

VALIDATION OF MISSION CRITICAL SOFTWARE DESIGN AND IMPLEMENTATION USING MODEL CHECKING

Paula J. Pingree
Jet Propulsion Laboratory
Pasadena, CA

Erich Mikk
Erlangen, Germany

Gerard J. Holzmann
Margaret H. Smith
Dennis Dams
Bell Laboratories
Murray Hill, NJ

Abstract

Over the years, the complexity of space missions has dramatically increased with more of the critical aspects of a spacecraft's design being implemented in software. With the added functionality and performance required by the software to meet system requirements, the robustness of the software must be upheld. Traditional software validation methods of simulation and testing are being stretched to adequately cover the needs of software development in this growing environment. It is becoming increasingly difficult to establish traditional software validation practices that confidently confirm the robustness of the design in balance with cost and schedule needs of the project. As a result model checking is emerging as a powerful validation technique for mission critical software. Model Checking conducts an *exhaustive exploration* of all possible behaviors of a software system design and as such can be used to detect defects in designs that are typically difficult to discover with conventional testing approaches.

Stateflow[®] by The Mathworks was used to develop the mission critical Fault Protection (FP) flight software (FSW) for NASA's Deep Space 1 (DS1) mission. Demonstrating the trend toward statechart modeling and auto-code generation, Stateflow has also been adopted for the FP FSW development on NASA's Deep Impact project, scheduled to launch in 2004. Both missions share a core component of FSW for which the design has been validated using Spin. Our aim is to validate mission-specific components of FSW that are specified using statecharts and used in the auto-code generation of the final flight code for the mission. We establish an automatic translation

procedure from Stateflow statecharts to Spin for the validation of the mission-specific components. To guarantee compliance with the generated code our translation tool set preserves the Stateflow semantics. We are now able to specify and validate portions of mission critical software design and implementation using exhaustive exploration techniques.

The DS1 Fault Protection System

System Description

Fault Protection is autonomous flight software that provides the robustness and autonomy needed to ensure survival of a space mission in the event of detected on-board failures.

The DS1 Fault Protection (FP) software architecture consists of three main components: the FP Engine, Monitors & Responses. These three components work together to notify the ground that a fault has occurred, determine the appropriate fault response(s), and orchestrate the execution of various responses.

Distributed throughout the flight software, the **FP Monitors** are designed to detect the symptoms of hardware and software failures. They perform symptom diagnosis that includes limit/persistence checking on the reported symptom.

The **FP Engine** services fault declaration by mapping symptoms to faults. The use of RED and GREEN status flags prevent the re-triggering of faults once they are detected and a response is in progress. The Engine also manages multiple message queues of different priorities to serialize the handling of faults, commands, requests, etc. The Engine services response execution by

mapping faults to responses. Only one response may execute at a time and there are separate queues for interruptible and non-interruptible responses. Interruption of an executing response is handled via a “waypoint” mechanism. This works as follows.

If an interrupting response is triggered while a non-interrupting response is in progress, the non-interrupting response is temporarily suspended until all interrupting responses have run to completion. Triggered responses go into one of two queues: the interrupting queue, and the non-interrupting queue. When each response comes up for execution, the current status of the monitor is polled and the response is executed only if the triggering monitor is still RED, otherwise the response is aborted.

A non-interrupting response is interruptible only at predetermined “waypoints” in the statechart. When execution of a non-interrupting response gets to a waypoint, a check is performed to see if there are responses in the interrupting queue. Only when all responses, if any, in the interrupting queue have executed to completion, does the non-interrupting response continue. After successful response execution, the Engine clears appropriate fault flags.

The **FP Responses** carefully command the spacecraft for successful fault recovery. Most responses for DS1 are two tiered. The first level of response either swaps to a backup device if available, or power cycles the faulty component while a fault that persists after the first tier response action may require spacecraft entry into one of three Standby Modes: SUN_STANDBY_SSA, SUN_STANDBY_SRU, or EARTH_STANDBY.

Successful entry into a Standby Mode puts the spacecraft into a power positive (e.g., solar arrays pointed to the sun) and communicative (e.g., antenna pointed toward Earth) state. The entry logic for each of these modes ensures that the required hardware is available to successfully enter that mode. For example SUN_STANDBY_SSA requires that the sun sensor assembly (SSA) complete a successful sun search. If the SSA has failed, thereby causing the sun search to “timeout”, the Standby logic will direct the spacecraft to SUN_STANDBY_SRU mode, which does not depend on the SSA to get the spacecraft power positive.

System Implementation

DS1 chose to implement model-based code-generation for the spacecraft system-level fault protection software. This was accomplished using Stateflow® and Stateflow Coder [RNC99]. This approach enforced standard diagrammatic conventions for representing statecharts and the resulting logic, thereby allowing systems engineers, rather than software engineers, design and implement the system. The mission-specific elements of the FP design, the Monitors and Responses, were developed in this manner.

Advancing Automation to Verification

The implementation of the Standby modes as a FP Response, for example, is complex. The statechart is detailed and contains several sub-charts with executions that are dependent on certain conditions and configurations of the spacecraft. It is an interesting challenge to validate such a system. Complications of response execution dependencies on resources make model checking an attractive method for this validation.

We chose to use the Spin model checker developed at Bell Laboratories by Gerard Holzmann for our verification effort [Hol97]. Spin accepts model representations of software systems in the language Promela. We desired a method that allowed generation of Promela from the same modeling source that produces the flight code, i.e., the statechart. Such an automated translation process more efficiently utilizes human resources and avoids human errors that are potentially introduced via manual coding. Adding a formal methods verification technique such as model checking improves the FP software development effort by providing a way to thoroughly test that the system does not violate its requirements and high-level design imperatives. The result that satisfies these desires and objectives is the automated translation method and tool set that we describe.

Our automated translation method is applied to the FP Response statecharts. Spin requires a closed system so we wrap an environment model around the translated FP Response statecharts. The environment model consists of an Abstract Spacecraft Model, which is automatically produced via a separate translation, and an Abstract FP

Engine, which is hand-written Promela. These environment model components will be described in detail later.

Stateflow Description [SFUG]

What Is a Finite State Machine?

A finite state machine (FSM) is a representation of an event-driven (reactive) system. In an event-driven system, the system transitions from one state (mode) to another prescribed state, provided that the condition defining the change is true.

Stateflow Representations

Stateflow uses a variant of the finite state machine notation established by David Harel [Har87]. Using Stateflow, you create Stateflow diagrams. A Stateflow diagram is a graphical representation of a finite state machine where states and transitions form the basic building blocks of the system. You can also represent flow (stateless) diagrams using Stateflow. Additionally, Stateflow enables the representation of hierarchy, and parallelism. Hierarchy enables you to organize complex systems by defining a parent/offspring (or child) object structure. For example, you can organize states within other higher-level states. A system with parallelism can have two or more orthogonal states active at the same time.

Notations

A notation defines a set of objects and the rules that govern the relationships between those objects. Stateflow notation provides a common language to communicate the design information conveyed by a Stateflow diagram.

Semantics

Semantics describe how the notation is interpreted and implemented. A completed Stateflow diagram illustrates how the system will behave. A Stateflow diagram contains actions associated with transitions and states. The semantics describe in what sequence these actions take place during Stateflow diagram execution. Knowledge of the semantics is important to make

sound Stateflow diagram design decisions for code generation. Different use of notations results in different ordering of simulation and generated code execution.

Stateflow Diagram Objects

This section describes many of the graphical and non-graphical objects in a Stateflow diagram along with the concepts that relate them.

A **state** describes a mode of an event-driven system. The activity or inactivity of the states dynamically changes based on events and conditions. Every state has a parent. In a Stateflow diagram consisting of a single state, that state's parent is the Stateflow diagram itself (also called the Stateflow diagram root). States have labels that can specify actions executed in a sequence based upon action type. The action types are entry, during, exit, and on. Stateflow provides two types of states: parallel (AND) and exclusive (OR) states. You represent parallelism with AND (parallel) states. Exclusive (OR) states are used to describe modes that are mutually exclusive.

A **transition** is a graphical object that, in most cases, links one object to another. One end of a transition is attached to a source object and the other end to a destination object. The source is where the transition begins and the destination is where the transition ends. A transition label describes the circumstances under which the system moves from one state to another. It is always the occurrence of some event that causes a transition to take place.

Events drive the Stateflow diagram execution but are non-graphical objects and are thus not represented directly in a Stateflow chart. All events that affect the Stateflow diagram must be defined. The occurrence of an event causes the status of the states in the Stateflow diagram to be evaluated. The broadcast of an event can trigger a transition to occur or can trigger an action to be executed. Events are broadcast in a top-down manner starting from the event's parent in the hierarchy.

Hierarchy enables you to organize complex systems by defining a parent and offspring object structure. A hierarchical design usually reduces the number of transitions and produces neat, manageable diagrams. Stateflow supports a

hierarchical organization of both charts and states. Charts can exist within charts. A chart that exists in another chart is known as a subchart. Similarly, states can exist within other states. Stateflow represents state hierarchy with superstates and substates. A transition out of a higher level, or superstate, also implies transitions out of any active substates of the superstate. Transitions can cross superstate boundaries to specify a substate destination. If a substate is active its parent superstate is also active.

A **condition** is a Boolean expression specifying that a transition occurs, given that the specified expression is true.

Actions take place as part of Stateflow diagram execution. The action can be executed either as part of a transition from one state to another or based on the activity status of a state. Transitions ending in a state can have condition actions and transition actions.

A system with **parallelism** has two or more states that can be active at the same time. The activity of each parallel state is essentially independent of other states. You represent parallelism in Stateflow by specifying parallel (AND) state decomposition. Parallel (AND) states are displayed as dashed rectangles.

Default transitions specify which exclusive (OR) state is to be active when there is ambiguity between two or more exclusive (OR) states at the same level in the hierarchy.

Connective junctions are decision points in the system. A connective junction is a graphical object that simplifies Stateflow diagram representations and facilitates generation of efficient code. Connective junctions provide alternative ways to represent desired system behavior.

Translation Method & Design

The authors of Stateflow adopted the graphical notation of Statecharts as proposed by D. Harel in [3] but designed a different semantics to this notation. The Statemate tool supports the original semantics developed by D. Harel and there are some advances that extend Statemate specified designs to model checking facilities [MLSH99].

The following list illustrates the differences in semantic design between Stateflow and Statemate Statecharts, which makes clear that we can not use Statemate-based tools for Stateflow Statechart verification.

- In Stateflow semantics there is at most one event active at a time. In Statemate semantics any finite number of events are allowed.
- Emitting an event in Stateflow semantics means to pass control to the receiver chart of the event just in the moment of emitting the event. In Statemate semantics events are collected until the end of the step and then broadcast to the entire chart.
- In Stateflow semantics the execution order of transition segments (they constitute transitions) is determined by their graphical placement: outgoing transition segments of one state are considered for execution in clockwise order. Hence, in Stateflow semantics it is not possible to select one enabled transition non-deterministically, as it is allowed in Statemate.
- In Stateflow the execution order within an AND-state is determined by the graphical placement of the AND-composed charts. In Statemate all AND-states are executed simultaneously.
- In Stateflow semantics the effect of changing variables takes place immediately. In Statemate semantics a variable change takes effect only at the end of the step.

We propose a new format, hierarchical sequential automata (HSA), that accurately reflects Stateflow semantics.

The translation idea is to associate to each Statechart a hierarchical sequential automaton that is semantically equivalent to the source Statechart. A hierarchical sequential automaton consists of a finite set of cooperating sequential automata that can be implemented as parallel processes in Promela (consider Figure 1). For the two-step translation the diagram in Figure 1 has to commute. Therefore we require an equivalence notion between the semantics of the source Statechart, intermediate HSA and the resulting Promela code (e.g. bisimulation).

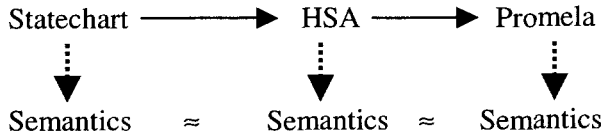


Figure 1. Two step translation of Statecharts

Here the translation of Statecharts to HSA and the semantics of HSA have to be defined using the same rigorous methods as was done by E. Mikk in [MLSH99, Mikk00]. This will be considered in further work.

The general idea of the translator-generated code is the following. There will be one Promela process for each OR-state (which corresponds to one automaton in HSA) such that the process implements its OR-state. Statechart states, events and variables are encoded as Promela variables and Promela processes change the values of these variables in order to simulate state changes, event generation and variable changes according to the semantics of Stateflow. The observable behavior is defined with respect to the variables representing Statecharts states, events and variables. These ideas are very close to the ones in the paper on translating Statechart Statecharts to Promela [MLSH99].

In this paper we discuss three additional topics beyond semantics and translation definition that are inevitable for practical verification setup:

- How to achieve a closed model for the code generated from the FP response specification in Statecharts? Here we manually extend the automatically generated code by user-written Promela code, which will model the environment. Both the model of the environment and its integration with the generated code must yield a closed system which is a valid model of the real system.
- What kinds of verification conditions are needed for the analysis of control systems designed in Stateflow statecharts? Here we introduce the notion of a responsive controller that eventually reacts to each environment input.
- We argue that some features of the Stateflow semantics might give rise to design faults that

are hard to detect because the semantics determine one specific execution order. We compensate for this by changing the translation of HSA to Promela such that every execution order is considered when a property of interest is verified. Formally speaking, this change is a generalization of the original semantics that means that we consider the original execution order but alternative execution orders as well.

Integration of user written Promela code with translated code

In this section we consider the flow of control between user-written Promela code and the translated code. The motivation is to allow user-written Promela code to model the environment of the translated code such that these two parts together constitute a closed system. We consider the following issues:

- The interface of translated code that allows the user-written code to pass control and data to the translated code.
- How to ensure mutually exclusive flow of control between user-written code and the translated code: whenever translated code is executed the user-written code waits for this execution to be completed and vice versa. Here we adopt the ideas of the synchrony hypothesis: the controller program (here represented by the translated FP response specification) reacts infinitely faster than the environment ("zero-time") and therefore the environment can be considered as "waiting" for the controller to complete. This construction is valid only if the controller program is responsive; i.e. eventually reacts to each environment input. We will discuss how we verify responsiveness.
- The definition of the so-called execution step of the translated code, i.e. the set of translations and actions taken and executed while active before the flow of control is passed back to the user-written code.

The interface of translated code

The translator generates for the given Statechart model the following Promela definitions.

- For every event of this model the translation creates a boolean variable in Promela code such

that when the variable is true the event is considered to be available in the current step. Since at most one event can be available at a time we can, alternatively, associate the presence of events to non-zero values of an integer variable (assuming the value 0 to represent "no events available"). This alternative approach is a memory optimization for verification runs. In the following treatment we assume that we have a boolean variable for each event.

- For every boolean or integer variable, that is read or written by conditions and actions of our model, the translation creates the respective variable in the Promela code. User-written Promela code can access these variables for reading and writing.
- For every OR-state there is one Promela process: *execute_state* that implements the states and transitions of this OR-state.

For passing control between the above-mentioned Promela processes there is a notion of top-down activation and bottom-up notification as explained in the following. There are activation and notification conditions for each Promela process of the generated model. Initially all Promela processes are waiting for their respective activation conditions to be satisfied. When this condition gets satisfied (e.g. set by the environment) the Promela process executes its respective code. For example, in the case of *execute_state* an enabled outgoing transition is taken. After that the process sets the notification condition of its activator and returns to the state of waiting for activation.

The following example shows the use of activation and notification for the implementation of the example in Figure 2.

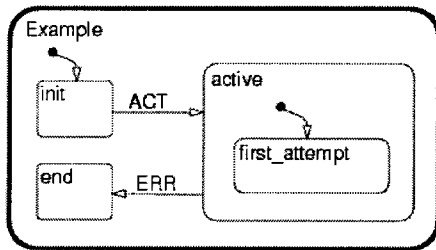


Figure 2. Activation & Notification

```

#define WAIT_ACT(st)           st?1
#define ACTIVATE(st)          st!1;
#define RESET_NOTICE(n)       n=0;
#define INC_NOTICE(n)         n++;
#define WAIT_NOTICE(n,i)      (n==i)

active proctype example_chart()
{
  loop: atomic{
    WAIT_ACT(act_example_chart) ->
    if
    :: cur_st_example_chart==state_init ->
      if
      :: (event_ACT) -> {
        cur_st_example_chart=state_active;
        /* activate sub-state */
        cur_st_active_chart=state_first_attempt;
      }
      :: else -> {
        skip;
        /* no during actions */
      }
    fi;
    :: cur_st_example_chart==state_active ->
      if
      :: (event_ERR) -> {
        cur_st_active_chart=notactive_example_state;
        cur_st_example_chart=state_end;
      }
      :: else -> {
        /* pass control to sub-state */
      }
    fi;
    goto loop;
  }
}

```

Coding conventions for integration of user-written Promela code with translated code

The coding conventions for integration of translated code with user-written Promela code are the following:

1. The user-written Promela code may set values of event, boolean and integer variables.
2. The user-written Promela code may pass control to the translated code *execute_root* by setting the activation condition of this function. Similarly, after activation the user-written Promela code waits for the corresponding notification condition to be satisfied. This is done using the pattern of the previous example.

Implemented in this way the flow of control between the user-written Promela code and the translated code guarantees mutual exclusiveness in their execution. While user-written code is executed the translated code waits for an activation condition to be satisfied and while the translated code is executed the user-written code waits for the notification to be satisfied.

Next we consider in more detail under which condition the control is passed back from the translated code to the user-written code.

Maximum step approach

We can distinguish between two kinds of behavior in the Stateflow model

1. Execution of actions as a result of taking transitions, or executing state actions associated with exiting and entering of states or to the situation where the control remains in the state it was in before execution (during actions).
2. No execution of actions because there are no enabled transitions and the set of actions associated with the entering and exiting of states or during action set is empty.

Here we follow the so-called maximum step approach: the control is passed back to the user-written code only if there is no execution of actions in the current step as described in point 2.

We detect this state as follows: we define a new boolean variable *action_executed* and initialize it to false. The intention is that whenever an action is executed, this variable is set to true. When the function *execute_root* is activated, it initializes this

variable to false. When the function *execute_root* completes the execution as described above, it tests the value of *action_executed*: when the value is true, the function resets the variable *action_executed* and continues its execution by activating the function corresponding to its direct active child. When the value is *false*, the notification condition of the caller of *execute_root* is established and the function *execute_root* moves to the waiting state for activation.

Responsive control program

Reactive systems are characterized by infinite interaction with the environment, e.g., in the case of Statecharts by receiving events from the environment and by responding to them. Since the FP responses designed in Stateflow are playing the role of the controller in a reactive system we are interested in verifying that the Stateflow model indeed interacts with its environment infinitely. We are interested in the verification condition that expresses the responsiveness property of the control program. A non-responsive controller might have design faults of the following kind:

- The controller program loops without responding to the environment and without accepting new inputs from the environment.
- The controller program executes no actions in response to the environment input and does not change its state.

As we learned in Section "Integration of user written Promela code with translated code" the concepts of activation of the controller and notification of the activator exist. We call a control program *responsive* if every execution passes the state where the notification condition is satisfied infinitely often. This property can be expressed as a formal Linear Temporal Logic (LTL) property and can be verified using Spin.

Verification of all execution orders

The semantics of Statecharts determine the execution order of transitions; this influences the execution order of actions associated with states, state activation and deactivation. One major benefit of graphical representations like Statecharts is that even complex systems can be described in a clear

way that is intuitive to both the reader and the designer of the specification. In the following we present some counterintuitive examples that demonstrate caveats to the designer and reader of the specification. We think these particular semantic features might give rise to human error. To compensate for this, we propose a more general verification that is not sensitive to these features of the semantics.

Outgoing transition segments of one state

Please consider the example in Figure 3.

Assume that in both charts state A is active and we consider its outgoing transitions. Due to clockwise execution order, the left Statechart moves from state A to state B but the right chart moves from state A to state C. The other transitions next in the clockwise execution order are not considered at all.

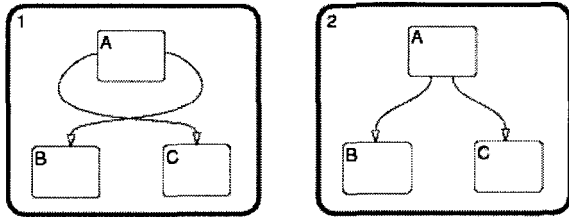


Figure 3. Different results of clockwise execution order

This example shows that a small change in graphical representation of the design idea might decide between a design fault and sound design. For rigorous verification we propose to change the semantics of Statecharts: instead of clockwise execution order we use non-deterministic execution order of outgoing transitions of one state. For the charts in Figure 2 this means that their behavior is equivalent: from state A one non-deterministically moves to state B or state C.

Execution order of AND-composed states

Design faults related to execution order might rise from the graphically determined execution order of AND-states are well. Consider the charts in Figure 4. The left chart executes its AND-states in the order ABDC but the right chart executes order ABCD; here the mutual position of state C and D make the difference. Note that if state B were one pixel higher than state A in the graphical

representation then the execution order of A & B would be swapped in both charts.

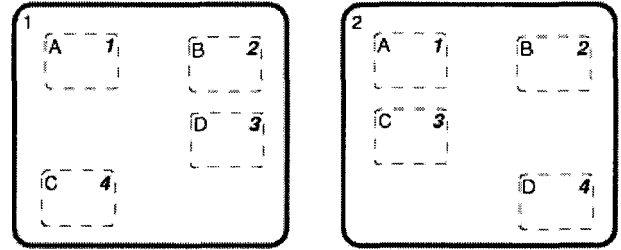


Figure 4. Example of AND-composition

Here, too, we propose to change the semantics for rigorous verification such that every possible execution order of AND-states is examined.

Backtracking order

Finally we consider backtracking sensitive execution order. Please consider the chart in Figure 5. Assume that condition C22 is false and all other conditions are true; assume that state A is active initially. Clock-wise execution order determines that transition labeled with C2 is evaluated and taken. Then C22 is evaluated to false which means that the execution has to backtrack [1]. If the backtracking strategy considers C21 first, state D becomes active. This strategy is implemented in Stateflow. However, if the backtracking strategy considers C1, then the execution moves finally to state C.

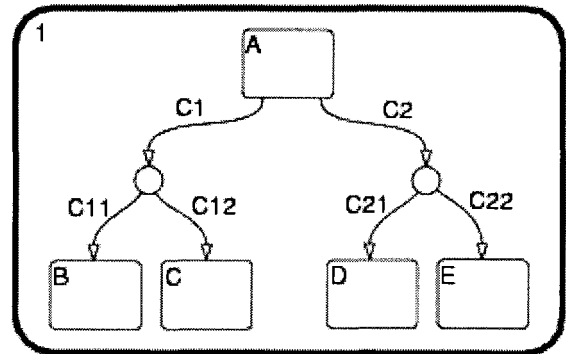


Figure 5. Example with backtracking behavior

Here, too, we propose to change the semantics for rigorous verification such that every possible backtracking order is examined.

Components of the translation tool set

We now describe the various programs of the tool set that achieve the translation presented in the previous Sections. There are three core programs in the translation tool set: SfParse, sf2hsa and hsa2pr. SfParse takes a Stateflow model file (.mdl) as input and parses from the statechart the relevant (non-graphical) data for our translation.

The sf2hsa program performs the translation of the parsed data set into the HSA format. As previously mentioned, HSA is an intermediate format that offers a small set of syntactical elements for defining the syntax and semantics of the powerful language of statecharts. It was developed primarily to promote the exchange of finite state

machine models between Stateflow and the Spin model checker.

The hsa2pr program translates one hierarchical sequential automaton to Promela, the input language of the Spin model checker. A support program, HSAMerge, allows multiple HSA files to be merged so that their translation produces an integrated Promela model of the input automata.

Prototyping for Mission Critical FSW

Figure 6 illustrates the Translation Tool Set and its interfaces with the closed-loop system components that maximize the Spin model checking results of our DS1 FP system design.

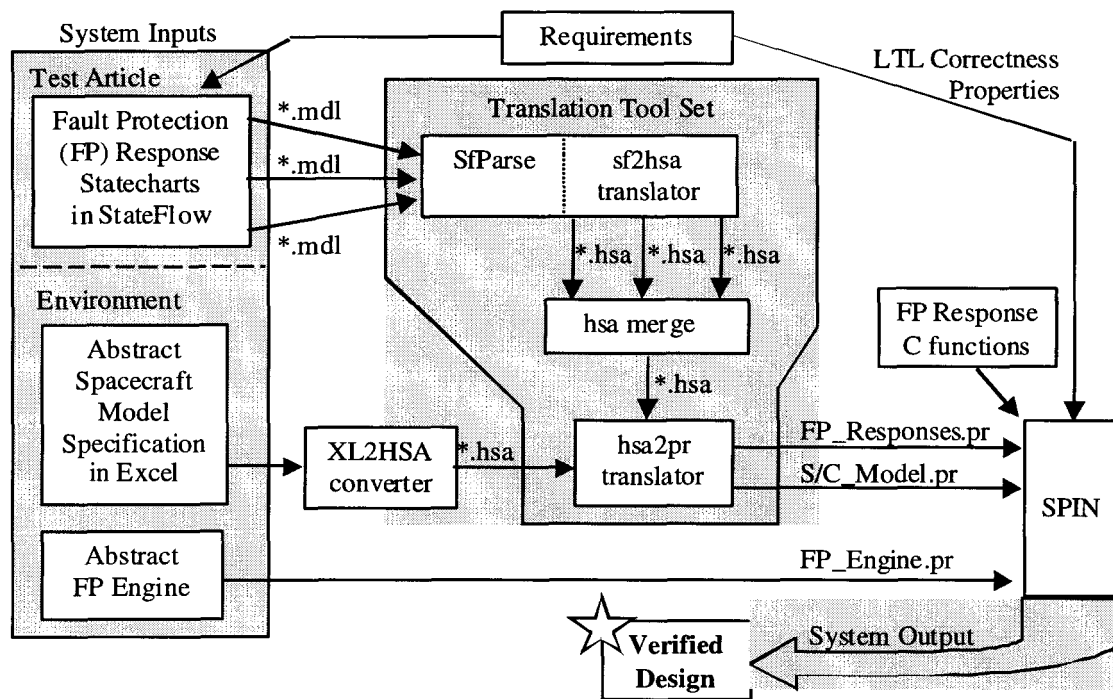


Figure 6. System Diagram for Fault Protection Model Checking Prototype

The FP Responses...the Test Article

We seek to demonstrate our approach on the set of DS1 Fault Protection Responses (described earlier) by translating each response .mdl file from Stateflow to HSA and then to Promela with our tool set. There are 25 unique response statecharts in the

DS1 FP design. Many of these charts are simple (5 or less basic states and transitions), yet some are quite complex (e.g., Standby – mentioned earlier, and the DS1 Launch sequence).

As an example demonstration of our prototyping technique we first translate two FP response statecharts: detumble & power_config, which is a subchart of detumble. The detumble

response performs the critical activity of stabilizing the spacecraft after separation from the launch vehicle or after any entry into a Standby mode where the flight computer is rebooted and inertial reference is lost. The power_config response performs power_cycling (off/on) for devices selected by the calling response. In the case of detumble, power_config power cycles the Inertial Measurement Unit (IMU) and the Propulsion Drive Electronics (PDE) to clear any faults in the hardware. Next we describe the environment model that we create to close the loop around the FP responses.

The Closed Loop Environment

To properly verify the FP Responses we provide an interactive closed-loop environment that emulates the response execution actions on the spacecraft. We introduce two abstract models; one of the FP Engine, the central component of the FP architecture and interface between the FP Responses and the spacecraft system, and one of the spacecraft itself. We abstract only those features of the FP Engine and spacecraft that are necessary for proper FP Response execution.

The **Abstract FP Engine** is hand-coded in Promela. It is primarily responsible for the non-deterministic scheduling execution of the FP responses. The following FP Engine features are also supported in our abstraction:

- Waypoints for permitted response interruption
- Tiered response execution
- Internal variable management

We abstract away the features that are not required to support FP response validation in our system such as symptom to fault mapping and fault to response mapping.

The abstract FP Engine provides for the distinction of 3 priority levels among executing responses. They are:

- Interruptible responses (low priority)
- Non-interruptible responses (medium)
- Ground-issued commands (high)

The following sample Promela segment selects the response to be executed based on these priorities.

```
active proctype fp_engine()
{
    pid r;

    do
    :: d_step {
        running < 0 ->
        if
        :: nempty(high) -> high?r
        :: else ->
            if
            :: nempty(medium) -> medium?r
            :: else ->
                if
                :: nempty(low) -> low?r
                :: else -> r = -1
            fi
        fi fi fi;
        running = r
    }
    od
}
```

We chose to capture the **Abstract Spacecraft Model** specification such that partial translation via HSA is permitted. The spacecraft model information consisting of variables, transitions, states, their hierarchy, and default status is captured in a simple tabular notation and formatted into HSA using a custom Perl script. Hence we are now able to auto-translate the spacecraft model from the intermediate HSA format to Promela using the hsa2pr program. This is a powerful extension to the original translation path directly from Stateflow-specified designs.

For our first example including detumble and power_config we specify the environment model based on the commands that are issued to the spacecraft as a result of response execution. These commands become the transition events between components of the spacecraft model. For example, when the power_config response issues the command IMU_POWER_OFF it declares the following transition event in the spacecraft model:



We specify hierarchy by indicating any parent/child relationships between components of the spacecraft, and we identify the default states of all components. We prove our technique for the detumble example to pave the way for specifying the full abstract spacecraft model that can close the loop around the entire set of DS1 FP responses.

Next Steps – Integration & Validation with Spin

The Promela model fragments that are generated from Stateflow and by the user can be integrated into a single verification model with the help of the FeaVer system [8]. For the generated models, we rely on the availability of a newly extended version of the Spin model checker (Spin version 4) that allows for the use of embedded C code fragments inside Promela code. Via this mechanism, the model checking code can be linked with original C code libraries that implement elements of the flight software that can be executed as atomic functions during the model checking exercise.

We are in the process of building a library of correctness properties for this code, from which we plan to derive formal Spin never claims that can then be verified mechanically by the model checker. The final system should allow us to check the Stateflow specifications within their intended context with a thoroughness that is virtually impossible to achieve by other means.

Once the design is validated we will also investigate how to correlate the validated design to the real implementation. The open issues to be investigated here are:

1. Does the code generation of Stateflow preserve the semantics of Statecharts?
2. Is our verification model (closed system) a conservative abstraction of the real system, i.e., are all errors of the design reproducible in the implementation?

We will report on the validation of the implementation and of the system design in future work.

Technology Infusion Plans

A JPL proposal has been submitted to apply our translation tools & methods for Spin model checking of the following state-based systems:

- Deep Impact (DI) FP Responses; design implemented in Stateflow; DI launches in January 2004
- Attitude Control Subsystem (ACS) Mode Commander; subsystem component used in many JPL spacecraft designs
- Mars Exploration Rover (MER) Surface Operations Behaviors (comprised of Activity Constraint Monitor & Resource Arbiter components); new & complex system to achieve mission goals; MER launches in June 2003, and arrives at Mars in January 2004.
- Mission Data System (MDS) Threading Policy; portion of execution architecture that coordinates software activities among threads with allocated resources (time, CPU utilization, message bandwidth, etc.); a revolutionary software architecture under development for future JPL flight missions including Mars Smart Lander, planned for launch in 2009.

References

- [Har87] D. Harel, Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, ASSP-34(2):362, 1986.
- [Hol97] G.J. Holzmann, The model checker Spin, *IEEE Trans. on Software Eng.*, 5(23):279-295, 1997.
- [HS00] G.J. Holzmann and M.H. Smith, Automating software feature verification, *Bell Labs Technical Journal*, 5(2):72-87, 2000.
- [Mikk00] E. Mikk, Semantics and Verification of Statecharts. PhD Thesis. *Technical Report of Christian-Albrechts-University in Kiel*, October 2000
- [MLSH99] E. Mikk, Y. Lakhnech, M. Siegel and G. Holzmann, Implementing Statecharts in PROMELA/SPIN. In *Proceedings of the 2nd IEEE Workshop*

on Industrial-Strength Formal Specification Techniques. pages 90-101. IEEE Computer Society 1999.

- [RNC99] N.Rouquette, T. Neilson, and G. Chen, The 13th Technology of DS1. In *Proceedings of IEEE Aerospace Conference, 1999*.
- [SFUG] The Mathworks Stateflow Users Guide, <http://www.mathworks.com>

Biographies

Dennis Dams is a member of technical staff in the Computing Principles Research Department at Bell Labs in Murray Hill, New Jersey. He works on methodologies and tools for computer-aided software verification. He received his Ph.D. degree in 1996 from Eindhoven University of Technology in The Netherlands (title thesis: "Abstract Interpretation and Partition Refinement for Model Checking").

Gerard J. Holzmann is Director of the Computing Principles Research Department at Bell Laboratories, in Murray Hill, NJ. He joined Bell Labs in 1980, after receiving his PhD from Delft University in The Netherlands. In 1995 he was appointed Distinguished Member of Technical Staff. Dr. Holzmann has done research in software verification techniques, requirements engineering, and computer graphics. He is best known as the author of the Spin model checker, one of the most widely used verification systems for distributed systems. Dr. Holzmann has published over 70 technical papers, authored 3 books, and holds 6 US patents.

Erich Mikk is a Principal Engineer in the Software and Engineering Department of the Siemens Corporate Technology Division in Erlangen/Germany. Currently he is involved in the design and implementation of engineering tools for systems design. Dr. Mikk obtained his Ph.D. degree in 2000 at the Christian-Albrechts University in Kiel. His thesis titled "Semantics and Verification of Statecharts" considers mathematical foundations and tool construction for model checking statechart specifications. Before obtaining his degree Dr. Mikk worked at the Christian-Albrechts University in Kiel in a research project that cooperated with

Daimler Benz on formal methods and test automation.

Paula J. Pingree is a Senior Staff Engineer in the Autonomy & Control Section at JPL. She has been involved in the design, integration, test and operation of several flight projects including Mars Observer, Cassini, Mars Global Surveyor, and Deep Space 1. She currently works on software systems engineering and technology infusion. Ms. Pingree holds a Bachelor of Engineering degree from Stevens Institute of Technology in Hoboken, NJ, and an MSEE degree from California State University, Northridge where she also teaches part-time in the Electrical & Computer Engineering Department.

Margaret H. Smith is a distinguished member of technical staff in the Scientific Computing Research Department at Bell Labs in Murray Hill, New Jersey, where she works on application of computer-aided verification tools and technology transfer. She received B.S. and M.S. degrees in industrial engineering from University of Michigan in Ann Arbor.