Architecture-led Diagnosis and Verification of a Stepper Motor Controller

Peter H. Feiler, Chuck Weinstock, John B. Goodenough, Julien Delange, Ari Klein, Neil Ernst

Architecture-led Diagnosis and Verification of a Stepper Motor Controller

Peter H. Feiler, Charles B. Weinstock, John B. Goodenough, Julien Delange, Ari Z. Klein, Neil Ernst
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA
[phf|weinstock|jbg|jdelange|azk|nernst}@sei.cmu.edu
Keywords: Architecture modeling & analysis, fault analysis, embedded software system

ABSTRACT

This paper discussed an architecture-led approach to diagnosing time sensitive issues with a stepper motor controller that manages fuel flow of an engine. A real engine control system design had originally been modeled and verified with SCADE®. The potential for missed steps that result in misalignment in the fuel valve position is difficult to test for and was not discovered until after the engine went into operation. We utilize the execution and communication timing semantics of AADL to architecturally characterize the interaction between the elements of the stepper motor control systems. We then characterize the functional behavior in the context of the task dispatch and input handling semantics using the AADL Behavior Annex and identify potential fault sources and their impact using the AADL Error Model Annex. The identified the potential error sources, early arrival and mismatched command rates, we quantify the condition for this to occur and analyze the system based on timing data from scheduling analysis and actual timing measurements. We use this analysis to evaluate several proposed design corrections.

I. INTRODUCTION

This case study shows how an analytical architecture fault-modeling approach can be combined with static analysis techniques to diagnose a time-sensitive design error in a control system and to verify that proposed changes to the system address the problem. The analytical approach demonstrates the value of the SAE Architecture Analysis & Design Language (AADL) standard [1] with its well-defined timing and fault behavior semantics in discovering hard-to-test errors and correcting them early in the life cycle, thereby reducing rework cost. This virtual system integration approach is a key element of an architecture-centric framework for improving the qualification assurance of software-reliant safety-critical systems [2].

In this case study, we investigated an actual stepper-motor system (SMS) that is part of an aircraft engine control system that manages fuel flow by adjusting a fuel valve. The original design was developed and verified in a model-based development environment called SCADE Suite®, and an implementation was tested on actual equipment. In some test situations, actual fuel flow did not correspond to the desired fuel flow. The failure was suspected to be due to execution time jitter in the stepper-motor control system, which resulted in some steps being missed. Missed steps were not immediately detectable by the controller to take corrective action. Two repairs were proposed to correct the problem, but there was little evidence other than testing that either proposed solution would address the problem of missed steps. We use the same analysis techniques to diagnose the problem and to determine whether the proposed design changes addressed the problem. A full case study report elaborates on how to manage the results of the diagnostic analysis and verification as an assurance case [3].

We first describe the SMS and characterize its architecture as an AADL model. We then diagnose the SMS in three steps: discuss the behavioral verification of the different SMS elements to eliminate computational errors; discuss the fault analysis of the SMS with focus on time sensitive issues, but also supporting a full fault impact analysis; and discuss a quantification of the timing condition under which the potential for missed steps can occur. We then proceed with applying this analysis to proposed design changes to correct the problem.

II. THE STEPPER MOTOR SYSTEM (SMS)

The stepper-motor control system operates open loop (i.e., there is no direct feedback on the successful execution of a step by the motor). The enclosing engine control system can detect deviation from desired fuel flow, but not at the granularity of individual stepper motor steps. In other words, the problem is not detected until multiple steps are missed.

The SMS is commanded to open the fuel valve in terms of a percentage with zero being closed and 100 being completely open. The stepper motor takes a known number of steps to move the fuel valve from a completely closed to a completely open position. The SMS is expected to reach the commanded position within a bounded time that is proportional to the distance between the current

position and the desired position. At command completion the stepper motor is expected to have reached the commanded position closest to the requested opening percentage.

The position control system (SM_PCS) operates periodically, and converts the percentage requested into the desired position in terms of stepper motor steps. It then commands the actuator to move the stepper motor a specified number of steps in the open or close direction. The maximum number of steps that can be taken per frame is a function of the frame rate and the time it takes the motor to move one step. To move the fuel valve as quickly as possible to the new position—that is, in a time roughly proportional to the number of steps required to move from the current position to the desired position—the position-change command sequence passed to the actuator consists of a sequence of maximum step count commands followed by a single command with the remaining steps less or equal to the maximum step count.

SM_PCS maintains a record of the desired position and the position to be reached through the most recent position-change command (commanded position). On completion of the position command, the desired position, commanded position, and actual position of the motor are expected to be the same. A *homing* command (to a fully closed position) is executed during initialization to synchronize the actual position with the initial desired and commanded position assumed by the SM_PCS.

The SMS may receive a new command before it completes the previous command. SMS is expected to immediately respond to the new command (i.e., immediately moves the fuel valve to the most recent commanded position without first continuing to the previously commanded position). As we will show this immediate response feature is not the culprit. Instead, the behavior of the actuator is sensitive to command arrival timing.

The SMS was implemented according to the above design and it was discovered that the expected location of the stepper motor deviated from the actual location of the stepper motor over time (that is, steps were missed). The functionality of the stepper motor had been modeled and verified using SCADE®, but without detecting the potential missed steps problem.

III. SMS Modeling in AADL

The SMS was modelled in AADL in three levels of abstraction:

- SMS as the system of interest in its operational context to capture the commanded input and expected result of the controlled system,
- the runtime architecture of the SMS as a set of interacting tasks to capture the execution and communication timing semantics of the implementation to analyze the time-sensitive nature of the problem (Figure 1),
- a specification of SMS component states and functional behavior that provides the basis for quantifying the conditions under which some commanded steps may be missed.

The SMS and the Engine Control System (ECS) are represented as AADL system components. This allows us to decompose the SMS as necessary to elaborate its architecture. The ECU is represented as an AADL processor, and the device bus for transferring data between any sensor, the actuator, and the processor as an AADL bus. The power supply is also modeled as an AADL bus, in this case transferring electricity. The fuel valve is represented as an AADL device.

The SMS consists of three components: digital position control software for the stepper motor SM_PCS, an actuator SM_ACT that translates commands from the position control software into electrical signals to a stepper motor, and the stepper motor SM_Motor. Note that the commands received by SM_ACT (*Commanded_Position*) take the form of a step count to be completed within a frame in the original system design. The interface between SM_ACT and SM_Motor (*SM_Command_Signals*) is represented by a feature group, i.e., a collection of event ports. The mechanical interface of the stepper motor (*Mechanical_Control_Position*) is represented by an abstract feature, i.e., a feature without specific communication timing semantics that are associated with ports.

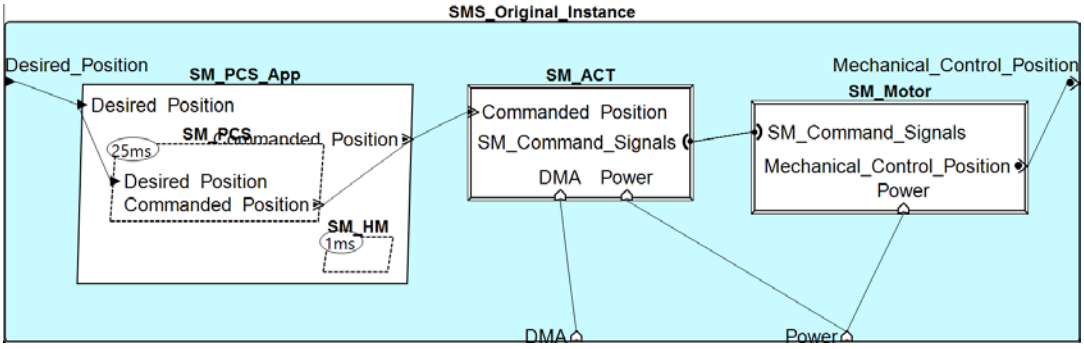

Figure 1: The SM_PCS Architecture

The position control system software SM_PCS is an AADL thread with a period of 25ms that resides in an AADL process called SM_PCS_App. The figure also shows a health monitor (SM_HM) thread with a period of 1ms in the same process that has no logical

interaction with SM_PCS. This indicates that the two threads share the same address space; thus, a coding error in one can potentially affect the other.

A Binding property in the operational environment of SMS, which contains the ECU, indicates that SM_PCS and SM_HM execute on the ECU. A Priority property indicates that SM_HM takes precedence over SM_PCS, thus, can affect the completion time of SM_PCS due to preemption. A Scheduling_Protocol property on the ECU indicates that preemptive scheduling is used.

The SM_ACT and SM_Motor are modeled as AADL devices to reflect that they are separate physical components. We can specify the execution and communication timing behavior of devices the same way as for threads. We specify that the actuator responds immediately to a new command from SM_PCS. We reflect this in the AADL specification of SM_ACT (Figure 2) by the *aperiodic* dispatch protocol. The *Queue_Size* and *Overflow_Handling_Protocol* properties allow us to specify details of handling arrival of input (see Section IV.B). Similarly, the stepper motor immediately responds to each step signal from the actuator – expressed by *aperiodic* dispatch.

```
device SM_ACT
features
-- logical interface
  Commanded_Position: in event data port StepCount { Queue_Size => 0;
    Overflow_Handling_Protocol => Error; };
  SM_Command_Signals: feature group inverse of SM_Command_Signals;
-- physical interface
  DMA: requires bus access DirectAccessMemory;
  Power: requires bus access Power_Supply.Volt28;
flows
  flowpath : flow path Commanded_Position -> SM_Command_Signals;
properties
  Dispatch_Protocol => Aperiodic;
end SM_ACT;
```

Figure 2: The SM_ACT Interface Specification

IV. Diagnosis of the SMS

We diagnose the SMS problem in three steps. First, we establish a record of the functional behavior verification and its assumptions. Second, we utilize the fault modeling capability including a taxonomy to identify potential fault contributors and refine the AADL model to more fully capture the timing behavior of the SMS. Third, we formalize the time-related condition under which the problem can occur.

A. Behavioral Verification of the SMS

The original behavior specification was modeled and verified in the SCADE Suite®. This verification asserts that the desired, commanded, and actual positions of SM are correctly maintained. However, verification tools such as the model checker in SCADE® or Simulink® make assumptions about synchronous execution behavior and timing. Typically, execution timing is separately verified through scheduling analysis and benchmark measurements of code. For example, TAXYS [4] combines verification and code generation from Esterel models (predecessor to SCADE) with timing verification based on timed automata. Recent research in verification of time-sensitive applications includes model checking of specifications extracted from source code [5] and verification of time sensitive behavioral constraints of AADL models annotated with BLESS [6]. We have created a BLESS specification of SMS, but were faced with the challenge to represent with multiple execution rates and aperiodic execution behavior.

To better understand the effects of execution and communication timing of the different elements of SMS affects functional behavior we elaborate the AADL specification with AADL Behavior Annex (BA) annotations [7]. These annotations specify how the functional behavior originally specified in SCADE interacts with the execution and communication timing behavior of the SMS components. This mapping into the AADL tasking model allows us to diagnose where the synchronous execution model of SCADE is potentially violated.

The behavior specification of SM_PCS is shown in Figure 3 and is equivalent to the SCADE model. It maintains two states: *DesiredPositionState* to represent the target position received by the last command, and *CommandedPositionState* to represent the position resulting from the execution of the last command sent to SM_ACT. It states that on every periodic dispatch SM_PCS will

1) check whether a new *Desired_Position* command has arrived, validate that the value is within the expected range of 0 to 100, and set it to be the *DesiredPositionState*; an out of range command is ignored.

2) compute a step count up to a maximum of 15 steps per frame (SPF) in the appropriate direction to move the *CommandedPositionState* towards the *DesiredPositionState*; and issue the appropriate step count command to SM_ACT; this results in a series of 15 step command, with the last non-zero count possibly less than 15, and then a count of zero once the desired position is reached.

```
thread implementation SM_PCS.impl
subcomponents
  DesiredPositionState: data SM_Position;
  CommandedPositionState: data SM_Position;
annex Behavior_Specification {**
  variables
    distance: Base_Types::Integer;
    stepcount: Base_Types::Integer;
  states
   Ready: initial complete state;
  transitions
    Ready -[on dispatch]-> Ready { -- on every 25ms dispatch
    -- check for new command and out of range if a new command has been received
    if ((Desired_Position'fresh = true) and (Desired_Position >= 0 )
        and ( Desired_Position <= PCSProperties::MaxPercent)){
    -- convert from PercentOpen to Steps
      DesirePositionState := PCSProperties::MaxPosition*Desired_Position/100
    } end if;
    distance := DesiredPositionState - CommandedPositionState ;
    if (abs(distance)> PCSProperties::MaxStepCount)
      stepcount := PCSProperties::MaxStepCount
    else
      stepcount := abs(distance)
    end if;
    if (distance>0){
      Commanded_Position := stepcount;
      CommandedPositionState := CommandedPositionState + stepcount
    } else {
    -- this case handles steps in the close direction as well as zero steps
    -- note that zero step commands are expected to be issued
      Commanded_Position = - stepcount;
      CommandedPositionState = CommandedPositionState - stepcount
    } end if;
    Commanded_Position!;
  }; -- end action
**};
end SM_PCS.impl;
```

Figure 3: Behavior Specification of SM_PCS

Both the SCADE model and the AADL BA specification of SM_PCS have been verified to correctly send a command sequence to SM_ACT. The verification also showed that the arrival of a new desired position command before the previous desired position has been reached is handled correctly, i.e., that SM_PCS responds within one frame to the new command and sends a command sequence to actually reach the latter desired position. The verification results show that the SCADE and BA specifications reflect the same behavior with respect to the verified invariant.

Furthermore, we have used an end to end flow specification in the AADL model and performed latency analysis [8] to determine that the system responds "immediately" to the new command, the amount of time it takes for SMS to respond to the new command, i.e., that it responds within less than two frames as shown in Figure 4. This latency includes the one frame sampling delay by SM_PCS and the processing and communication time by SM_PCS and SM_ACT to issue the first step command to the motor.

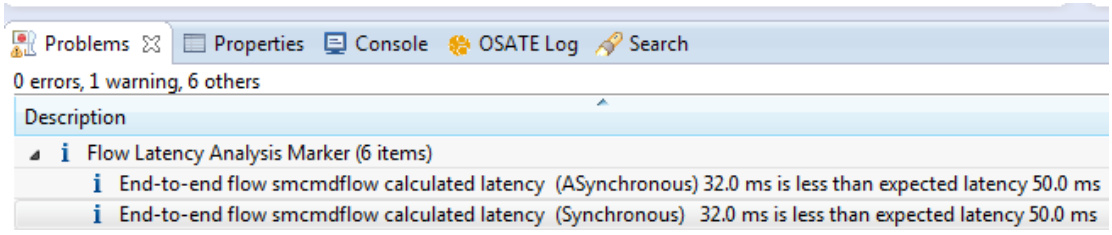

Figure 4: End-to-End Latency Analysis Results for New SMS Command

The specification of the actuator as a physical device is a little more interesting in terms of timing. In this case, we reflect in the BA specification that SM_ACT responds to two inputs, the arrival of the next step count from SM_PCS and the completion of a step from the motor. This is determined by the specification of SM_ACT as an aperiodic thread with a queue size of zero for arriving commands. The SCADE model had assumed that the commands arrive at the frame rate of 25ms and that at that time the previous command had been completed.

The actuator maintains system state in the form of a persistent *StepsToDo* count as shown in Figure 5. This count is set to the step count value as soon as a command is received from SM_PCS. The actuator then sends individual step signals to the motor at the specified rate until the count is zero. For that purpose, the specification is characterized by three states:

1. *Ready*, indicating that it is waiting for a command from SM_PCS
2. *WaitOnStep* to indicate that the execution of a step by SM is in progress
3. *Decide* as an intermediate state dealing with the decision of whether there are steps left to be executed by SM.

Arrival of a *Commanded_Position* in the form of a step count is handled by the *Ready* state and the *WaitOnStep* state. *StepsToDo* is set to the newly arrived value. A positive value results in increasing the stepper motor position, while a negative value results in a decrease. The first transition out of the *Decide* state determines *that* no step has to be taken. The other transition out of the Decide state specifies whether an *Increment_Step* or *Decrement_Step* signal is to be issued according to the specified Direction and updates the step count. The transition out of the *WaitOnStep* state triggered by the step completion signal leads to the *Decide* state, which determines whether additional steps are to be performed.

```
device implementation SM_ACT.impl
subcomponents
  StepsToDo: data StepCount;
annex Behavior_Specification {**
  states
    Ready: initial state;
    WaitOnStep: complete state;
    Decide: state;
  transitions
    Ready -[on dispatch Commanded_Position]-> Decide {
      StepsToDo := Commanded_Position
    };
    WaitOnStep -[on dispatch Commanded_Position]-> WaitOnStep {
      StepsToDo := Commanded_Position
    };
    WaitOnStep -[on dispatch SM_Command_Signals.StepDone]-> Decide ;
    Decide -[StepsToDo = 0]-> Ready ;
    Decide -[StepsToDo > 0]-> WaitOnStep {
      If StepsToDo > 0
        StepsToDo :=  StepsToDo - 1;
        SM_Command_Signals.DoIncrement!;
      else if StepsToDo < 0
        StepsToDo :=  StepsToDo + 1;
        SM_Command_Signals.DoDecrement!;
      end if
    };
**};
end SM_ACT.impl;
```

Figure 5: The Functional Behavior of the Actuator

As mentioned above the SCADE model assumed that the previous command had completed, i.e., the *StepToDo* count is zero at the time the new command arrives. The verification of the BA specification shows that in the *Ready* state the *StepsToDo* count is always zero. In the case of *WaitOnStep* the count may be non-zero unless the last step has been issued. The verification shows that if the precondition of a zero *StepsToDo* count does not hold the number of individual step commands to the motor will deviate from the incoming step count command sequence.

B. Fault Analysis of the SMS

We use a fault taxonomy that is part of AADL Error Model Annex specification [9,10] to systematically identify potential contributors to missed steps. We do so by annotating every incoming and outgoing port with error propagation types as shown in Figure 6. The Error Model Annex comes with fault taxonomy to identify omission, commission, value, timing, rate, sequence, replication, concurrency, authentication, and authorization error types. The notation lets us use guidewords relevant to the domain as aliases to the more abstract terms used in the taxonomy, e.g., missing command as alternative to omission.

Since the functional behavior of SM_PCS has been verified we specify that certain errors are not expected to be propagated. Figure 6 shows the actuator assuming that the step count is within range and that the command does not reflect an incorrect position. For SM_PCS we also specify that incoming commands with out of range values are mapped into *missed commands* to reflect the behavior specified in Figure 3.

To diagnose the problem, we focus on timing related error propagations. We specify that early or late command delivery or a command sequence at an incorrect rate may occur (see Figure 6). Late command delivery delays the commanding of SM_ACT and indirectly the stepper motor, i.e., we specify an incoming late delivery is propagated as outgoing slow response by the stepper motor. Similarly, arrival of commands at a rate lower than expected results in slower response by the stepper motor.

```
annex EMV2 {**
use types SMErrorTypes;
error propagations
  Commanded_Position : in propagation { MissingStepCountCommand, TimingError, RateError};
  Commanded_Position : not in propagation { StepCountOutOfRange, IncorrectPosition};
  SM_Command_Signals : out propagation {MissingStepCommand, SlowResponse, NoCommandSequence};
  ElectricalPower : in propagation {PowerLoss};
flows
  MissingCmd: error path CommandedPosition{MissingCommand} -> SM_Command_Signals{MissedSteps};
  LateCmd: error path CommandedPosition{LateDelivery} -> SM_Command_Signals{SlowResponse};
  EarlyCmd: error path CommandedPosition{EarlyDelivery} -> SM_Command_Signals{MissingStepCommand};
  Fast: error path CommandedPosition{ HighRate} -> SM_Command_Signals(MissingStepCommand);
  Slow: error path CommandedPosition{ LowRate} ->  SM_Command_Signals(SlowResponse);
  MechanicalFailure: error source SM_Command_Signals{NoCommandSequence} when {ActuatorFailure};
  NoPower: error path Power{PowerLoss} -> SM_Command_Signals{NoCommandSequence};
end propagations;
**};
end SM_ACT;
```

Figure 6: Fault Propagation Specification for the Actuator

Early command arrival at SM_ACT is the interesting case. The specification of SM_ACT in Figure 2 indicates that input is not queued (*Queue_Size* of zero in Figure 2). In other words, SM_ACT responds to the arrival immediately. The BA specification in Figure 5 shows that the new step count is assigned to *StepsToDo* at arrival. As discussed in the previous section this potentially leads to overriding a non-zero count. We reflect this behavior in the SM_ACT interface specification by the *Overflow_Handling_Protocol* property value of *Error* for the incoming port of SM_ACT (see Figure 2). Based on this observation we proceed in the next section to define the condition in terms of time for which the *StepsToDo* count is non-zero.

The fault propagation specification for SM_ACT shown in Figure 6 includes physical failures as well, e.g., the mechanical failure of the actuator device, and the loss of electrical power from a power source external to the actuator (and SMS). Once we have completed such fault propagation specifications for each of the SMS component we can perform a fault impact analysis [11] and identify other potential contributors to missed steps or other stepper motor malfunction. Figure 7 shows a portion of a fault impact report for SMS.

Component	Initial Failure Mo	1st Level Effect	Failure Mode	second Level Effect	Failure	third Level Effect
SM_ACT	{ActuatorFailure}	{NoCommandSe	SM {NoCommandS	{NoService} MechanicalControl -> SMS_Original_Instance:Mechan		
SM	{StepperMotorFai	{NoService} MechanicalControl -> SMS_Original_Instance:MechanicalControl [External Effect]				
SM_PCS_App.SM_PCS	{TimingError}	{TimingError} Cc	SM_ACT {LateDelive	{SlowResponse} SM_Cor	SM {Sl	{LateDelivery} MechanicalControl
SM_PCS_App.SM_PCS	{TimingError}	{TimingError} Cc	SM_ACT {EarlyDeliv	{MissingStepCommand}	SM {M	{MissedStep} MechanicalControl -
SM_PCS_App.SM_PCS	{RateError}	{RateError} Com	SM_ACT {HighRate}	{MissingStepCommand}	SM {M	{MissedStep} MechanicalControl -
SM_PCS_App.SM_PCS	{RateError}	{RateError} Com	SM_ACT {LowRate}	{SlowResponse} SM_Cor	SM {Sl	{LateDelivery} MechanicalControl

Figure 7: Fault Impact Report for SMS

C. Formalized Time Sensitve Fault Condition

Once we understand the issue of overriding a non-zero *StepsToDo* count, we can quantify how early a command must arrive for this condition to occur. The latest time for a non-zero value is when the last step of a position-change command is issued. Since the actuator is a reactive component, the maximum early arrival time between two commands from SM_PCS must not exceed the difference of the period of 25ms and this time limit.

This leads to the following condition that must be satisfied in order to avoid a missed step. The maximum early arrival time for commands arriving at SM_ACT must be less than the time difference between the latest time for a non-zero step count value and the next frame, which we refer to as *AStepMissBound*..

```
Max(EarlyArrivalTime) < StepMissBound
```

The maximum early arrival time is determined by the maximum early send time by SM_PCS and variation in communication time. Figure 8 illustrates how the maximum early send time is determined. The time difference between a send at maximum completion time followed by a send at minimum completion time is the frame time minus the delta between maximum and minimum completion time. Thus, the early send time corresponds to this delta. This is reflected in the formula

```
Max(EarlyArrivalTime) = Delta(CompletionTimeSM_PCS) + Delta(Comm)
```

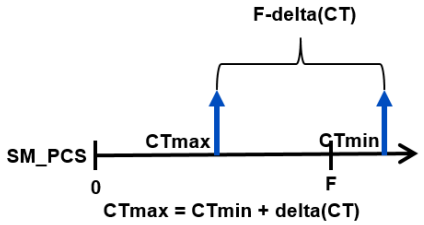

Figure 8: Maximum Early Send Time to Actuator

The value of step count is non-zero until the last step of a position-change command has been issued (i.e., until (Step_Count -1) * Step_Duration). This results in a worst-case *step miss bound* for the maximum acceptable variation of inter-arrival time of

```
StepMissBound = 25ms – ((MaxStepCount -1) * max(Step_Duration))
```

According to the stepper motor specification the step duration varies between 578 (1.730ms step duration) and a maximum of 621 steps (1.61ms step duration). Using the maximum step duration as worst case, this results in a *step miss bound* of 0.78ms.

For a particular set of execution times for SM_PCS and SM_HM we can calculate the inter-arrival time variation. The worst-case send time variation corresponds to the variation in execution completion time for SM_PCS. The completion time can be determined by a scheduling analysis and confirmed by actual timing measurements of executing code. Effectively, the completion time is determined by the variation in actual execution time of SM_PCS plus any preemption variation by SM_HM. Note that preemption variation by SM_HM may be multiples of SM_HM execution time as the minimum and maximum number of preemptions may differ. The resulting number is compared to the bound of 0.78ms to determine the possibility of missed steps.

A rate mismatch between the sender SM_PCS and the receiver SM_ACT can occur for two reasons:

- SM_PCS executes at a rate faster than the specified 25ms frame rate
- SM_ACT requires more than 25ms to complete the execution of the position-change command.

SM_PCS can execute faster than 25ms if the hardware clock of the ECU operates faster. For our case study we assume that this is not the case.

Notice that this bound is less than the minimum duration of one step execution. In other words, the last step may not complete before the end of the 25ms frame. Recall that the data sheet for the stepper motor indicates that the stepper motor's step duration varies between 578 and 621 steps per second when executing at the rate of 15 steps per frame. This results in a SM_ACT completion time variation between 24.15ms and 25.95ms for performing 15 steps. In other words, SM_PCS may send commands at a higher rate than SM_ACT is processing them.

In a worst-case scenario, the stepper motor could continuously operate at the longest step duration, falling behind 0.95ms for every frame that executes a position-change command of 15 steps. The completion delay is cumulative for a sequence of consecutive maximum step count commands. Note that SM_PCS sends the maximum step count only until the desired position is reached. A step count less than the maximum allows the stepper motor to catch up with the SM_ACT commands and make up for the time delay.

This leads to a derived requirement for the SMS implementation.

```
max(StepDuration) * MaxStepCount * max(MaxStepCountCommandSequenceLength) < StepMissBound
```

It takes 16 commands (250/15) to go from a completely closed to a completely open position with a cumulative delay of 15.2ms. This number is larger than the step miss bound, leading to missed steps.

To make matters worse, the maximum step count sequence potentially can be longer. A new desired position may be issued before the previous one has been reached. The new position may be in the opposite direction from the current position. As a result, the sequence of maximum step commands can be larger than 16 step count commands. In this case the cumulative delay may exceed a frame, resulting in missing a complete step count command.

V. Verification of Alternative Designs

Two possible design changes had been proposed to address the missed step problem. The first proposed design change was to minimize output jitter by the SM_PCS by a second periodic thread sending the output at a fixed offset from its dispatch time (*Fixed Send Time* solution). The offset was chosen to be half the period as this would allow an implementation with a single thread executing

at double the rate to alternate between computing the step count and sending the resulting command. In the AADL model we chose to specify a second thread with an offset start time. When analyzing this design alternative, we determine that early arrival time has been reduced but not eliminated. However, we have additional thread dispatches. Furthermore, we have to assure that the computation of SM_PCS completes before the chosen output time. Otherwise, the old step count value may be sent again. Finally, the solution does not address rate errors.

The second proposed design change was for the actuator to buffer the incoming command until the execution of the previous command is completed (*Buffered Command* solution). This can be accomplished by actually buffering the command or by adding the incoming step count to the *StepsToDo* count. If we choose command buffering it is natural to assume a queue size to be sufficient since commands are issued at the frame rate. However, the rate error analysis has shown that there is potential for cumulative delay. The addition of the step count to the *StepsToDo* count requires more complex functional logic in the actuator device. For this solution, we can verify that the count is updated correctly, even when the direction changes. The solution is less sensitive to rate errors. Cumulative delay reduces the responsiveness of the stepper motor.

In the original design and the two proposed design changes SM_PCS sends a step count to the actuator. This step count represents a state change, i.e., the difference between the current state and a new state. The full fault analysis considers the potential of data corruption or loss when the step count is communicated to the actuator. Communication of state change is sensitive to such faults, i.e., results in missed steps or execution of an incorrect number of steps. Therefore we consider an architecture design of SMS where the state, i.e., the target position is communicated to the actuator.

In in this design the desired position is validated by SM_PCS and then passed to the actuator. The functional logic of the actuator is slightly more complex, i.e., it has to compare to values instead of testing a single value for zero. However, the complexity of SM_PCS is significantly reduced. We have eliminated the functional logic of SM_PCS to transform the desired state (Desired Position) into a sequence of state changes (step count). Note that a design that operates with state changes assumes guaranteed communication and execution of every state change. In other words, it is sensitive to malfunction such as incomplete execution or data corruption during data transfer. An architecture design that communicates the desired state more robust, e.g., transient data corruption during transfer does not lead to a permanently inconsistent state.

In the case study we have considered not only logical design defects in the SMS, but also the effect of other contributors to missed steps. Table 1 presents a comparison of the four architecture design alternatives in terms of their full fault analysis. The first row focuses on logical failures in the SMS design, the second row describes mechanical failures within the SMS, the third row captures the effects of computer hardware on the SMS, and the last row represents mechanical failures in the operational environment.

The comparison shows that the position-commanded actuator design is not sensitive to early delivery or high rate errors, nor is it sensitive to transient message corruption or loss, while the original design is sensitive to transient data corruption. This is due to the design choice of commanding the actuator by desired position rather than by a sequence of position-change commands.

We can also see that mechanical failures affect the SMS the same way in all designs and must be addressed at the enclosing system level (e.g., by replication of the engine control system and the engine).

Missed Step	Original Design	Fixed Send Time	Buffered Command	Position Command
SMS logical failures	EarlyDelivery HighRate	HighRate	HighRate	
SMS mechanical failures	ActuatorFailure StepperMotorFailure	ActuatorFailure StepperMotorFailure	ActuatorFailure StepperMotorFailure	ActuatorFailure StepperMotorFailure
Transient comm failures	MessageCorruption MessageLoss	MessageCorruption MessageLoss	MessageCorruption MessageLoss	
Mechanical failures in Op Environment	ECUFailure PowerLoss ValveFailure	ECUFailure PowerLoss ValveFailure	ECUFailure PowerLoss ValveFailure	ECUFailure PowerLoss ValveFailure

Table 1: Comparison of Architecture Design Alternatives

VI. Conclusion

The purpose of this case study was to show how architecture fault modeling and analysis can be used to diagnose a time-sensitive design error encountered in a control system and to investigate whether proposed changes to the system address the problem. The analytical approach demonstrates that such errors that are hard to test for can be discovered and corrected early in the life-cycle, thereby reducing rework cost.

The case study example is a stepper motor controller to manage the fuel flow of an engine. Its original design had been verified with SCADE® without discovering until system integration and operational testing the potential for missed steps due to variation in command inter-arrival time. The use of models to capture the behavior of a system and their verification through simulation or model checking is an established practice. For time sensitive applications these models assume a particular execution model, e.g., a periodic

sampling processing model with deterministic sampling behavior. Scheduling analysis is used to assure that a set of tasks are schedulable, i.e., the tasks meet their deadline. Application code and the runtime executive may be generated and configured from such verified models to ensure consistency. The resultant system still goes through system integration and operational tests.

We have presented an architecture-led approach that is more comprehensive in utilizing model-based analysis early in development and as diagnostic tool. Our unique contribution is to complement the above mentioned techniques with a combination of AADL BA, and EMV2, to represent the system, utilize a fault taxonomy to identify potential faults, and quantify timing related faults.

AADL captures a specification of the task and communication behavior of software as well as hardware devices in AADL that captures both synchronous and asynchronous system execution behavior of software and physical devices. BA associates functional behavior specification with the task and communication model, which allows us to identify mismatched assumptions about execution and communication timing semantics. We utilize a fault taxonomy and EMV2 annotations of AADL models to identify potential issues and analyze their impact throughout the system. We quantify timing related conditions that violate a behavioral assumption about command completion and utilize scheduling analysis result of variability between best case and worst-case completion times to assess whether this condition can occur.

To diagnose the time sensitive nature of the problem we have captured the original SMS architecture design and three design alternatives in AADL, the Behavior Annex, and the EMV2 Annex, and quantified a timing related condition due to early rather than late arrival times that allows us to analytically assess whether the condition can actually occur.

The ability of AADL abstractly capture the dispatch and input handling of software threads like the stepper motor controller and physical devices like the actuator helped us focus on the essential architecture aspects of the system. The ability to specify the functional behavior of each component in the context of its dispatch and input handling behavior allowed us to recognize command execution is aborted due to the fact that a counter is set to a new target value even under circumstances when it contains a non-zero value. We have applied the fault taxonomy of the AADL Error Model Annex to perform a full safety analysis that includes logical design errors as well as physical errors. This fault taxonomy includes error types that deal with the time-sensitive nature of systems, both in terms of early or late arrival and in terms of mismatched arrival rates. We have been able to quantify the condition for timing related faults. We have then used results from scheduling analysis or actual timing measurements to analytically determine whether and when the missed step failure can occur. We have shown that in addition to early arrival, rate mismatch can lead to missed steps in the operation of the stepper motor.

We have applied the analysis to the original design as well as the three design alternatives. Three design communicate state change, i.e., the number of steps to be performed, while the fourth communicates state, i.e., the target position. During fault analysis we have identified system designs that involve state change to be more sensitive to transient faults such as data corruption or message loss. They result in persistent incorrect state for the receiver. When communicating complete state repeatedly, transient data corruption and message loss is limited to transient effects. This leads to a more robust system design.

VII. References

1. Peter H. Feiler, David P. Gluch, Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. Addison-Wesley Professional, 2012.
2. Peter H. Feiler, John B. Goodenough, Arie Gurfinkel, Charles Weinstock and Lutz Wrage, "Four Pillars for Improving the Quality of Safety-Critical Software-Reliant Systems," April 2013. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=47791.
3. Peter Feiler, Charles Weinstock, John Goodenough, Julien Delange, Ari Klein, and Neil Ernst, "Improving Quality Using Architecture Fault Analysis with Confidence Arguments," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-2015-TR-006, 2015.
4. Bertin, V.; Closse, E.; Poize, M.; Pulou, J.; Sifakis, J.; Venier, P.; Weil, D.; Yovine, S., "TAXYS=Esterel+Kronos. A tool for verifying real-time properties of embedded systems," Proceedings of the 40th IEEE Conference on Decision and Control, 2001.
5. Sagar Chaki, Arie Gurfinkel, Ofer Strichman, Time-Bounded Analysis of Real-Time Systems, Proceedings of Formal Methods in Computer-Aided Design (FMCAD), 2011.
6. Brian R. Larson, Patrice Chalin, John Hatcliff: "BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software." Proceedings of the 2013 NASA Formal Methods Conference, pp. 276-290.
7. ”SAE Architecture Analysis and Design Language (AADL) Annex Volume 2, Annex D: Behavior Model Annex”. SAE standard AS5506/2. SAE International, 2011.
8. Julien Delange, Peter Feiler. Incremental Latency Analysis of Heterogeneous Cyber-Physical Systems. In Real-Time and Distributed Computing in Emerging Applications (REACTION) 2014.
9. ”SAE Architecture Analysis and Design Language (AADL) Annex Volume 1A, Annex E: Error Model V2 Annex”. SAE standard AS5506/1A. SAE International, 2015.

10. Julien Delange, Peter H. Feiler, Architecture Fault Modeling with the AADL Error-Model Annex. 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2014).
11. Julien Delange and Peter Feiler, Supporting the ARP4761 Safety Assessment Process with AADL Proceedings of Embedded Real-Time Software and Systems 2014 (ERTSS2014), Toulouse, France. February 2014.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0002917