Figure 2, Function Designators (FDs) represent unique commands and measurements to and from the actual hardware EICs, respectively.

In Figure 2, only two EICs are shown within the EIM application on the far left. In actual applications, there will be hundreds to thousands of EICs within one EIM application. Each EIC will generally contain one to multiple executable finite state machines. Thus thousands of finite state machines are being designed and implemented.

Executable Specification

NASA is implementing the finite state machines using a modern graphical programming language called ControlShell by Real-Time Innovations [22]. ControlShell combines a graphical language with a component-based design methodology. The ControlShell finite state machine

Figure 2. Component Level View of End Item Manager and Intercomponent Communication

component type is semantically similar to the statecharts of the Unified Modeling Language (UML) [20]. The ControlShell finite state machine is built graphically and is executable. Its contents form a state transition diagram that represents an event-driven program. In developing the system, this ControlShell component is used by NASA both for modeling and for implementing dynamic behavior. Each finite state machine becomes an executable specification.

Object Oriented, State-based Design

EIMs are both object oriented and state-based. As seen in Figure 2, a Composite Object Group (COG) is a container that houses both atomic components (ATCs) and finite state machines. An ATC is simply a diagrammatic wrapper for a C++ class. A COG models a physical object, such as a power supply, and is used for capturing both the static and dynamic views of an object.

EIMs for a hardware subsystem may have a hierarchical structure, where one EIM controls the entire hardware subsystem with lower level EIMs controlling individual components of the hardware subsystem. For example, in a liquid oxygen (LO2) or liquid hydrogen system (LH2), one EIM may be in control of the entire LOX or LH2 subsystem, with separate EIMs controlling individual valves and pumps. EIMs receive requests from user displays (Command and Control Workstation in Figure 2) or other EIMs. EIMs also receive end item measurement information from a data distribution service and notification of constraint violations via the Function Designators.

Finite State Machine Thread-Centric Design

All of the essential processing for an EIM occurs within one thread of a process. The finite state machine thread-centric design of an EIM leverages the ControlShell tool by using the inherent, event driven mechanism of the finite state machine engine. Since the ControlShell finite state machine executes in a single thread, a design was sought that would minimize concurrency issues by taking advantage of this single thread of execution but still maintain an acceptable level of concurrency.

Each incoming CORBA command to the EIM arrives in its own thread. Immediately after arriving, the incoming CORBA command results in at least one event being posted to the finite state machine engine queue. The CORBA thread performs minimal work with its primary job being the posting of a message to the finite state machine engine queue. This results in an asynchronous stream of multiple CORBA requests being woven into one synchronous stream of events on the finite state machine queue. This simplifies the concurrency semantics of the design. Rather than having multiple CORBA threads performing work simultaneously, events are queued and consumed by the finite state machine in an ordered fashion. Concurrency is still possible but now C++ class methods in the ATC become atomic operations rather than low level assembly language calls. Thus, the finite state machine engine

manages both events and concurrency.

Component-based Design Details

While Figure 2 illustrates the connection between components, the lower left EIC2 Server COG contains an exploded view of the major components. The following sub-sections describe each of the primary components of an EIC COG.

Implementation Atomic Component—In Figure 2, the Implementation ATC provides the central location for the detailed implementation code of all of the methods and data associated with an EIC. All inputs and dependencies required to make the EIC functional are defined in this component.

The Implementation ATC houses atomic operations. An atomic operation is either an individual C++ method or set of methods that is not interruptible. The atomicity, or interruptibility, of an EIC operation is determined by the requirements defining that operation. Depending upon those requirements, the EIC operation will be decomposed into one to multiple ATC C++ methods. In turn, these methods are executed on various transitions within the finite state machine.

The Implementation ATC also has methods to implement the CORBA server which receives incoming CORBA messages from clients of the EIC, such as other EIMs or a Command and Control Workstation. A CORBA request results in one operation within the Command Dispatcher finite state machine being invoked. The Cmd_FSM_if interface is used to pipe the request to the Dispatcher. Within the Dispatcher, the Cmd_Stim_Splitter will deliver the message to the CMD_Stim_Sender. The CMD_Stim_Sender actually generates the finite state machine event. A CORBA server operation in the Function Designator Container may result in two stimuli being generated within the Command Dispatcher:

- The first stimulus, "new_cmd", causes the finite state machine to pop from its current level and return to the IDLE state. This stimulus is the one that causes the finite state machine to interrupt its current operation and begin processing the new command. Depending upon the requirements of the particular system, this leveling and interrupt scheme may be easily modified. Some systems may not want an operation to be interrupted and therefore want the incoming messages queued.
- The second stimulus, such as "oper_x", causes the finite state machine to enter the composite state that corresponds to the invoked command. For example, if a Shuttle Engineer at the Command and Control Workstation commanded a power supply to turn ON, the finite state machine would transition from the IDLE state to the composite operation (i.e. Operation X) and perform the necessary steps.

Finally, note the signal lines that connect the

Implementation ATC to the Command Dispatcher finite state machine. These signal lines represent constants, such as "max voltage", as called out in the software requirements specification.

Command Dispatcher—The Command Dispatcher finite state machine contains the primary finite state machine that models the reactive behavior of the EIC and provides a state machine implementation of all commands and reactions. The Command Dispatcher finite state machine contains three essential ATC splitter components plus the state graph.

- The Function Designator Splitter allows the finite state machine layer to invoke methods upon the Function Designator Container within the COG layer. In other words, the Function Designator Splitter allows the Command Dispatcher finite state machine and all Operation finite state machines to invoke simple commands on the Function Designators.
- The State Splitter allows the Command Dispatcher finite state machine to maintain and publish the hardware's state via the State Evaluator finite state machine. Typically, several function designator values and class attributes are fused to determine the hardware's state.
- The Cmd_Stim_Splitter pipes CORBA generated messages from the Function Designator Container to the Cmd_Stim_Sender. In turn, the Cmd_Stim_Sender generates an event to invoke an operation(s) within the stategraph.

Although operations within a finite state machine are interruptible, they execute sequentially. Therefore, to provide concurrency several Command Dispatcher finite state machines may be necessary within the EIC Server COG. The number of Command Dispatchers will generally equal the number of EIC operations that must execute concurrently ("and" states). For example, assume three Command Dispatchers are used and each contains five operations. One operation from each of those Dispatchers may be executing simultaneously. In this case, three operations can be executing concurrently. Any one of those three operations may be preempted by another operation within its finite state machine.

State Evaluator—The State Evaluator finite state machine instance is housed in the EIC Server COG, outside of the Command Dispatcher finite state machine. This design is necessary to allow the Command Dispatcher and the State Evaluator to operate concurrently. The EIC's state may be re-evaluated continuously as data changes, concurrent with the operation of the Command Dispatcher. This re-evaluation allows the published EIC state to accurately reflect the state of the hardware.

3. MODEL CHECKING

NASA is investigating the capability of a variety of automated software testing and validation technologies, including model checking, to support the achievement of the following project objectives:

- Increase the cost-effectiveness of test and verification activities.
- Detect errors early in the life-cycle.
- Detect subtle errors.

This section provides a brief introduction to automated model checking and discusses the applicability of model checking techniques to support achievement of the project objectives.

Model checking, also called finite state verification, is a collection of techniques used to verify specified properties of finite state software systems. These properties are formalized as correctness claims and include both safety (i.e. nothing bad will ever happen) and liveness (i.e. something good will eventually happen) properties such as absence of deadlock and livelock, respectively. Model checking is particularly useful for detecting subtle errors in finite state models.

Atlee and Gannon [1] demonstrated the utility of model checking the requirement specifications for two reactive software systems: an automobile's cruise control unit and a water-level monitoring system. Subsequent to this demonstration, model checking has been applied to other software requirement specifications with varying degrees of success. Although it has been actively researched for over two decades, model checking has not matured into a state-of-the-practice technique. Software practitioners' lack of exposure to model checking methods and the learning curve associated with the model checking technology are prime contributors to the low popularity of these tools. Model checking also suffers from the state explosion problem. Since software specifications often contain data types with infinite ranges, model checkers are frequently limited by execution time and/or space (i.e. memory) constraints. Various reduction techniques such as abstraction have been employed to produce tractable models from infinitely large state spaces [11].

Spin and Promela

One popular model checking tool, Spin [13], has been used for verification in domains with similar process control applications and event-driven systems [4, 5, 10, 16]. Spin is a finite state model checking tool used in the verification of concurrent systems and developed in the Bell Laboratories formal methods and verification group [13]. Spin uses a partial order reduction method in the verification of concurrent systems to curtail state explosion and reduce the size of the state space [14, 21].

Spin's input language, Promela (Process Meta Language), is a validation language with C-like constructs extended with

special primitives for representing processes and message channels [12]. Each process is translated into a finite automaton. The system specification is decomposed into one to multiple finite automata. Collectively, the finite automata represent the state space of the system.

A property of interest to the model verification effort is referred to as a correctness claim. Spin verifies the system's state space against the correctness claim, exhaustively checking for safety, invariance, and liveness property violations. After analyzing all reachable states, if no violations are detected, it can be stated that the correctness claim holds. If violations are detected, Spin produces a counterexample containing the sequence of reachable states leading to the property violation. SMV [2,18] is another model checking tool with comparable capabilities.

Processes specified in Promela can contain state variables and constructs with non-deterministic selection of sequence declarations. Before choosing a sequence or branch, Spin tests if it is executable. If more than one branch is executable, one is randomly chosen for execution. If none is executable, the process blocks.

Linear Temporal Logic

The correctness properties of a system can be formalized as correctness claims using Linear Temporal Logic [17] or as assert statements within the Promela model itself. A Linear Temporal Logic formula in Spin may contain propositional symbols, Boolean operators, and temporal operators. Spin verifies that a Promela model satisfies a Linear Temporal Logic property by converting the positive property into a negative property referred to as a "never claim". Never claims represent system behavior claimed to be impossible.

Applicability to the NASA Project

A significant portion of the NASA Checkout and Launch Control System process control applications software will be implemented as graphical, executable finite state machines using ControlShell. The ControlShell finite state machine is particularly useful for model checking because the validation model can be built against an implementation, not just a design model. This is superior to traditional uses of model checking that build a validation model against a specification model [1].

Spin and Promela are well suited for model checking ControlShell finite state machines. When a finite state machine contains states with multiple transitions that may be simultaneously enabled, the ControlShell engine non-deterministically selects one transition. Similar to Spin's blocking behavior, if none of the transitions emanating from a state is enabled, the machine remains in that state.

4. CASE STUDY

A case study was performed to demonstrate the efficacy of model checking on an actual finite state machine from the Checkout and Launch Control System. The case study's objectives were as follows:

- Evaluate the success of model checking methods in detecting a subtle error.
- Determine the difficulty or ease in manually constructing Promela models.
- Investigate opportunities for automating the construction of Promela models.
- Determine whether future investigation into model checking is warranted for the project.

The case study consisted of using Spin to create a validation model for one operational requirement within the Checkout and Launch Control System Space Shuttle Main Engine Ground Support Equipment software set. This section demonstrates and discusses the development of that validation model, describes the success of model checking methods as applied during the case study, and the opportunities for automating the construction of validation models.

Problem Description

Within the Shuttle main engine ground support equipment power bus, the turn off operation must asynchronously monitor various events (i.e. hardware indicators) and issue commands to effectuate the turning off of a power bus. If an "abort" event arrives while the operation is executing, the operation must discontinue the current sequence and return to the "Idle" state. Figure 3 shows the actual ControlShell finite state machine implementation for the power bus turn off operation. At the time the case study was performed, this implementation had not been formally tested or validated.

The design shown in Figure 3 contains one error that was detected by visual inspection. On the left side, the transition with the guard "abort" emanating from the TurnOffLevel is not specified as an override transition. Diagrammatically, an override transition would have contained a node at the root of the transition. Within ControlShell, an override transition changes the precedence that the ControlShell engine uses for processing transitions. By default, if all of the matching transitions are normal transitions, the one at the lowest level becomes active. If one or more of the matching transitions are an override transition, the one at the outer-most level becomes active. In this case, if the system engineer issued an abort command, the control application represented in Figure 3 might discard the command and continue processing as if the abort command was never sent. See Figure 4 and Figure 5 for an illustration of the implications of the override transition.

Figure 3. ControlShell Finite State Machine for the AC Power Bus Turn Off Operation

In the design of Figure 4, assume B becomes the current state just as the "abort" command arrives. Assume IndicatorValue equals OFF. By default, the "abort" event will be discarded since the transition from B to C has higher precedence than the transition from TurnOffLevel to E. When the B to C transition fires, the latched "abort" event is lost.

In the design of Figure 5, the transition from TurnOffLevel to E is specified as an override transition as indicated by the node at the root of the transition. Again, assume B becomes the current state just as the "abort" command arrives and IndicatorValue equals OFF. The transition from TurnOffLevel to E will fire since it has higher precedence than the transition from B to C. Thus, the "abort" command will be correctly processed.

The project team interpreted the leveling semantics of ControlShell as providing an immediate pop from the current state if the highest level transition's guard was true. The team assumed the precedence for processing transitions was from the "outside in", not "inside out". This precedence

assumption mirrored the precedence semantics as defined by classical statecharts such as Statemate [9]. In classical statecharts, higher level states take priority over lower level states. In support of this usage, the ControlShell vendor added the capability to establish either precedence rule at design time via the normal transition or the override transition. With this new capability, any transition specified as an override has precedence over other non-override transitions.

Many of the Checkout and Launch Control System designs, such as the one in Figure 3, have not yet been retrofitted with the override transition. Although visual inspection can readily identify these errors, NASA expects that automated verification may be more efficient. Thus, this problem was selected for the case study investigation of model checking capabilities.

Given the use of leveling, the override transition is required to prevent a fault. In the design of Figure 3, the fault results in the inappropriate discarding of an event, the abort command. This fault would unlikely occur during dynamic

testing and thus would probably be undetected. The subtle fault is exacerbated by several generic properties of finite state machines. The properties themselves are sound and typical of automaton theory.

Figure 4. Without Override Abort Transition

Property 1: Latched stimuli deactivated when a transition fires

Many of the states within Figure 3 contain transitions that may be immediately enabled upon entry to that state. For example, in state "busOnInd", one transition has the guard "::DISCRETE_OFF == busOnIndValue". This guard may be true as soon as the "busOnInd" state becomes active. If so, this guard's transition will fire and discard all other latched stimuli such as "abort". In ControlShell, a latched stimulus remains active until any transition is executed. At that time, all latched stimuli such as "abort" are deactivated. Since the inside "::DISCRETE_OFF == busOnIndValue" transition has precedence over the "abort" transition, if the "abort" stimulus arrived upon entry to the "busOnInd" state, it would have been deactivated (i.e. command lost) as soon as the higher precedence "inside" transition fired.

Property 2: Transitions execute rapidly

Even if the tester had the foresight to issue the abort command while the finite state machine was on a transition, instrumenting the code could be difficult. Transitions execute rapidly (on the order of microseconds) by design, and there is no appreciable delay. Thus the window of opportunity to inject the fault is narrow.

Property 3: States may represent lasting condition

By design, finite state machines spend most of their time (> 99.9%) in a state, not on a transition. In Figure 3, any state with a transition containing a guard of the form "ElapsedTime > some_constant" may result in the machine waiting several seconds for events to arrive. The states "busOnCmd", "busOnInd", and "VerifyRemoteModeInd" may result in appreciable waits. If the abort command arrives while the machine is in one of these states waiting,

the abort command will be processed correctly. Since the machines spend the majority of time waiting in a state, most events such as "abort" will arrive while the machine is in one of the wait states. Thus most of the time the "abort" will be processed correctly.

Figure 5. With Override Abort Transition

Collectively, these properties result in a very low probability of the aforementioned fault occurring during dynamic testing. The tester would have to force the abort command to arrive as the engine is in the middle of a transition step, not in a wait state. If the abort command arrived on a transition step, the fault would occur. However, the abort command arriving on a transition step is unlikely since transitions execute rapidly. As such, when the abort command is issued there is a very high probability that the engine in Figure 3 will correctly process the abort event and pass the test case.

This particular fault is dependent upon timing between events. Although the error is always present, the fault only manifests itself under improbable temporal conditions. Errors dependent upon timing are not only difficult to identify, but difficult to produce.

Even though it is possible for the abort command to be "inappropriately" discarded, the tester would unlikely encounter this errant behavior since the window of opportunity is so narrow. However, after thousands of hours of usage in the field, the error would likely result in a fault. Model checking is particularly effective at uncovering these type of problems since the model checker does not rely upon absolute timing among events. Rather, the model checker tool performs an exhaustive exploration of the state space attempting to prove or disprove the existence of particular orderings of events.

Promela Validation Models

Two Promela models were built: one for the faulty implementation shown in Figure 3 and another for the corrected implementation with the abort transition specified as an override. Figure 6 shows the message sequence chart for the three processes contained in both Promela models. The process on the far right represents the ac power bus hardware component. This process publishes hardware indicator values representing the state of the power bus and represents the environment model. This process also causes the entire model to be closed with respect to the environment. The process on the far left, the command and control workstation process, represents the console where the system engineer issues commands. In this example, two commands, abort and turn off, can be issued from the command and control workstation process. Finally, the turn off power ac process in the middle represents the reactive control process that accepts commands from the system engineer and receives measurements from the ac power bus.

stress the finite state machine under test, the turn off power ac process, as much as possible. Not only does this approach save time in not having to precisely model each and every process, but it also yields more comprehensive results.

Figure 7 shows the Promela fragment for the turn off power ac process without the override transition.

Correctness Claim Formulation

The following correctness claim was tested against both validation models:

If the system is not currently in the Idle state, the system shall always return to the Idle state immediately when an abort command is received.

To represent this liveness claim in Linear Temporal Logic, define the propositional symbols p and q as follows:

$p = \text{"abort"} = \text{true} \ \&\& \ \text{state} \neq \text{Idle}$,

$q = \text{"state"} = \text{Idle}$

Figure 6. Message Sequence Chart for Promela Model

Because only the turn off power ac process was being model checked, it was not necessary to create full models of the command and control workstation process and the ac power bus process. Thus, Promela models were developed for the other processes to randomly and exhaustively publish all combinations of commands and measurements possible. During real operations, some of these combinations of commands and measurements may be unlikely or procedurally prohibited. For example, during operations a system engineer has no reason to issue a turn off command if the ac power bus is already off. However, because model checking is automated, it was determined to be beneficial to

Using the temporal operators always, $[]$, and next, X , along with the Boolean operator implication, $\rightarrow$, the Linear Temporal Logic formula is: $[] (p \rightarrow X q)$

This formula states that whenever an abort event is dispatched, the machine transitions to the Idle state immediately after receipt of the event. Spin automatically converted the formula to a never claim. The never claim along with the propositions p and q were inserted into each model and the verification was run.


```

active proctype sme_power_ac() {
do
  :: atomic {
    if
      :: state == Idle -> check_msg_queue();
      if
        :: turnOff -> doTurnOff() -> state = busOnCmd;
        :: else -> skip;
      fi;
      :: state == busOnCmd -> check_msg_queue();
      if
        :: busOnCMDHRS.value == DISCRETE_OFF -> state = VerifyFailBit ;
        :: else if
          :: abort -> state = Idle;
          :: elapsed_time -> doBusOnCMDHRSCmdOFF() -> state = VerifyFailBit ;
          :: else -> skip;
        fi;
      fi;
      :: state == VerifyFailBit -> check_msg_queue();
      if
        :: busOnInd.failValue == DISCRETE_ON -> state = VerifyRemoteFailBit;
        :: busOnInd.failValue == DISCRETE_OFF -> state = BusOnInd;
        :: else if
          :: abort -> state = Idle;
          :: else -> skip;
        fi;
      fi;
      :: state == BusOnInd -> check_msg_queue();
      if
        :: busOnInd.value == DISCRETE_OFF -> state = VerifyRemoteFailBit;
        :: else if
          :: abort -> state = Idle;
          :: elapsed_time -> doBusOnIndIsOn() -> state = VerifyRemoteFailBit;
          :: else -> skip;
        fi;
      fi;
      :: state == VerifyRemoteFailBit -> check_msg_queue();
      if
        :: remoteModeInd.failValue == DISCRETE_ON -> state = Done;
        :: remoteModeInd.failValue == DISCRETE_OFF -> state = VerifyRemoteModeInd;
        :: else if
          :: abort -> state = Idle;
          :: else -> skip;
        fi;
      fi;
      :: state == VerifyRemoteModeInd -> check_msg_queue();
      if
        :: remoteModeInd.value == DISCRETE_OFF -> state = Done;
        :: else if
          :: abort -> state = Idle;
          :: elapsed_time -> doRemoteModeIndIsOn() -> state = Done;
          :: else -> skip;
        fi;
      fi;
      :: state == Done -> check_msg_queue();
      if
        :: DEFAULT -> doTurnOffEMON() -> state = Idle;
        :: else if
          :: abort -> state = Idle;
          :: else -> skip;
        fi;
      fi;
    fi;
    if
      :: transition_fired -> clear_msg_queue();
      :: else -> skip;
    fi;
  } /* end atomic */
od;
}

```

Figure 7. Promela Process for Turn Off Operation Without Override Transition

Model Checking Execution Results

Both simulations were executed on a 500 MHz Pentium III processor with 128 MB RAM using Spin version 3.4.0. For the model without the override transition, the verifier found a violation of the correctness claim in 7 seconds. An error trail leading to the property violation was automatically generated. Since the error trail was only 112 execution steps, it was possible to simulate the entire sequence leading to the violation. The simulation produced a window containing a message sequence chart and a window that tracked and monitored the program's variables. Both windows were useful for analyzing the behavior leading to the violation.

For the corrected model with the override transition, the verifier proved the correctness claim in 88 seconds. No violations were found.

As part of the investigation, both models were also simulated without regard to any correctness claims. This allowed the verifier to step through the entire state space and report any unreachable code. Unexpectedly, both models contained an unreachable code segment. In the "state == Done" sequence block, the "else if" construct was reported as unreachable for both validation models. This indicated that the "abort" transition could never be traversed (i.e. fire) from the Done state.

These findings were unanticipated. After further analysis, it was realized that the DEFAULT transition emanating from the Done state will always fire. The ControlShell DEFAULT transition is comparable to UML's completion transition without a guard [20]. It is a triggerless transition that is taken automatically. Thus, within ControlShell, it produces a transient state.

No errant behavior results from this anomaly since the immediate successor state is Idle. The model is not ill-formed. However, the unreachable code message suggests that the TurnOffLevel state does not have to contain the Done state. The Done state may reside outside the TurnOffLevel state without adverse effects. Consequently, an unanticipated result of the simulation was the discovery of a more efficient design.

Automated Construction of Promela Models

For the case study, the Promela model for the turn off operation was built manually from the diagrammatic implementation of the finite state machine in Figure 3. While the learning curve and the time to build the model was small, it would typically not be feasible to manually build and verify such models for each finite state machine in an industrial sized application. Automated construction of Promela models is essential if model checking techniques are to be scaled up to be useful on an industrial sized NASA project.

In traditional model checking, the Promela model is built from the specification model [1]. More recent research has shown the applicability of generating Promela models directly from UML statecharts [19]. Although ControlShell finite state machine diagrams are semantically close to UML statecharts, the ControlShell finite state machine diagrams are executable, whereas the UML statecharts are not. Therefore, for this case study, the Promela model was built directly from the system implementation. Other research has recently shown the suitability of automatically translating ANSI C [23] and Java source code to Promela [24].

Automated translation from ControlShell finite state machines to Promela models is straightforward since the ControlShell finite state machines and the Promela language share similar semantics, including non-deterministic behaviors. ControlShell stores all diagrammatic finite state machines as simple text files. It would be possible to build a parser to automatically extract a Promela model from this text file. However, the file contains information unnecessary for model checking, including all coordinate locations of the states and transitions and other visual modeling elements.

When the ControlShell application is executed, the tool provides a built-in run-time menu system, an interactive interface that is particularly useful for probing the application. One built-in command within the menu allows the developer to display, in textual form, the transition table for any finite state machine. Figure 8 shows a fragment of the actual state transition table for the finite state machine of Figure 3 as it was stored in the ControlShell engine.

States are preceded by the qualifier "CSFsmState:". The table indicates the outgoing transitions from each state. Each transition consists of the stimulus expression, return code, and the next state in the automaton. The ControlShell state transition table contains all of the essential information to automatically construct a Promela model for any ControlShell finite state machine. NASA is currently working to develop software to automatically construct Promela models from the implemented finite state machines.

5. CONCLUSIONS AND FUTURE WORK

With automated construction of Promela models from the ControlShell executable finite state machines and the use of Spin to automatically check the finite state machines, it is feasible to consider the use of model checking on industrial sized projects. The case study demonstrated the success of model checking in detecting subtle errors such as the error in the design of Figure 3. The case study's unexpected identification of unreachable code also demonstrated that model checking could be used to drive improvements in the design with respect to efficiency.

```

CSFsmState: Idle
EntryTC: {} ExitTC: {} "turnOff"/
{*doTurnOff}() [override]
-100002 -> Idle -> ... -> TurnOffLevel -> TurnOffLevel/busOnCmd ->
CSFsmState: TurnOffLevel
EntryTC: {} ExitTC: {} "abort"/
{}() [override]
-100002 -> TurnOffLevel -> ... -> Idle ->
CSFsmState: VerifyRemoteFailBit
EntryTC: {} ExitTC: {} "::DISCRETE_OFF == remoteModeIndFailValue"/
{*NULL}() [override]
-100002 -> VerifyRemoteFailBit -> ... -> TurnOffLevel/VerifyRemoteModeInd -> "::DISCRETE_ON ==
remoteModeIndFailValue"/{}() [override]
-100002 -> VerifyRemoteFailBit -> ... -> TurnOffLevel/Done ->

```

Figure 8. Fragment of ControlShell State Transition Table for Turn Off Operation Finite State Machine

Automated model checking of executable finite state machine models has the potential to increase the cost-effectiveness of test and verification activities, detect errors early in the system life-cycle, and detect subtle errors that might otherwise escape detection during inspection and test. Thousands of finite state machines are now being designed and implemented for NASA's Checkout and Launch Control System. With automated construction of Promela models from the implemented finite state machines and the leveraging of model checking technology on these implementations, a significant amount of automated verification is possible.

In addition to verifying correctness properties, NASA is also interested in investigating model checking for automatically generating test sequences to build a comprehensive test suite. Other organizations have reported success in such an application of model checking [3, 8]. For the case where model checking was already applied against an actual implementation in the Checkout and Launch Control System, it might appear that the generation of test sequences from the model checker would not add value. Presumably, all test sequences would pass if the model checker did not find any violations. However, as in the case study reported in this paper, the model checker may be applied against the top level application software, not the underlying system software. Even though the application software may have been model checked with respect to the correctness claims, the underlying system software was not model checked. Therefore, test sequences automatically generated from the model checker could be employed for dynamically testing the underlying system software. The cost to automatically generate the system software test suite could be minimal, given that the validation models already existed for the application software.

REFERENCES

- [1] Atlee, J.M., Gannon, J., "State-Based Model Checking of Event-Driven System Requirements," *IEEE Trans. on Software Engineering*, Vol: 19 1, pp. 24-40, Jan. 1993.
- [2] Burch, J., Clarke, E., Long, D., McMillan, K., and Dill, D., "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. on Computer-Aided Design*, Vol. 13, No. 4, April 1994.
- [3] Callahan, J., Easterbrook, S., and Montgomery, T., "Generating Test Oracles via Model Checking," NASA/WVU Software Research Lab, Fairmont, WV, NASA-IVV-98 015. Paper online at: <http://research.ivv.nasa.gov/docs/techreports/1998/NASA-IVV-98-015.pdf>
- [4] Cimatti, A., Giunchiglia, F., Mongardi, G., Romano, D., Torielli, F., Traverso, P., "Model Checking Safety Critical Software with Spin: an Application to a Railway Interlocking System," *Proc. of 3rd Spin Workshop*, Twente University, Enschede, The Netherlands, April 1997. Paper online at: <ftp://netlib.bell-labs.com/netlib/Spin/ws97/cimatti.ps.Z>
- [5] Duval, G., Cattel, T., "From Architecture Down to Implementation of Safe Process Control Applications-the Lift Case Study," *Proc. of the 30th Hawaii Inter. Conference on System Sciences*, Vol: 1, pp. 24 -33, 1997.
- [6] Dwyer, M.B., Avrunin, G.S., and Corbett, J.C., "Patterns in Property Specifications for Finite-State Verification," *Proc. of the 1999 International Conference on Software Engineering*, pp. 411 -420, 1999.
- [7] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1997.
- [8] Gargantini, Angelo, and Heitmeyer, Constance, "Using Model Checking to Generate Tests from Requirements Specifications," *Proc. of the 7th European Engineering Conference* held jointly with the 7th ACM SIGSOFT

symposium on Foundations of software engineering, pp. 146 - 162, 1999.

[9] Harel, David, and Naamad, Amnon, "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, Volume 5, Issue 4, pp. 293-333, 1996.

[10] Havelund, K., Lowry, M., Penix, J., "Formal Analysis of a Space Craft Controller using Spin," *Proc. 4th Spin Workshop*, Paris, 11/99. Paper online at: <ftp://netlib.bell-labs.com/netlib/Spin/ws98/p22.ps.gz>

[11] Heitmeyer, C., Kirby, J., Jr., Labaw, B., Archer, M., and Bharadwaj, R., "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications," *IEEE Trans. on Software Eng.*, Vol: 24 11, pp. 927-948, Nov. 1998.

[12] Holzmann, G., *Design and Validation of Computer Protocols*, Bell Labs, Prentice-Hall, NJ, 1991.

[13] Holzmann, G., "The Model Checker Spin," *IEEE Trans. Software Eng.*, Vol. 23, #5, pp. 279-295, 1997.

[14] Holzmann, G., and Peled, D., "An Improvement in Formal Verification," *Proc. Seventh FORTE Conference Formal Description Techniques*, Bern, Switzerland, pp. 177-194, Oct. 1994.

[15] <http://www.cis.ksu.edu/santos/spec-patterns/>

[16] Loeffler, S., and Serhouchni, A., "Creating a Validated Implementation of the Steam Boiler Control," *Proc. of 3rd Spin Workshop*, Twente University, Enschede, The Netherlands, April 1997. Paper online at: <ftp://netlib.bell-labs.com/netlib/Spin/ws97/loeffler.ps.Z>

[17] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.

[18] McMillan, K., *Symbolic Model Checking*, Kluwer, Boston, MA, 1993.

[19] Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G., "Implementing Statecharts in Promela/Spin," *Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pp. 90 -101, 1999.

[20] *OMG Unified Modeling Language Specification*, Version 1.3, June 1999.

[21] Peled, D., "Combining Partial Order Reductions with On-The-Fly Model Checking," *Proc. 6th Inter. Conf. Computer Aided Verification*, pp. 377-390, Stanford, CA, Lecture Notes in Comp. Science 818, Springer-Verlag, 1994.

[22] Real-Time Innovations, *ControlShell Version 6.0*

User's Manual.

[23] Holzmann, G., "Logic Verification of ANSI-C Code with Spin," *Spin Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, 7th International Spin Workshop, Stanford, CA, pp. 131 - 147, Aug. 30 - Sept. 1, 2000.

[24] Corbett, J.C., Dwyer, M.B., Hatcliff, J., and Robby, "A Language Framework for Expressing Checkable Properties of Dynamic Software", *Spin Model Checking and Software Verification*, Lecture Notes in Computer Science 1885, 7th International Spin Workshop, Stanford, CA, pp. 205 - 223, Aug. 30 - Sept. 1, 2000.

Glenn Semmel is an Application Software Test Lead for NASA. He serves on the Checkout and Launch Control System Project at the Kennedy Space Center where he develops and tests command and control software. He received a B.S. in Electrical Engineering in 1990, an M.S. in Engineering Management in 1994, and an M.S. in Computer Engineering in 1997 from the University of Central Florida. He is currently pursuing a Ph.D. in Computer Engineering.

Gwendolyn Walton is an Associate Professor of Computer Engineering at the University of Central Florida. She has over 20 years of industry experience in all aspects of the software life cycle. Her research interests include rigorous methods for software specification and testing. She received a B.S. in Mathematics Education in 1974, M.S. in Mathematics in 1977, and a Ph.D. in Computer Science in 1995 from the University of Tennessee.