CLASS SAFETY CONDITION MONITORING SYSTEM

John Knight and Jonathan Rowanhill
Dependable Computing LLC

Dependable Computing Technical Report TR-2014-2

12/1/2014

Dependable Computing LLC
2120 North Pantops Drive, Charlottesville, VA 22911-8648
www.dependablecomputing.com

CLASS Safety Condition Monitoring System

Abstract

CLASS safety analysis depends on assumptions about the correct implementation of the development process for the subject system and the associated safety case. In the event that an assumption is false or becomes false during system development, the rationale for belief in a claim within a safety argument might be invalidated and the safety of the associated system compromised. Assumptions are also made within safety arguments, and these assumptions made during development of the safety argument must remain true throughout the lifecycle, especially during operation. Assurance that both types of assumptions actually hold when they are supposed to is not guaranteed, and so monitoring of assumptions is essential. Monitoring the development of a system and monitoring the system itself during operation to check for violations of assumptions made in the system's safety argument is an important aspect of the system safety lifecycle. The CLASS Safety Condition Monitoring System provides comprehensive monitoring of the assumptions made in both the system development process and the system's safety argument together with an alert infrastructure that allows flexible responses to violations of assumptions.

1 Introduction

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, where residual risk is defined as:

> **Residual risk**: the expected loss per unit time or per demand that results from use of the system.*

In deployment decisions for safety-critical systems, decision making is based, in part, on assessment of whether the estimated residual risk value and variance exceed what is determined to be an acceptable level.

Inevitably, all safety analyses depend upon assumptions or expectations about the system being analyzed and the way that the system will be used. Such assumptions include:

- Details of the operational context within which the system will operate.
- Failure rates and failure semantics of physical devices.
- Failure rates and failure semantics of software entities.
- Performance of systems in terms of physical capabilities such as strengths of elements, wear resistance, and corrosion resistance.
- Performance of systems 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 such as these are often stated explicitly in safety arguments and become part of the rationale for belief. In the event that an assumption is false or becomes false once the system is deployed, the rationale for belief in a claim within a safety argument might be invalidated and the safety of the associated system compromised.

In CLASS, safety analysis also depends on assumptions about the correct implementation of the development process for the subject system and the associated safety case. Just as with operation, in the event that an assumption is false or becomes false during system development, the rationale for belief in a claim within a safety argument might be invalidated and the safety of the associated system compromised.

In summary, the effectiveness of CLASS relies upon two conditions:

- The detailed activities that constitute an instanceCLASS are conducted as defined.
- The assumptions used in the CLASS analysis of the subject system are true throughout the lifecycle of the subject system.

CLASS cannot dictate that either of these conditions hold, and so a fundamental aspect of CLASS is to *monitor* both conditions. More specifically, monitoring in CLASS plays two roles:

- *Development Monitoring.* Development monitoring takes place throughout the lifecycle and monitors adherence to the lifecycle processes and procedures.
- *Operational Monitoring.* Operational monitoring takes place during operation of the subject system and monitors adherence to the operational assumptions made about the system in the lifecycle analysis.

The two conditions imply predicates on the state of a system, both during development and during operation. These predicates must remain true, and the role of monitoring is to check the value of the predicates. For purposes of discussion, we refer to these predicates as *state invariants* or simply as *invariants*.

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

The SCMS does *not* monitor hazardous states or violations of safety requirements. The requirements for the monitoring system derive from the creation of 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 SCMS 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 a system utilizing the SCMS during operation is shown in Figure 1.

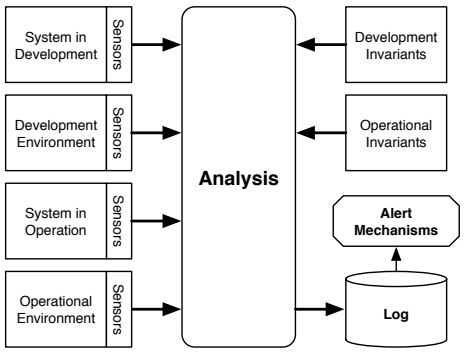

Figure 1: CLASS Safety Condition Monitoring System

2 SCMS Requirements

The high-level requirements for CLASS monitoring are relatively easy to state: the monitoring system shall detect any violations of invariants that derive: (a) from the instanceCLASS by which the system was created and certified, or (b) from the invariants derived from assumptions made in the safety case. In practice, however, the detailed requirements are quite complex and difficult to state. The major questions that need to be addressed in defining the requirements are:

- What process requirements are defined by the associated instanceCLASS?
- What invariants are derived from the assumptions made in the system safety argument?
- What data needs to be acquired during development and certification in order to determine violations of the defined process?
- What data needs to be acquired during system operation in order to determine violations of the invariants?
- How should violations of process elements or operational invariants be communicated to the system's stakeholders?

We address each of these questions in turn.

2.1 Determination of instanceCLASS Invariants

Invariants in instanceCLASS are derived from the CLASS instantiation process. Instantiation involves:

- Determination of the guidance to be applied. Guidance is determined by the project type.
- Specific assets acquired from the CLASS Resource Repository.
- Specific assets to be maintained in the Safety Information Repository.
- Specific process elements to be used by the instantiation.

In CLASS Prototype 3, the determination and definition of these invariants is carried out manually. The SCT handles all aspects of process and workflow management, and so the SCT can be used to help define the details of what needs to be monitored.

2.2 Determination of Safety Case Invariants

Invariants in safety arguments derive from four sources:

- Assumptions stated explicitly as nodes in the GSN argument.
- Justifications stated explicitly as nodes in the GSN argument.
- Parameters associated with contexts.
- Contract guarantees associated with module interfaces.

For any specific system, the condition monitoring requirements begin with an elaboration of the invariants derived from these four sources.

In CLASS Prototype 3, the collection of these invariants is carried out manually. The Safety Case Toolkit (SCT) maintains safety arguments with sufficient detail that scanning an argument to support determinations of assumptions will be simple. This enhancement to the SCT is planned for the future.

2.3 Determination of Sensing Requirements

Once items to be monitored have been determined, the necessary associated data, *i.e.*, the data necessary to check the item, needs to be determined and a sensing structure determined to capture the data during development or operation.

2.4 Determination Alert Actions

Violation of an invariant during system development does not necessarily indicate that a mistake has been made in development nor that a defect has entered either the system or the safety case. Similarly, violation of an invariant during system operation does not necessarily indicate that the system has entered a hazardous state. Many invariants are present to ensure system integrity over time, for example, and their violation should be investigated but investigation is not necessarily urgent. Thus, the alert associated with violation of some invariants needs nothing more than recording in a suitable log.

Violation of some invariants do indicate entry to a state that requires attention. Exactly how such state transitions should be indicated and to whom, however, depends on the invariant and the state that the system has entered.

Invariants are sometimes relatively benign yet their violation could lead to states in which other invariants become more significant. Thus an important requirement for alert actions is to be able to affect the operational parameters of the monitoring system.

2.5 Notification Mechanisms

A Notification Mechanism is a device that can effect some sort of notification outside the SCMS if required. For *development* monitoring, typical Notification Mechanisms might include: (a) invokation of software within the development environment to take some form of remedial action including changes to displays, and (b) notification of engineers via on-screen messages, social media or e-mail. For *operational* monitoring, typical Notification Mechanisms might include: (a) audio warnings, (b) visual warnings, and (c) changes to displays.

3 SCMS Design

In practice, development environments and safety-critical systems are often *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 monitor a number of different system elements and for the associated monitoring to integrate the results of analyses in order to ascertain the state of complex conditions.

The SCMS accommodates this system architecture by operating itself as a distributed system with the various elements of the SCMS communicating in a manner determined by the structure and details of the invariants being monitored.

As an example, the distributed structure of an operational SCMS and how the SCMS might be integrated into a simple avionics architecture is shown in Figure 2.

The design of the SCMS is shown Figure 3. The design assumes that all requisite sensors have been deployed through all the relevant environments with which the system has to operate. For a development SCMS, these environments would include, minimally, the system's CRR, the subject system's SIR, the process and workflow definitions, and the various assets in use for development. For an operational SCMS, these environments would include, minimally, the system's operational and maintenance environments.

3.1 Event Bus

Central to the design of the SCMS is the *Event Bus* (see right-hand side of Figure 3). The Event Bus accepts event notifications from any part of the SCMS and delivers those notifications to any destination within the SCMS. The

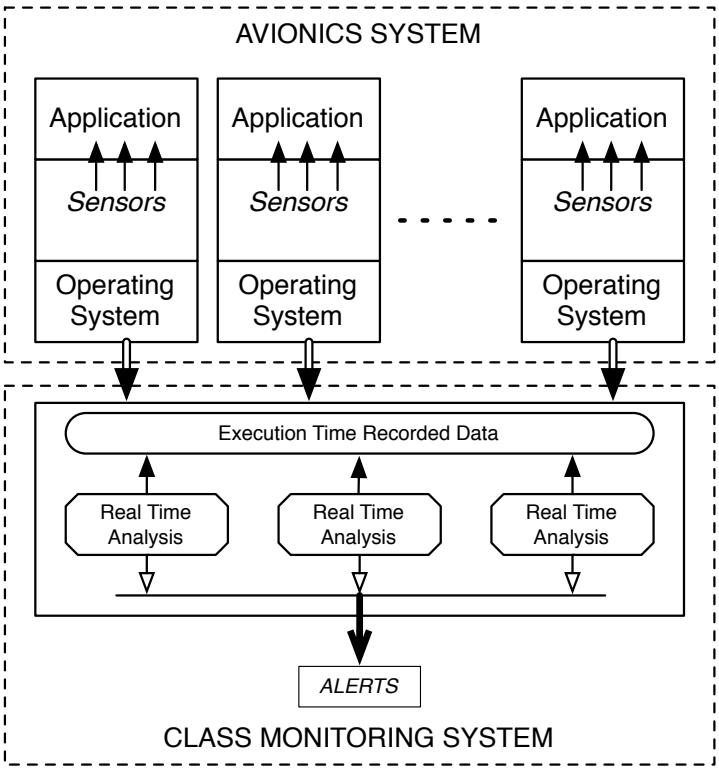

Figure 2: CLASS Safety Condition Monitoring System

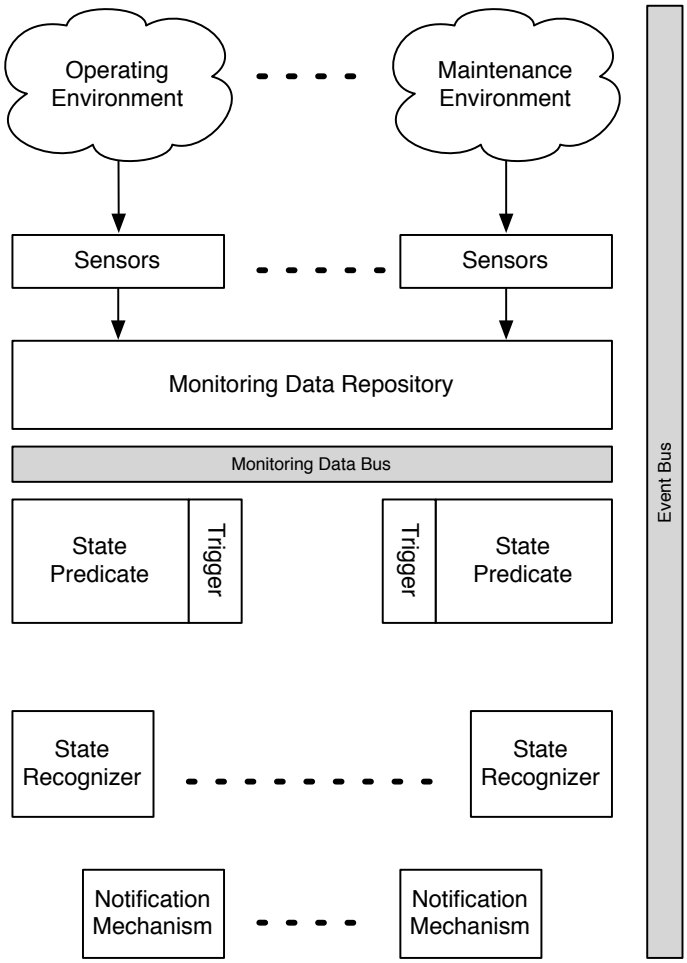

Figure 3: SCMS Design

purpose of the Event Bus is to provide a comprehensive notification mechanism within the SCMS. Thus changes within one part of the SCMS that require action elsewhere result in event generation and transmission.

3.2 Data Repository

Sensor data will arrive at intervals determined by the SCMS but no polling of sensors is assumed. Separate scheduling and timing control is assumed 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 the data repository to signal the availability of sensor data.

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

3.3 State Predicates

The State Predicates codify the safety conditions, and the evaluation of a state predicate to *true* indicates a safety condition has arisen. 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 to be equipped with ADS-B In and ADS-B Out 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:

$$\text{altitude} <= 300$$

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

State Predicates have to be executed when suitable data is available for their evaluation and when evaluation is meaningful. Each state predicate is controlled by a guard referred to as a *Trigger* (see Figure 3). A Trigger is also a predicate that causes the associated State Predicate to be evaluated. A Trigger encodes details such as the availability of the data and the requisite time for the evaluation of the associated State Predicate.

The evaluation of a State Predicate to *true* causes the predicate to generate a *token* (implemented as an event) that is transmitted to the State Recognizers.

3.4 State Recognizers

The SCMS State Recognizers encode the alert semantics that the system stakeholders require for violations of invariants from either development process elements or assumptions in the safety argument. 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 might be 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 SCMS is shown Figure 4. The example is for the operational 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 of the assumptions warrants a warning to the UAS operator and the UAS range safety officer, and
- a third violation requires that the UAS descent under autonomous control.

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 SCMS by sensors on the aircraft, possibly ADS-B. The Triggers for the State Predicates are initiated by events generated from the Monitoring Data Repository as new data arrives from the UAS sensors.

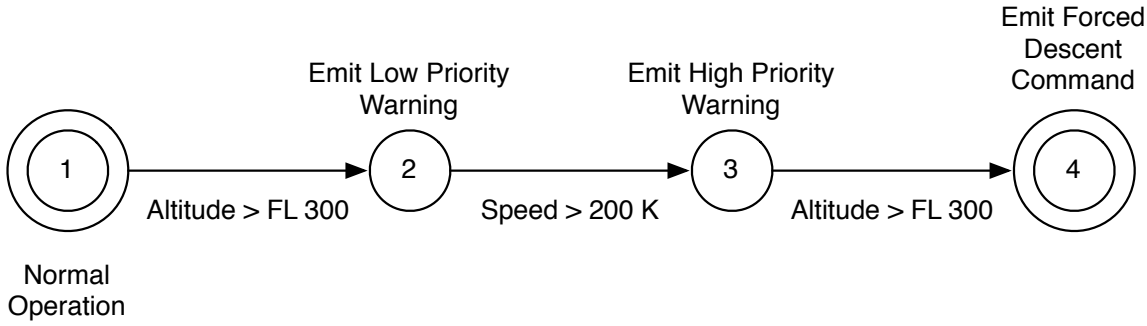

Figure 4: SCMS State Recognizer Example

4 SCMS Sensors

4.1 Development Monitoring Sensors

Invariants in development are tied closely to:

- The various processes used throughout the lifecycle.
- The workflows used throughout each process.
- The actions associated with use of the CRR.
- The actions associated with the creation and use of the SIR.
- Updates to the CRR both from within and without the resources associated with the subject system.

Sensors for monitoring development are merged into the tools and resources used during development.

4.2 Operational Monitoring Sensors

Invariants in safety arguments span a wide range of application semantic levels and application timeframes. In terms of application semantic levels, at the highest level of abstraction invariants are based upon quantities that are closely related to real-world entities. Examples include aircraft operating parameters related to flight dynamics. Such quantities are most likely involved in invariants that are relied upon in model-based computing statements.

In terms of application timeframes, at the shortest

The SCMS design breaks the monitoring task during operation into three semantic levels corresponding to:

- The model-based development level (*e.g.*, the SIMULINK® level).
- The source-code level (*i.e.*, the level of the data structures and algorithms within the application).
- The binary-code level (*i.e.*, system implementation level).

At the two higher semantic levels, monitoring requires domain expertise and explicit provision of probes into the subject software, either a model or a source program. Tools exist (separate from CLASS) for languages such as SIMULINK® to allow the specification of monitoring. At the source-code level, editors, macro processors, and specialized tools can be used to develop and install necessary sensors.

4.3 Monitoring by Dynamic Binary Translation

The third semantic level of monitoring is far less familiar as is the planned level of implementing of this level. The semantics of the third level correspond roughly to the interaction between the application system and the platform upon which it runs. Examples of system characteristics on interest at this level are:

- Timing margins, jitter rates, and frame overruns in real-time systems.
- Data losses, errors, delays and bandwidth limitations in communications systems.

The way in which the CLASS Prototype 2 design includes a monitoring capability for this semantic level is via probes inserted into the software using *dynamic binary translation*. The major benefits of this technique are:

- Probes can be inserted to measure virtually any parameter of interest.
- No modification of the application software is required. Any software can be monitored thereby ensuring that the developers of the software do not need to make provision for monitoring nor modify the software once completed if a safety case is to be employed in any given application.
- Essentially all of the information that might need to be sensed is available at the binary level although the information might be difficult to locate.

Further detail of the CLASS monitoring structure design based on dynamic binary translation is shown in In this design, each monitored application executes as a virtual process with Strata providing the virtualization. Strata provides the sensing mechanism by translating binary code sequences to include the necessary sensing instructions. These instructions capture the data and transmit the data to a CLASS data recording process that further transmits the data to a set of CLASS real-time analysis processes. Real-time analyses that detect a violation generate a suitable alert.

4.4 Monitoring for Real-time Systems

Application monitoring at any of the three semantic levels presents a challenge in many systems, especially avionics systems, because such systems frequently operate in hard real time. Any changes made to effect monitoring will inevitably disturb the basic timing characteristics of the system. Dynamic binary translation is the technique of choice because it operates with unmodified software. However, little is known about how to ensure that real-time deadlines will be met with a system operating under dynamic binary translation.

CLASS Prototype 2 does not address the real-time issue with monitoring other than to enumerate the possible approaches to supporting real-time operation. The possible approaches that future CLASS prototypes might support include:

- **Modeling**. Models that can predict worst-case execution time (WCET) of monitored software given the WCET of the software operating unmonitored might be developed.
- **Estimation**. Estimation of WCET for monitored software based on traditional WCET assessment techniques might be possible.
- **Selective monitoring**. Monitoring for much of the system's operating time could be disabled thereby eliminating the disturbance to real-time processing. Most safety-related assumptions in software could probably be monitored at the level of seconds (or more) rather than milliseconds (or less). Thus, by operating selectively, monitoring might impose a negligible burden.
- **Static binary translation**. Dynamic binary translation will be used initially. Once the approaches are fully developed, effecting the necessary monitoring might be possible using either static binary rewriting or (less likely) adding wrapping capabilities to functions that have to be monitored. The advantage of static binary translation is that the enhanced binary is built before execution and traditional WCET techniques can be applied.

