A Safety Condition Monitoring System

John Knight[1], Jonathan Rowanhill[2], Jian Xiang[1]

[1] University Of Virginia, Department of Computer Science, Charlottesville, USA
{jck, jx5c}@virginia.edu
[2] Dependable Computing LLC, Charlottesville, USA
jonathan.rowanhill@dependablecomputing.com

Abstract. In any safety argument, belief in the top-level goal depends upon a variety of assumptions that derive from the system development process, the operating context, and the system itself. If an assumption is false or becomes false at any point during the lifecycle, the rationale for belief in the safety goal might be invalidated and the safety of the associated system compromised. Assurance that assumptions actually hold when they are supposed to is not guaranteed, and so monitoring of assumptions might be required. In this paper, we describe the Safety Condition Monitoring System, a system that permits comprehensive yet flexible monitoring of assumptions throughout the entire lifecycle together with an alert infrastructure that allows tailored responses to violations of assumptions. An emphasis of the paper is the approach used to run-time monitoring of assumptions derived from software where the software cannot be easily changed.

Keywords: safety argument · safety assumption · safety condition monitoring

1 Introduction

The *Comprehensive Lifecycle for Assuring System Safety* (CLASS) is a safety-engineering system lifecycle that extends the Assurance Based Development software concept [1][2] to the system level. CLASS encompasses system development, approval, maintenance, and decommissioning. An important element of CLASS is a system for *monitoring* safety assumptions. In this paper, we present the overall design of the monitoring system together with details of one complex part of the system, sensor technology for monitoring run-time assumptions in software where the software cannot be easily changed.

The safety analysis that is undertaken when developing a new, safety-critical system is *predictive*. The goal is to provide an estimate of the residual risk that remains as a result of the system's design, the planned operational context, and the planned mission profiles. In classical safety analysis, a variety of techniques are used to provide an estimate of the residual risk and associated variance. In deployment decisions for safety-critical systems the decision is based, in part, on assessment of whether the estimated residual risk value and variance exceed that which is determined to be acceptable.

Inevitably, all safety analyses depend upon assumptions or expectations about: (a) the system being analyzed, (b) the way that the system was built and approved, and (c) the way that the system will be used. Such assumptions are made about topics such as:

- Conduct and reporting of development processes and analysis.
- Details of the context within which the system will operate, including ranges of input variables.
- Failure rates and failure semantics of physical and software components.
- Performance of components in terms of physical capabilities such as strengths of elements, wear resistance, and corrosion resistance.
- Performance of components in terms of computing capabilities such as computing rates, data transmission rates, and data generation rates.
- Aspects of human performance in areas such as operator fault rates and response times.
- Maintenance timing and expected application of maintenance procedures.

Assumptions about topics such as these are often stated explicitly in safety arguments or are implied by statements such as operational limitations. Irrespective of the source, all assumptions become part of the rationale for belief in a safety goal. In the event that an assumption is false or becomes false once the system is deployed, the rationale for belief in a goal within a safety argument might be invalidated and the safety of the associated system compromised.

In summary, the effectiveness of any safety-engineering activity, and in particular the effectiveness of CLASS, relies upon two conditions:

- The detailed lifecycle (development, approval, maintenance, and decommissioning) activities are conducted as defined.
- The assumptions used in the safety analysis of the subject system (and therefore the assumptions contained both explicitly and implicitly in the system safety case) are true throughout the lifecycle of the subject system.

These two conditions imply predicates on the activities and state of a system throughout the lifecycle. These predicates must maintain their assumed values, and the role of monitoring is to check the values of the predicates. For purposes of discussion, we refer to these predicates as *lifecycle invariants* or simply as *invariants*.

Adherence to these conditions cannot be dictated, and so a fundamental aspect of CLASS is to support selective monitoring of both conditions. More specifically, monitoring in CLASS plays two roles:

- *Process Monitoring*. Process monitoring supports monitoring of adherence to the processes and procedures employed throughout the lifecycle.
- *State Monitoring*. State monitoring supports monitoring of adherence to the system state assumptions made about the system artifacts in the lifecycle analysis.

The application of monitoring has to be selective in order to control overhead, and has to be adjustable over time to control measurement granularity. System safety analysis

and observation of operation will determine which process and state elements are monitored, when, and with what frequency.

The CLASS Safety Condition Monitoring System (SCMS) is the mechanism that implements the necessary monitoring. The SCMS is identical for all lifecycle invariants. Only the platform, the sensors, and the alerts that are used are different.

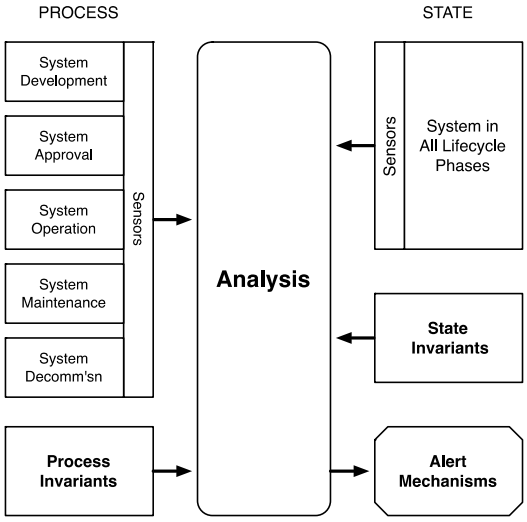

Fig. 1. CLASS Safety Condition Monitoring System

It is important to note that the monitoring system does *not* monitor hazardous states or violations of safety requirements. The requirements for the monitoring system derive from the creation and content of both the process used to create the system and the safety argument for the system, *not* the safety requirements of the system. The system monitors the conditions upon which the safety argument depends so as to facilitate continuous justified belief in the safety claim for the target system.

The overall structure of the monitoring system is shown in Figure 1.

2 Monitoring System Design

Frequently, development environments for safety-critical systems and the systems themselves are distributed, consisting of a number of components operating independently. Each such component often implements more than one service. Such system architectures lead to the need to: (a) monitor a number of different system elements, and (b) to integrate the results of analyses in order to ascertain the state of complex conditions.

The monitoring system accommodates this system architecture by operating as a distributed system with the various elements of the system communicating in a manner determined by the structure and details of the invariants being monitored. Thus, for example, the SCMS could be monitoring invariants derived from maintenance

requirements and also supplying maintenance measurements to state invariants that are defined in terms of expected maintenance activities.

As an example, the distributed structure of an SCMS and how the SCMS might be integrated into a simple avionics architecture is shown in Figure 2. In this example: (a) sensors monitor a variety of applications, (b) predicates local to each application are evaluated, (c) predicates distributed across the applications are evaluated, and (d) data needed elsewhere in the SCMS (other instances) is transmitted as necessary.

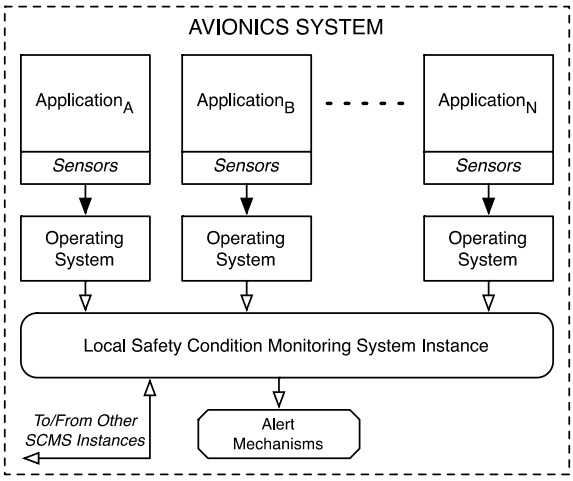

Fig. 2. Example CLASS monitoring system in a hypothetical avionics system.

The design of the monitoring system is shown Figure 3. The design assumes that all requisite sensors have been deployed through the relevant environments with which the system has to operate. During development, these environments would include, minimally, asset libraries used in the system's development, the subject system's design and analysis documentation, the process and workflow definitions, and the various assets in use for development. After deployment, these environments would include the system's operational, maintenance, and decommissioning environments.

2.1 Event Bus

Central to the design of the monitoring system is the *Event Bus* (see lower center of Figure 3). The Event Bus accepts event notifications from any part of the monitoring system and delivers those notifications to any destination within the system. The purpose of the Event Bus is to provide a comprehensive, asynchronous notification mechanism. Thus changes sensed within one part of the monitoring system that require action elsewhere result in event generation, transmission, delivery and processing.

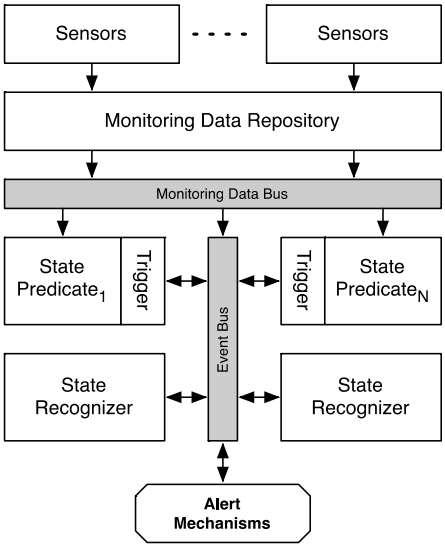

Fig. 3. The monitoring system design

2.2 Monitoring Data Repository

Sensor data will arrive at intervals determined by the monitoring system but no polling of sensors is assumed. Separate scheduling and timing control is assumed to be provided either by the host operating system or the sensors themselves.

Sensor data is placed into the *Monitoring Data Repository* as the data becomes available. As appropriate, events are generated by agents within the data repository to signal the availability of sensor data.

Clocks within the system are treated as sensors so that logical timing information is maintained within the Monitoring Data Repository. The passage of time that triggers sensing is made known to the remainder of the monitoring system as necessary by the generation of events.

2.3 State Predicates

The *state predicates* codify the safety conditions, and the monitoring system forms all predicates so that the assumed value is always *true*. Thus, evaluation of a state predicate to false indicates that a safety condition has been violated. State predicates are documented using the standard operators from predicate and propositional logic with data values from the Monitoring Data Repository, including time.

As an example, consider the development of a UAS that is to be operated within the NAS. In the safety case for the UAS operation within the NAS, an invariant might be required that the altitude of the UAS would not exceed FL 300. That assumption might be encoded in a state predicate from an original invariant as:

```
altitude <= 300
```

Since the UAS' altitude is available from onboard sensors, checking this invariant merely requires acquisition of the current altitude and comparing that altitude with the constant 300.

State predicates have to be evaluated when suitable data is available for their evaluation and when evaluation is meaningful. A predicate referred to as a *trigger* controls each state predicate (see Figure 3). A trigger encodes details such as: (a) arrival of relevant data in the data repository, (b) arrival of time for evaluation, and (c) system state as determined by other state predicates requiring evaluation of the subject state predicate.

The evaluation of a state predicate to *false* causes the predicate to generate a *token* that is transmitted to the state recognizers.

2.4 State Recognizers

The *state recognizers* encode the alert semantics that the system stakeholders require for violations of any of the system's invariants. The *state* of interest is any sequence of invariant violations that requires some action. Possible actions include:

- **No action**. There might be circumstances in which system stakeholders decide that violation of an invariant does not impact the system's safety.
- **Indicate the violation to a system operator**. Alerting an operator will allow human intervention should that be indicated for the invariant violation.
- **Change monitoring parameters**. Violation of an invariant might be best handled by more extensive or more detailed monitoring of the state. Thus, an action that might be required is adjustment of the parameters controlling a subset of the sensors or adjustment of the trigger(s) for one or more state predicates.
- **Modify the state of the system**. A violation might be sufficiently serious that the preferred response to a violation is to modify the development state or the operating state of the subject system, such as suspending development or shutting down all or part of the system.
- **Record details of the violation**. The response to violation of an invariant might depend upon prior violations. To accommodate such sequential actions, a necessary action might be merely to record details of a violation so as to modify the action taken on future violations of invariants.

To deal with the variety of actions that might be required from a state recognizer, the state recognizers are designed as *finite-state machines* and the actions they must take are defined with regular expressions. Inputs to the finite-state machines are the tokens generated by the state predicates. Each token that is generated is supplied to the subset of finite-state machines that have registered an interest in the type of token.

Every action arises from a transition in a finite-state machine and is implemented as an event. Each action event is sent to the required destination.

An example of a simple finite-state machine of the type used in the monitoring system is shown Figure 4. The example is for the state monitoring of a hypothetical UAS that is constrained to operate below FL 300 and at speeds less than 200 knots. The

safety argument's validity depends upon the assumptions that these limits are respected. Systems analysis has determined that:

- A single violation of either assumption only warrants a warning to the UAS operator.
- Two violations during a single mission of the assumptions warrant a warning to the UAS operator and the UAS range-safety officer.
- A third violation during a single mission requires that the UAS descend and slow down under autonomous control.

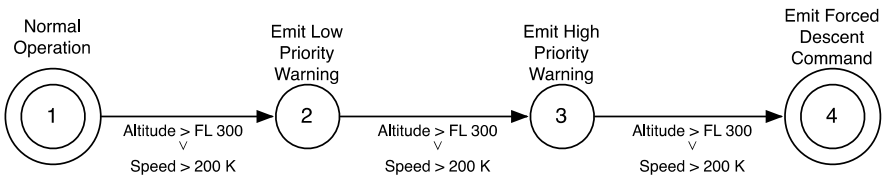

Fig. 4. A monitoring system state recognizer example

The finite-state machine implements these policies. Changes of state of the machine occur as a result of tokens that are generated by the state predicates. The state predicates are defined in terms of the altitude and speed data supplied to the monitoring system by sensors on the aircraft. The triggers for the state predicates are initiated by events generated from the Monitoring Data Repository as new data arrives from the UAS sensors.

3 Sensors

In all lifecycle phases, the necessary monitoring system sensors might have to measure a wide variety of signals with diverse characteristics, and respond to demands to start sensing, stop sensing or change the sensing frequency. In this section, we summarize the characteristics of sensors and examine one particularly challenging type of sensor, that which arises in software systems that cannot be modified easily.

3.1 Sensor Characteristics

The sensors used by the monitoring system in any particular circumstance have to be tailored to the specific application of interest and to the associated invariants. In general, sensors have to cover both periodic and aperiodic measurement, and have to handle a wide range of sampling frequencies, data types and data volumes. All sensors operate with the same basic interface to the associated monitoring repository.

Process invariants are tied closely to:

- **Process** - the various processes used throughout the lifecycle. Processes must be executed as expected if appropriate value is to be obtained.

- **Workflows** - the workflows used by each process. Workflows have to be executed by the expected entity (human, machine, or combination) and in the expected way.
- **Reuse** - the actions associated with use of the asset libraries, such as argument pattern, process pattern, and software libraries. Reuse of assets must be based upon appropriate selection and instantiation.
- **Updates** - updates to the asset libraries. If defects are detected in an asset after the asset has been used, one or more revisions to the subject system's artifacts might be required.

Sensors for process monitoring are merged into the tools and resources used to manage the various process elements. Process templates are defined in the *Business Process Model and Notation* 2 (BPMN2) [4]. BPMN2 is a graphical language that is easily read by humans and is executable on a wide range of common workflow engines. BPMN2 processes consist of activities that must be performed by particular roles and can involve humans or be automated. They follow partially ordered sequences and are often separated by decision points that execute flow control. Process monitoring sensors can be integrated relatively simply into BPMN2 specifications.

State monitoring requires sensors that capture data from the system artifacts and could include state information about development activities, approval activities, operational activities, maintenance activities and so on. Sensed data could be a record of human action as observed by a computer system, details of component failures or performance, calculated values within a software component, etc.

Many, perhaps most, state sensors will be implemented in software, and the sensor implementation will need access to the state so that the requisite data can be captured. As an example of state sensing, in the next section we discuss details of the monitoring system' approach to a particularly difficult type of state sensor, the state of software that cannot be easily modified.

3.2 Sensors and Unmodifiable Software Systems

In general, a sensor that samples data from a software system necessitates the introduction of additional software into the system, i.e., modifying the subject software. Such a modification usually requires access to the subject software's source code and subsequent rebuilding of the system. Modifications of this type might not be desirable, might be inconvenient, or might not be possible for several reasons including:

- **Inaccessible source code**. In some cases, the source code of an element of the system software might not be accessible. The source code for reusable libraries and software obtained from independent suppliers is frequently unavailable and so required sensors have to be installed in a different way.
- **Temporary requirement**. A circumstance might arise in which monitoring an element of the state for which no sensor exists becomes necessary. This situation might arise if, through exceptional circumstances such as an emergency, concern arises that the system might need to operate outside of the planned environment. Introducing a sensing capability without having to modify the source code and rebuild the software is highly desirable in such a case.

- **Change in data demand**. Sensors would typically be installed as part of system development, and sensing rates would be determined and set as part of the design. Unplanned changes in sensing parameters might arise if field observations indicate the need. Again, modifying the sensing parameters such as the sampling rate without modifying the source code is highly desirable.

We characterize these situations as needing to modify software that is not easily modified. Though unusual, we expect such situations to arise, and the monitoring system deals with the difficulties of this type using *dynamic binary translation*. In effect, the binary version of the software is modified dynamically during execution to effect the desired change without having to modify the source code. Assurance of desired system properties is achieved by relying on formalism to a large extent.

Dynamic binary translation operates by executing the subject software in an application *virtual machine*. The translator is an execution-time fetch-execute loop that fetches a fragment of the binary program, examines and optionally modifies the fragment, and then executes the fragment. For monitoring system sensing, this translator can modify the binary by inserting sensing instructions into a fragment as part of the fetch-execute loop. Modern dynamic binary translation systems add very little overhead to the program. The monitoring system uses a specific system called *Strata* [3]**Error! Reference source not found.**. Strata does not require adaptation for a particular application. Rather, Strata uses formal specifications of the desired changes to machine instructions to generate sensing instructions.

Software invariants in safety arguments span a wide range of application semantic levels and application timeframes. At the highest level of abstraction, invariants are based upon quantities that are closely related to real-world entities such as aircraft operating parameters related to flight dynamics. At the lowest level of abstraction, invariants are based upon machine-level detail arising from the implementation.

For practical purposes, the software-state monitoring task can be divided into three semantic levels corresponding to:

- The *model* level, i.e., the application specification level. Software at this level derives from the application of tools such as MathWorks Simulink® that synthesize the associated high-level-language software.
- The *source-code* level, i.e., the level of the data structures and algorithms within the application. Outside of synthesis, software at this level derives from custom code developed by application engineers.
- The *binary-code* level, i.e., system implementation level. Software at this level derives from libraries and possibly other reused assets.

From the perspective of sensor changes in software that is hard to modify, the model-based-development level is the most complex of these semantic levels, and we discuss the use of dynamic binary translation for that case and give an example of the process.

The sensing technique for model-based development is shown in Figure 5. High-level-language source code is generated for the subject model by synthesis. The monitoring system supports models specified in Simulink® running under Linux on Intel X86 platforms. For Simulink®, the synthesis is provided by MathWorks tools that generate C. The source code is compiled and linked to form the binary program.

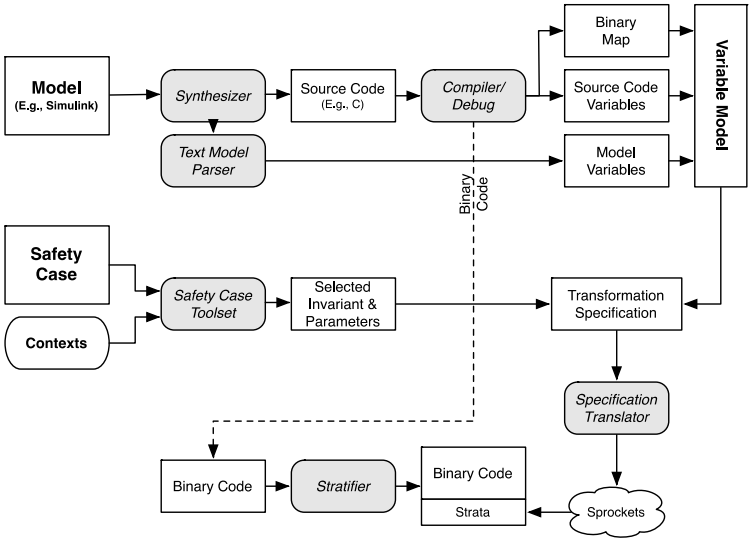

Fig. 5. Monitoring system sensing technique for model-based development.

For the monitoring system, the binary program is further processed by a utility called the *Stratafier* that installs Strata in the binary program. The semantics of the resulting program are unmodified although a modest overhead is introduced. Separately: (a) the Simulink® model is processed to extract the details of the variables used in the model, (b) the C code is processed to extract the variables used in the C program, (c) the binary program is processed to determine the locations of variables in the binary program, and (d) a *variable model* is built.

The variable model links the three sets of variables. The link between a Simulink® variable and the associated variable in C is derived from the naming convention that Simulink® uses in the generated code. The link between a variable in C and the associated memory and instructions in the binary program is derived from the symbol information placed in the binary file by the compiler.

The invariants are determined from the safety case arguments and contexts. The execution-time actions that Strata takes to effect the necessary sensing (machine instruction insertion and modification) are defined by specifications derived from the invariants and the variable model. These specifications are defined using Strata's translation specification infrastructure and are translated into Strata *sprockets*, low-level commands that control Strata during execution.

3.3 An Example of Sensing in a Simulink® Model

To illustrate the ideas outlined in the previous section, we present an example based on a hypothetical anti-lock braking system (ABS). In the example, we assume a requirement has arisen for temporary sensing of the vehicle speed (variable `Vehicle`

`Speed` in the model) after deployment and that installing the sensor using dynamic binary translation is the preferred approach.

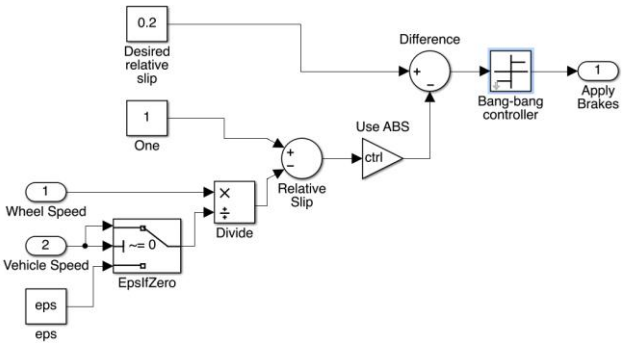

Fig. 6. Part of a hypothetical ABS defined using Simulink®.

Part of the Simulink® model of the ABS is shown in Figure 6, and part of the C code synthesized from the Simulink® model that includes an assignment to the C variable `VehicleWithABS_B.Vs` corresponding to the Simulink® variable `VehicleSpeed` is shown in Figure 7. The associated machine instructions are shown in Figure 8. Based on a symbolic specification for the required sensing, Strata inserted a branch to a predefined instruction sequence that transmitted the value of the register holding the vehicle speed to the monitoring data repository.

VehicleWithABS.c

```
136         VehicleWithABS_X.WheelSpeed1_CSTATE =
137           VehicleWithABS_P.WheelSpeed1_LowerSat;
138       }
139     }
140
141     VehicleWithABS_B.Ww = VehicleWithABS_X.WheelSpeed1_CSTATE;
142
143     /* Gain: '<Root>/Vehicle speed (angular)' */
144     VehicleWithABS_B.Vs = 1.0 / rtP_Rr * VehicleWithABS_B.Vehiclespeed_o1;
145
146     /* Integrator: '<Root>/Stopping distance' */
147     rtb_Ff = VehicleWithABS_X.Stoppingdistance_CSTATE;
148
149     /* Fcn: '<Root>/Relative Slip' */
150     VehicleWithABS_B.slp = 1.0 - VehicleWithABS_B.Ww / ((real_T)
151       (VehicleWithABS_B.Vs == 0.0) * 2.2204460492503131e-16 + VehicleWithABS_B.Vs);
152
153     /* Lookup_n-D: '<Root>/mu-slip friction curve' */
154     rtb_Ff = look1_binlxpw(VehicleWithABS_B.slp, rtP_slip, rtP_mu, 20U);
155
156     /* Gain: '<Root>/Weight' */
157     rtb_Ff *= rtP_m * rtP_g / 4.0;
158
159     /* Gain: '<Root>/-1//m' */
160     VehicleWithABS_B.um = -1.0 / rtP_m * rtb_Ff;
161
162     /* ModelReference: '<Root>/ABS Controller' */
163     ABS_Controller_oem(&(VehicleWithABS_DW.ABSController_DWORK1.rtm),
164                        &VehicleWithABS_B.Ww, &VehicleWithABS_B.Vs,
165                        &VehicleWithABS_B.ABSController,
```

Fig. 7. Part of the synthesized code for the hypothetical ABS.

Inevitably, dynamic binary translation will disturb the timing characteristics of a system and impact real-time performance. Various techniques can be used to mitigate the effects of the timing disturbance including: (a) using models to predict worst-case execution time (WCET) of monitored software given the WCET of the unmonitored software, (b) selective monitoring for much of the system's operating time could be disabled thereby eliminating the disturbance to real-time processing, and (c) static translation of the binary, where possible – static binary translation allows traditional WCET techniques to be applied.

```
402d5b:       48 8b 05 7e 82 21 00    mov    0x21827e(%rip),%rax        # 61afe0 <_DYNAMIC+0x390>
402d62:       48 8b 50 08             mov    0x8(%rax),%rdx
402d66:       48 8b 05 53 81 21 00    mov    0x218153(%rip),%rax        # 61aec0 <_DYNAMIC+0x270>
402d6d:       48 89 50 10             mov    %rdx,0x10(%rax)
402d71:       48 8b 05 40 82 21 00    mov    0x218240(%rip),%rax        # 61afb8 <_DYNAMIC+0x368>
402d78:       f2 0f 10 08             movsd  (%rax),%xmm1
402d7c:       f2 0f 10 05 f4 2f 01    movsd  0x12ff4(%rip),%xmm0        # 415d78 <gblInportContinuous+0x28>
402d83:       00
402d84:       f2 0f 5e c1             divsd  %xmm1,%xmm0
402d88:       48 8b 05 31 81 21 00    mov    0x218131(%rip),%rax        # 61aec0 <_DYNAMIC+0x270>
402d8f:       f2 0f 10 08             movsd  (%rax),%xmm1
402d93:       f2 0f 59 c1             mulsd  %xmm1,%xmm0
402d97:       48 8b 05 22 81 21 00    mov    0x218122(%rip),%rax        # 61aec0 <_DYNAMIC+0x270>
402d9e:       f2 0f 11 40 18          movsd  %xmm0,0x18(%rax)
402da3:       48 8b 05 36 82 21 00    mov    0x218236(%rip),%rax        # 61afe0 <_DYNAMIC+0x390>
402daa:       48 8b 40 10             mov    0x10(%rax),%rax
402dae:       48 89 45 f0             mov    %rax,-0x10(%rbp)
402db2:       48 8b 05 07 81 21 00    mov    0x218107(%rip),%rax        # 61aec0 <_DYNAMIC+0x270>
402db9:       f2 0f 10 48 10          movsd  0x10(%rax),%xmm1
402dbe:       48 8b 05 fb 80 21 00    mov    0x2180fb(%rip),%rax        # 61aec0 <_DYNAMIC+0x270>
402dc5:       f2 0f 10 40 18          movsd  0x18(%rax),%xmm0
```

Fig. 8. Part of the binary code for the hypothetical ABS.

4 Conclusion

Monitoring safety conditions is important, because belief in safety goals frequently depend upon them. We have presented the Safety Condition Monitoring System (SCMS), a system designed to provide comprehensive, lifecycle monitoring of safety conditions. A key feature of the monitoring system is its ability to sense software state using dynamic binary translation without requiring any modifications to the subject software.

References

[1] Patrick J. Graydon, John C. Knight and Elisabeth A. Strunk, Assurance Based Development of Critical Systems, 2007 International Symposium on Dependable Systems and Networks (DSN), Edinburgh, Scotland (June 2007)

[2] Patrick Graydon and John Knight, Process Synthesis in Assurance Based Development of Dependable Systems. EDCC 2009: Eighth European Dependable Computing Conference Valencia, Spain (May 2009)

[3] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Benjamin Rodes, Matthew Hall, Clark Coleman, John Knight and Jack Davidson, A Framework for Creating Binary Rewriting Tools, Tenth European Dependable Computing Conference, Newcastle upon Tyne, UK (May 2014)

[4] Object Management Group, Business Process Model and Notation (BPMN) Version 2, http://www.omg.org/spec/BPMN/2.0/PDF/