A Resilient Real-Time Traffic Control System

Ahmed Serageldin
Email: Ahmed.Serageldin.1977@ieee.org

Axel Krings
Email: krings@uidaho.edu

***Abstract*—This paper describes a resilient control system operating in a critical infrastructure. The system is a real-time weather responsive system that accesses weather information that provides near-real-time atmospheric and pavement observation data that is used to adapt traffic signal timing to increase safety. Since the system controls part of a safety critical application survivability and resilience considerations must be an integral part of the system architecture. In order to provide adaptation to system behavior as the result of faults or malicious acts an architecture is presented that monitors itself and adapts its behavior in real-time. The main theoretical contributions are the combination and extension of approaches introduced in previous work. The theory of certifying executions is extended by three concepts: the detection of dependency violations, exceptions triggers, and sensor analysis are considered; a dual-bound threshold approach for detecting off-nominal executions is introduced; profiling is augmented with the concept of behavior sets. Extensive evidence of the effectiveness of the solutions based on a one-year observation of the system in action is presented.**

I. Introduction and Background

Advanced traffic signal systems that adapt to changing traffic conditions in real time are at the core of most Intelligent Transportation Systems (ITS) traffic management applications. In this paper, we describe a resilient real-time weather-responsive traffic signal control system that intents to improve the efficiency and safety of traffic signal operations during inclement weather conditions. The system receives and analyzes road weather information from an integrated surface transportation weather observation data management system and adapts signal timing in response to changes in road surface conditions and/or visibility level.

Real-time control systems, especially those governing critical infrastructures, e.g., transportation, need to be reliable and secure under normal operating conditions and survivable under abnormal conditions. They should be designed and operated so that essential services will function even in the presence of component failure or external or internal manipulations of the system or the data it relies on. The control system described here has to address these fault tolerance and resilience requirements during the execution of two tasks. The first consists of the system accessing near real-time atmospheric, weather, visibility, and road surface condition information from the Federal Highway Administration (FHWA) Clarus system [1]. The second task adapts signal timing in response to inclement weather based on this information. Since the system accesses data on domains outside its secured local communication networks, the data exchange architecture needs to be designed in a way that is resilient against cyber attacks and intrusions.

The control system was designed with resilience consideration in mind, utilizing two essential software design approaches: *Design for Survivability* [2] and a *Measurement-Based Methodology* [3], [4]. The first approach is derived from the concept of *Design for Testability*, which addressed the impossibility of completely testing VLSI circuits by designing them with testability in mind [2]. Now, survivability considerations are similarly integrated into the design, rather than in an add-on fashion. The second approach, i.e., Measurement-Based Methodology, was proposed for critical applications that rely on measurements of operational systems and on dependability models to provide quantitative survivability with certain user-defined confidence levels. The software design incorporates self-monitoring techniques for fault detection and recovery to maximize the resilience of the system.

A. Contributions

The contributions of this paper are of theoretical and practical nature. The main theoretical contribution is the combination of the approaches introduced in [3], [5], [6] into one comprehensive architecture with survivability and resilience characteristics. Furthermore, the subsystem that monitors the application program is extended to three monitoring approaches: 1) detection of dependency violations, 2) identification of anomalies through exception triggers and data sensor analysis, and 3) detection of off-nominal, non-certified executions. The theory of the latter is extended to allow for certification of executions based on *Behavior Sets*. Furthermore, a dual-bound threshold approach for detecting off-nominal executions is introduced. The practical significance is in the description of the actual system with extensive evidence to the resilience of the architecture based on observation of the system in action and data collected by the system during the year 2012.

B. Related Work

Run-time monitoring refers to the process of monitoring the system's behaviour in real-time. The goal is to determine whether the system performs its tasks to specifications or if there are anomalies in the execution patterns. The latter could indicate that the system is compromised. Has the software experienced a fault, has the system been attacked, or is it executing correctly in a fashion that we just have not observed before? These questions have plagued the dependability and security communities for decades. Fault detection and treatment have been researched by the dependability and software engineering communities. Attack recognition, i.e., intrusion detection, is a very complex problem and detecting patterns or anomalies has been a constant hot topic in the intrusion detection community, e.g., signature-based approaches or anomaly detection. Especially in anomaly detection the critical issue is where one should set the threshold for deciding what is normal and what is not [7]. It should be noted that the methods of intrusion detection, i.e., the claim made about the mechanisms used for detection, has not been without controversy [8].

Detection of off-nominal executions implies that one knows what a nominal execution looks like. We do not attempt to mimic anomaly detection, but utilize the detection of previously observed executions patterns, e.g., profiles, versus those

we have just not seen before. Instead of focusing on "what is abnormal", we focus on "what is normal". Thus everything outside of previously identified, i.e., nominal, behavior is simply assumed off-nominal.

The research presented here is based on early work using frequency spectra of observed system executions [9]. They presented a real time approach to detect system behavior deviating from normal activities, with focus on attacks on the system software. Similar approaches were taken in [10], where signatures related to observed frequency behaviors were constructed for attack signature detection. Both approaches used profiling based on injected execution handles, an approach that is also used in Unix systems, e.g., when compiling C programs with the -g option. The concepts were later combined into a measurement based methodology for embedded software systems [4], which was the starting point for this research.

II. Real-time Control Application

The traffic control infrastructure is augmented with capabilities driven by performance and safety improvement goals.

A. Control System Components and Operation

The real-time weather responsive system is shown in Figure 1. The non-shaded components are the existing ITS system whereas the shaded components are additions that implement real-time weather response and system resilience. The traffic lights in an intersection are controlled by a traffic controller hosted in a cabinet located in the intersection. The traffic controller is connected to a switch, or hub, to the ITS control network, which is either physically totally separated or connected to the Internet via a Firewall. It should be noted that the separation of the ITS control network and the Internet is critical and any access through the firewall has to be extremely limited and under strict compliance with security policies.

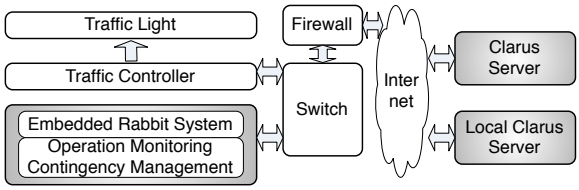

Fig. 1. Overview of the real-time weather response system

Weather data is collected by the Clarus system from a network of Environmental Sensor Stations (ESS) of participating states. The network of ESS can be viewed at [1] by the general public. This ESS data is accessible via the Internet from the Clarus server, after it undergoes quality and consistency checks based on Clarus quality checking algorithms [11]. Due to this quality check, survivability considerations do not include verification of the original Clarus data. An embedded Rabbit[1]-based system located in the traffic signal system in the intersection retrieves data from the Clarus server or a local mirror site, analyses the relevant data, and computes changes to the signal timing. Upon approval, signal timing changes are made in the Traffic Controllers by the Rabbit system. Signal timing plan adaptations include changes such as modified all-red or yellow clearance intervals or traffic signal efficiency parameters such as minimum green, maximum green, passage time as well as different coordination parameters. Suggested changes depend on multiple factors such as approach speed, pavement surface conditions, visibility, and the mode of signal operations.

[1]Rabbit is registered trademark of Digi International Inc., www.rabbit.com

B. The Clarus System Weather Data Support

The data that is needed to implement real-time weather responsiveness comes from the ESS sensors. The Clarus System shown in Figure 1 maintains the location of all ESS. The ESS most suitable for the specific traffic signal system, e.g., the one closest to the intersection, needs to be identified and a subscription for that ESS is generated. The subscription, which may include data from a single or multiple specified ESSs, is made available via the Clarus System's subscription web site in the format of a comma separated value (CSV) file. It should be noted that the data is not queried from a data base server, but simply accessed directly over the web and is, unless password protected, publicly readable. Specifically, a list of observations, i.e., the actual CSV files, is made available in regular intervals typically ranging from 5 to 15 minutes. The specific observations in the list depend on the capabilities of the ESS associated with the subscription. Within a subscription the observation files follow the file naming convention $date\_time.csv$. An observation file contains data for specific *Observation Type IDs* (ObsTypeID). The first line is a header line describing the values present in each line of data. A relevant subset of these values is used later by the system to calculate changes to be made to the traffic controller. Since a subscription can be specified to contain data from multiple ESS's sensors, e.g., including neighboring ESS, the control algorithms of the weather responsive system can take advantage of data fusion, thus having a "larger view".

C. Software System Architecture

An overview of the software system that controls the weather responsive system is shown in Figure 2, where shaded areas refer to external hardware interfaces. The Rabbit system,

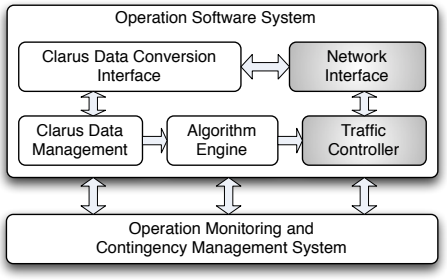

Fig. 2. Overview of the software system architecture

which we will refer to simply as "Rabbit", executes the application control software, which consists of the Operational Software System and an Operation Monitoring and Contingency Management System. The operation software system connects to either a *Local Clarus Server* (LCS), which is simply a local mirror supplying the Clarus subscription data, or the Clarus System, using the *Network Interface* to the Internet. In regular intervals specified by the Clarus subscription the Clarus data is read and converted by the Rabbit, the desired sensor data is extracted, and specific algorithms compute changes to the control parameters of interest, e.g., yellow timing adjustments. The traffic controller is then updated. All this is monitored at

run-time via the instrumentation telemetry by the *Operation Monitoring and Contingency Management System*, i.e., the Rabbit monitors the execution of its software in real-time by sensor points that are injected into the software.

III. Formal Model of Systems Architecture

The system architecture is guided by the design methodology and general principles shown in [3], [4]. It starts with the view of a general system as two distinct abstract machines that define the implementation in the development of any software system [5], [6]. The first, called *Operational Machine*, is the machine that interfaces directly with the hardware interface. The second abstract machine, called *Functional Machine*, is a set of functionalities that describe exactly how each system operation is implemented. As the system operates, operations cause functionalities, implemented by modules, to be invoked, or functionalities cause operations to be performed. During system operation, i.e., while the application is running, the operational and functional machines can be monitored in realtime, assuming appropriate instrumentation is in place to allow this. In our case, the execution of the application running on the Rabbit is monitored in real-time by three different monitoring mechanisms, shown in Figure 3. The

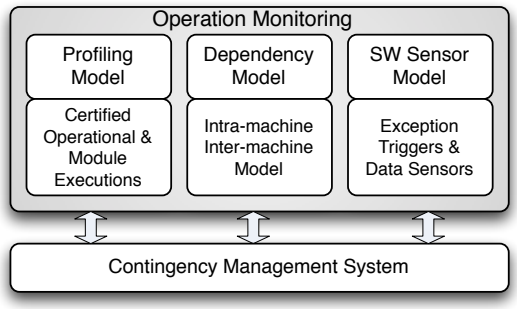

Fig. 3. Using Profiling, Dependencies, and Data Sensor Monitoring

first mechanism, described by a Profiling Model, is based on analysis of realtime execution profiles. It will be used to describe measurement of typical behavior as the basis for what to expect, with a certain probability of error, in the future. The second mechanism, covered by a Dependency Model, is monitoring for violations of state dependencies between, and within the machines. Any violation indicates an abnormal execution. The third mechanism, referred to as the Software Sensor Model, is based on the analysis of data supplied by specific data sensors within the software. These software sensors supply information that can be used for analysis or direct actions. All three mechanisms allow for the detection of off-nominal, unexpected, or invalid executions, which in turn are used by the Contingency Management System. We now describe each of the three models in detail.

A. Profiling-based Model

If one counts the invocations of operations, functionalities and modules over a specific period of time one can derive the respective *operational, functional* and *module profile*. These profiles will be used later in the analysis that may expose off-nominal executions. To stay compatible with the notation used in [5], [4] we will use letters u, q and p for operational, functional and module profiles respectively. The notation is introduced using module profiling as an example. Let p_i denote the probability that the system is executing module m_i. Then $\mathbf{p} = (p_1, p_2, ..., p_{|M|})$ is the module profile of the system, i.e., it is the probability vector of the modules in M.

B. Non-synchronized Profiling

During execution of the system we are interested in observing the module profile over n epochs. Here we assume that n is not synchronized to a particular higher level machine, e.g., the operational machine's epoch.

This observed profile is denoted by $\hat{\mathbf{p}} = (\hat{p}_1, \hat{p}_2, ..., \hat{p}_{|M|})$, where $\hat{p}_i = c_i/n$ is the fraction of system activity due to invocations of module m_i and c_i is the count of invocations of m_i. As the software executes, invocations of modules are continuously monitored and module profiles are generated and analyzed. We want to keep track of these profiles. Let $\hat{\mathbf{p}}^k$ denote the k^{th} module profile. Thus $\hat{\mathbf{p}}^k$ is the k^{th} observed module profile, observed over n epochs, which was preceded by $\hat{\mathbf{p}}^{k-1}$, observed over the previous n epochs, and so forth.

To get a feel for the expected evolving profile of the system we want to establish the module profile equivalent to an "h-day moving average" in financial stock movements, and derive a centroid that will serve as reference for observed profiles. For that, just as in [4], we consider h sequences of n epochs each and define a centroid $\overline{\mathbf{p}} = (\overline{p}_1, \overline{p}_2, ..., \overline{p}_{|M|})$, where

$$\overline{p}_i = \frac{1}{h} \sum_{j=1}^{h} \hat{p}_i^j \tag{1}$$

Thus $\overline{\mathbf{p}}$ is a $|M|$-dimensional vector, and using the above financial metaphor, each element represents the "h-day moving average" of a specific stock (module), where a day is measured as n epochs. Furthermore, just as in the stock market, we don't know what the future brings but find it useful to track the past in order to establish "nominal", i.e., expected, behavior.

C. Synchronized Profiling

In the previous discussion the profiles reflect the behavior of the system. However, it is a single behavior. If there are multiple behaviors that a machine may exhibit, then one has to consider sets of behaviors, which we refer to as *Behavior Sets*. Let's consider the case where modules may exhibit different behavior during an operational epoch. Therefore, assume we synchronize module epochs to the operational machine, specifically an operational epoch. Thus we make the assumption that n is the number of module epochs expiring during one operational epoch. In our application n is the number of module epochs during the 15 minute operation epoch at which the Clarus data is fetched. We now adapt the notation of the non-synchronized case and will switch from lower to upper case letters when considering behavior sets.

Now the observed profile is $\hat{\mathbf{P}} = (\hat{\mathbf{P}}_\mathbf{1}, \hat{\mathbf{P}}_\mathbf{2}, ..., \hat{\mathbf{P}}_{|\mathbf{M}|})$, where $\hat{\mathbf{P}}_\mathbf{i}$ is the behavior set of module m_i, i.e., it is a set of different profiles $\hat{p}_i = c_i/n$, which again represents the fraction of system activity due to invocations of module m_i and c_i is the count of invocations of m_i during the operational epoch of length n. Analogous to the non-synchronized case, let $\hat{\mathbf{P}}_i^k$ denote the behavior set of the k^{th} module profile. Thus $\hat{\mathbf{P}}^k$ is the k^{th} set of observed module profiles, observed over n epochs, which was preceded by $\hat{\mathbf{P}}^{k-1}$, observed over the previous n epochs, and so forth.

Considering h sequences of n epochs each, we define a centroid of sets $\overline{\mathbf{P}} = (\overline{P}_1, \overline{P}_2, ..., \overline{P}_{|M|})$, where

$$\overline{P}_r = \overline{P}_r \cup p_i, \quad 1 \leq r \leq |M| \quad p_i = \frac{1}{h} \sum_{j=1}^{h} \hat{p}_i^j \tag{2}$$

for each behavior i. Thus $\overline{\mathbf{P}}$ is a $|M|$-dimensional structure of sets, and again using the above financial metaphor, each element represents the "h-day moving average" of a specific *set of stocks* (module), where a day is measured as n epochs, and again we want to track the past in order to establish "nominal", i.e., expected, behavior from a *set of behaviors*.

It should be noted that if each behavior set consists of only one element, then essentially $\overline{\mathbf{P}}$ is the same as $\overline{\mathbf{p}}$.

D. Dependency-based Model

The above discussion about profiles does not capture any dependencies *between* operations, functionalities, and modules, nor does it capture dependencies *among* them. We will refer to the first case as inter-dependencies and the latter as intra-dependencies.

E. Inter-dependencies

The relationship between operations, functions, and modules is defined by a graph $\mathcal{G}^{OFM}$, where the superscript simply indicates that the graph maps from O to F to M. The term *inter-dependencies* stems from the fact that $\mathcal{G}^{OF}$ and $\mathcal{G}^{FM}$ are bipartite graphs and $\mathcal{G}^{OFM}$ is a tripartite graph. An example is depicted in Figure 4, which shows three operations o_1, o_2 and o_3. The operations utilize specific functionalities, e.g., o_1 uses functionalities f_1 and f_2. Incidentally, f_2 is also used by o_3. The functionalities are implemented by modules, e.g., f_3 is implemented by module m_4, whereas f_4 is realized by m_4, m_5, and m_6. Checking inter-dependencies allows to identify

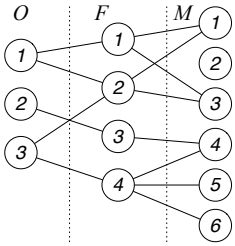

Fig. 4. Mappings in $(O \times F \times M)$

any violation of mappings. For example if during the service of functionality f_1 module m_2 would be invoked, then at the functionality level one can detect a violation, since checking the graph one knows that m_2 is not used by f_1. Similarly, at the module level the violation would be detected as m_2 finds out that it should not be called as part of f_1 services.

Violations of inter-dependencies may be the result of scenarios where the mapping from specification to code is different than the reverse mapping, i.e., from code back to specification. In the latter case the code does more than it is supposed to do.

Sometimes there is a *one-to-one* and *onto* mapping from operations to functionalities, which is the case in our application. Then the mapping of $(O \times F \times M)$ can be reduced to $(O \times M)$, which is defined by $\mathcal{G}^{OM}$. We will refer to this mapping and its implied simplification as *Mapping Simplification Assumption* throughout the paper.

F. Intra-dependencies

It is not only of interest to know which functionality is used by an operation or which modules are used by a functionality, but also to know dependencies within operations, functionalities, or modules. Those intra-dependencies can be defined by precedence graphs and are shown within the shaded areas of Figure 5. Specifically, dependencies between operations are defined by graph $\mathcal{G}^O = (O, \prec^O)$, where $\prec^O$ defines a precedence relation on the operations in O, i.e., if o_j depends on o_i then (o_i, o_j) is in $\prec^O$. Any violation of the precedence indicates a problem in the control flow of operations. We define similar graphs for functionalities and modules. Thus $\mathcal{G}^F = (F, \prec^F)$ and $\mathcal{G}^M = (M, \prec^M)$ are the graphs defining calling relationships between functionalities and modules respectively. It should be noted that $\mathcal{G}^M$ is the static call graph of modules in M created by the compiler. The operational, functional, and module dependency graphs are used to detect invalid or previously unobserved transitions. In Figure 5 the intra-dependencies are shown by solid and

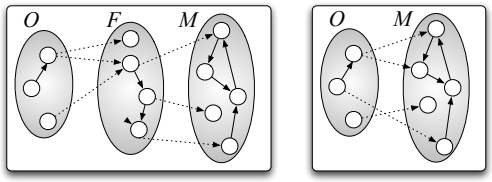

Fig. 5. Dependencies within operations, functionalities, and modules. Complete model (left), Simplified model with one-to-one and onto mapping in $(O \times F)$ (right)

inter-dependencies by dotted arrows. Figure 5 shows the complete dependencies (left), and the simplified model under the Mapping Simplification Assumption (right).

G. Sensor-based Model

Not every behavior can be extracted from profiles or dependencies. Sometimes specific data sensors are needed to observe specific data values or to trigger exceptions, e.g., as the result of abnormal, missing, or unknown data items.

1) Exception Triggers: An *exception trigger array* has been implemented to identify and profile exceptions. An example of such trigger is the detection that a file that is supposed to be accessed does not exist, that specific external sensor data is no longer available, or any other observation that will cause the program to adapt. In general, any error condition can be viewed as a exception trigger.

In our application the driving motivator for exception triggers lies in the uncertainty of the availability of data provided by environmental sensor stations (ESS). This can be due to ESS sensor failure, or simply a change in ESS configuration or hardware. Recall that there are large numbers of diverse data available from ESS and which specific data is available depends foremost on the capability of the ESS.

2) Data Sensors: Data sensors serve primarily for observation and analysis of specific numeric values. An example in our application is the value for the adjustment of the yellow period as computed by the algorithm engine shown in Figure 2. The correctness of data that is used in the computation is not questioned as Clarus provides quality and consistency checks based on its quality checking algorithm. However, the computed adjustment values provided by the data sensors can be analyzed to determine if they are feasible or make sense. Due to the NTCIP compliance of the traffic controller no safety violations can be forced, e.g., by selecting dangerously small yellow periods. But it could be possible that for some benign or malicious reason the values are constantly too large, thus constituting a denial of service attack. Thus, no matter what the reason for the extreme durations may be, the data sensor makes analysis and thus contingency management possible.

IV. Run-time Monitoring

Run-time monitoring refers to the process of monitoring the system's behavior in real-time. The goal is to determine whether the system performs its tasks to specifications or if there are anomalies in the execution patterns. The latter could indicate that the system is compromised. Has the software experienced a fault, has the system been attacked, or is it executing correctly in a fashion that just have not observed before? These questions have plagued the dependability and security communities for decades. Fault detection and treatment have been researched by the dependability and software engineering communities. Attack recognition, i.e., intrusion detection, is a complex problem and detecting patterns or anomalies has been a constant hot topic in the intrusion detection community, e.g., signature-based approaches or anomaly detection. Especially in anomaly detection the critical issue is where one should set the threshold for deciding what is normal and what is not.

The run-time monitoring employs three monitoring approaches: 1) validation of dependencies, 2) detection of anomalies through data sensor analysis, and 3) detection of off-nominal executions

Validation of dependencies is quite simple. Given the inter- and intra-dependencies discussed in subsection III-E and III-F respectively, the system can detect any violation of mappings from operations to functionalities to modules in $\mathcal{G}^{OFM}$, the simplified $\mathcal{G}^{OM}$, and any violations of precedence in each $\mathcal{G}^{O}$, $\mathcal{G}^{F}$ and $\mathcal{G}^{M}$. For example, any call graph precedence violation, i.e., a module call sequence that is not in the precedence relation of $\mathcal{G}^{M}$, indicates a call sequence that is not intended. The reason for such call sequence cannot be extracted from the simple detection, however, possible reasons could be incorrect function pointers or perhaps a code injection attack.

Detection of anomalies in values returned by code-embedded data sensors is highly dependent on the sensor type. For example, in our application one data sensor is the actual adjustment to the yellow period of the traffic light. If the rate of change of the period is not in character with the environment parameters, e.g., the surface temperature, then perhaps the value is not correct. A simple range check of parameters may detect values that are outside of the expected range.

Detection of trigger events, coming from the second type of sensors, is more straightforward. The trigger events are simple signals that indicate certain events. They can be used to initiate specific actions, or can simply serve as a tracking mechanism, e.g., to keep track of how many times certain events have occurred over a specific time interval.

Detection of off-nominal executions implies that observed profiles are checked to establish if they meet an expected certified behavior, as will be described next.

A. Certified Executions

Detection of off-nominal executions implies that one knows what a nominal execution looks like. We do not attempt to mimic anomaly detection, but utilize the detection of previously observed executions patterns, e.g., profiles, versus those we have just not seen before. Instead of focusing on "what is abnormal", we focus on "what is normal". Thus everything outside of previously identified nominal behavior is simply assumed off-nominal. Nominal behavior can be refined to a costate level [5], [6]. A costate is a task in the nonpreemptive multitasking model of the Rabbit's Dynamic C. Given that different parts of the system execute in different costates, e.g., the application control loop is in one costate, the granularity of run-time monitoring is that of a costate. This is more accurate than monitoring the system as a whole.

The specifics of the instrumentation and how simple data structures can be used to achieve costate-based profiling is described in [3]. Using the data from the instrumentation, i.e., the observed profiles, one can detect off-nominal executions. However, rather than identifying off-nominal behavior, we "certify" nominal executions. Here we describe a dual-bound approach to the execution certification introduced in [5] and expand it to consider behavior sets. For ease of presentation we first introduce the notion without behavior sets. Furthermore, we discuss certification using modules as an example, but the principle can be extended to functionalities or operations.

Certifying module behavior per costate is based on module profiles, $\hat{\mathbf{p}}^k[\alpha]$. The distance of the observed costate profiles $\hat{\mathbf{p}}^k[\alpha]$ from $\overline{\mathbf{p}}[\alpha]$ can be used so that departure beyond it indicates non-certified behavior of costate α. Specifically, we define two threshold vectors

$$\epsilon^{max}[\alpha] = (\epsilon_1^{max}[\alpha], ..., \epsilon_{|M|}^{max}[\alpha]) \tag{3}$$

$$\epsilon^{min}[\alpha] = (\epsilon_1^{min}[\alpha], ..., \epsilon_{|M|}^{min}[\alpha]) \tag{4}$$

where $\epsilon_i^{max}[\alpha]$ and $\epsilon_i^{min}[\alpha]$ are the upper and lower threshold values of m_i, representing a dual-bound threshold. Every observed profile that is in the region between the two vectors is assumed nominal. Thus we certify a profile $\hat{\mathbf{p}}^k[\alpha]$ to be a *nominal profile* if

$$\epsilon^{min}[\alpha] \leq \hat{\mathbf{p}}^k[\alpha] \leq \epsilon^{max}[\alpha] \tag{5}$$

i.e., if $\epsilon_i^{min}[\alpha] \leq \hat{p}_i^k[\alpha] \leq \epsilon_i^{max}[\alpha]$ for every $1 \leq i \leq |M|$. The values of threshold vectors $\epsilon^{max}[\alpha]$ and $\epsilon^{min}[\alpha]$ are experimentally determined while the system is in *test mode*. Test mode here assumes a controlled environment in which the system runs normal and is closely observed while no fault occurs and no attacks on the system take place. In practice this means that, while in normal operation, the profiles are tracked over time to derive (or calculate) the desired threshold vectors. In the simplest case the threshold vectors can be the minimal and maximal observed values of each $\hat{p}_i^k[\alpha]$.

If one needs to tune the sensitivity of the thresholds, one can introduce weight functions w, defined per costate,

to be multiplied with the threshold vectors. Then a nominal execution of module m_i is defined as

$$w\epsilon_i^{min}[\alpha] \leq \hat{p}_i^k[\alpha] \leq w'\epsilon_i^{max}[\alpha]. \quad (6)$$

An alternative representation of execution certification is based on the deviation of the observed profile from the mean that has to be within the threshold vectors. Then an execution of module m_i is nominal if

$$|\hat{p}_i^k[\alpha] - \overline{p}| \leq \epsilon_i^{max}[\alpha] - \epsilon_i^{min}[\alpha]. \quad (7)$$

B. Certified Executions with Behavior Sets

Now we consider that there may be more than one nominal behavior of the system. For example, if during one operational epoch the system may exhibit one of several known behaviors, the nominal behavior is not unique anymore. Assume we are processing data files that have several known sizes, e.g., size a or b. Then one would expect the observed profiles of the executions in both cases to be different, yet both are nominal. However, if a file is corrupted and its size deviates largely from sizes a or b, one would like to detect this, e.g., by observing a profile vastly different from that of nominal executions. Thus we need to extend the notation of certified execution to consider behavior sets, introduced in subsection III-C. Recall the notational convention of changing lower case letters to upper case letters when behavior sets are used.

When using behavior sets the elements of threshold vectors are sets. Let $E^{min}[\alpha]$ and $E^{max}[\alpha]$ denote the behavior threshold vectors, i.e. $E^{min}[\alpha] = (E_1^{min}[\alpha], E_2^{min}[\alpha], ..., E_{|M|}^{min}[\alpha])$ and $E^{max}[\alpha] = (E_1^{max}[\alpha], E_2^{max}[\alpha], ..., E_{|M|}^{max}[\alpha])$. For a module m_i there will be at least one threshold value $\epsilon_{i,r}^{min}[\alpha] \in E_i^{min}[\alpha]$ and $\epsilon_{i,s}^{max}[\alpha] \in E_i^{max}[\alpha]$. The second subscript is the element number in the behavior set, e.g., element r and s.

The execution of module m_i is nominal if for some $\hat{p}_{i,t}^k[\alpha] \in \hat{P}_i^k[\alpha]$ and some $\epsilon_{i,r}^{min}[\alpha] \in E_i^{min}[\alpha]$ and $\epsilon_{i,s}^{max}[\alpha] \in E_i^{max}[\alpha]$

$$\epsilon_{i,r}^{min}[\alpha] \leq \hat{\mathbf{p}}_{i,t}^k[\alpha] \leq \epsilon_{i,s}^{max}[\alpha]. \quad (8)$$

V. System Operation & Contingency Management

In this section we describe the system as it operates and present data that was collected as part of the system mission and its monitoring over months of operation, especially the winter months of 2012.

A. System State Space and Transition Violations

As the system operates in the field it goes through state transitions, which are monitored by the *Operation Monitoring and Contingency Management System* (see Figure 2), using the three monitoring approaches introduced in Section IV. The system state space is generally complex, as it is induced by the state space of operations, functionalities, and modules. In our application, using the Mapping Simplification Assumption it reduced to the operation state space and module state space. The module state diagram was presented in [6], where a total of 25 system states was observed, i.e., $S_0, ...S_{24}$.

As the system software executes in costatements the transitions are verified to detect inter-dependency and intra-dependency violations described in subsections III-E and III-F respectively. Any violation of the system state transitions indicates a serious problem. For example, an intra-dependence violation of the module states implies that the system is calling modules that it should not be calling or it is returning to modules other than intended, e.g., as the result of a buffer overflow. Another example is if a module is called under an operation that should not utilize it, or a module returns to another module that is not operating under the same operation. Both cases can be deducted from the inter-dependency mappings. Upon detection of any dependence violation the contingency management system initiates fail-safe mode and issues a notification about the nature of the violation.

B. Application Control

The actions of the actual control sub-system, which is running in the costatement that implements the real-time traffic control application, was presented in [6]. The operation epoch is 15 minutes, which is the fixed time interval at which real-time weather condition data is available from the Clarus server for specific subscriptions (indicated by subscription numbers). First the Rabbit determines the time and composes the URL that contains the comma separated values, a *csv* file. It then uses a recovery block strategy to get the data from a set of data base servers. In our current implementation this is a Local Clarus Server (LCS) and the Clarus server shown in Figure 1, thus implementing a dual redundant system. First the LCS is queried and if the data cannot be retrieved within a certain amount of retries, the Clarus server is tried. If it fails to provide the data after a certain amount of retries, the Contingency Management System initiates *fail-safe mode*, which is a forward recovery mechanism bringing the system into a desired default state. Once entering fail-safe mode the system attempts to reestablish normal operation again. If data acquisition was successful via one of the alternatives the traffic controller computes the adjustments that reflect the environment parameters and adjusts the signal controller accordingly.

Certain assumptions must be made about the quality of the data supplied by the Clarus server. Whereas we assume that the data received is correct since Clarus uses data quality checking algorithms [11]. On the other hand, the computed signal adjustment values computed by the Rabbit are not assumed to be correct, as a fault may have occurred during computation. Therefore detection of anomalies through data sensor analysis, the second monitoring approach in Section IV, is implemented. It tests the computed signal changes for range violations from what was expected. If a violation is detected the contingency management system enters fail-safe mode. There are many other checking mechanisms, including reaction to network connection problems, data corruption, loss of time synchronization, e.g., after reboot as the result of power failure, inability of finding valid Clarus data subscriptions, changing the LCS Internet address. Some of these issues require the contingency management system to enter a *receive mode*, in which configuration information, e.g., the IP of a new LCS or Clarus subscription number, is communicated to the system.

C. Application Control Performance

The system is installed in Northern Idaho and has been observed over the most interesting period, which are the winter months, as adverse weather conditions are common. The adjustments of the yellow time of the traffic signals has been observed over several months. As the environment conditions worsen the yellow time is increased to improve safety, e.g.,

during ice or snow a longer yellow period allows more time to safely clear the intersection. The adjustment value computed by the Rabbit is communicated to the traffic controller, which in turn makes changes to a default value according to the percentage given by the adjustment value.

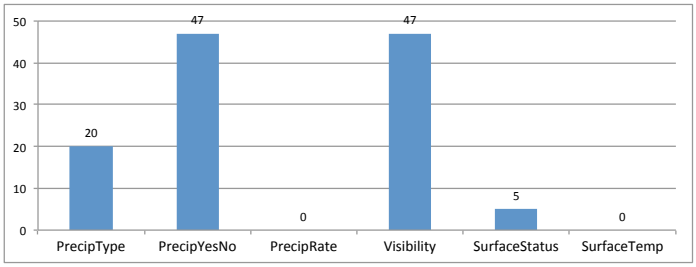

Fig. 7. Exception triggers

The yellow adjustment values for 53 interesting operational epochs during November and December of 2012 are shown in Figure 6. The figure shows the values *Surface Status*, *Surface Temperature*, which are Clarus parameters, *Weather Conditions*, and the adjustment values called *Yellow*. The first observation is that value *Yellow* follows the weather conditions. Next, looking at the surface temperature one can see that *Yellow* is increased as the temperature decreases. An alternate reference is the *Surface Status*, ST, a Clarus value shown in the lowest graph of the figure. Larger values of ST indicate deteriorating conditions, whereas conditions improve as ST become smaller. The values for ST are in the interval $[1, 14]$.

If one carefully examines the graph, one can see that there are five cases where $ST = 0$, which represents an error condition, i.e., the Clarus file was not available. The contingency management system recognized the fault and adjusted it to the most recent value, as can be observed in Figure 6.

In addition to the data sensor analysis we have exception triggers. Figure 7 shows five trigger and the count of how often exceptions were triggered, again based on the 53 observed operational epochs. For example, only two exceptions were never triggered. A trigger in this case implies that data needed for the computation of the yellow adjustments were missing in the data files. In fact, two of the five triggers were fired 47 times, i.e., out of 53 epochs, 47 did not include the specific sensor data, and 5 times the entire Clarus files were empty. However, the algorithm engine computing the adjustment could adapt to these scenarios since even in the absence of data the system could produce reasonable adjustment values. The data in the figure was actually observed as the consequence of changes in the data supplied from the ESS or Clarus. Since we have no control over what ESS sensor data may be missing, perhaps due to defects or product changes, the algorithm computing the adjustments to the application must be able to tolerate the missing data, which represent an omission fault. The exception triggers are used to verify the adjustment values, e.g., as seen in Figure 6, and provide adaptation.

D. Certified Executions for Resilient Operation

The third monitoring approach is to check for off-nominal executions. All but module m_{23} behave consistent in that their minimum, average, and maximum frequencies are equal. Only m_{23}, a module that filters Clarus data, experienced variation. Further examination of the behavior of m_{23} over time is shown in Figure 8, where the 52 operational epochs of Figure 6 are used. The counts of invocations of m_{23} is indicated, together with the minimum and maximum counts that would typically be used as the basis for the threshold function. However, m_{23} actually has two behaviors, i.e., one for non-empty files with no sensor data and another for standard file size. This causes the minimum and maximum to drift, as they are monotonically non-increasing and non-decreasing functions. As a result it is difficult to define effective thresholds for detection of off-nominal executions. This was solved by using a behavior set of size two, as shown in Figure 9. One behavior threshold reflects the small file size and another normal file size. The figure also identifies four readings that are off-nominal, by far overreaching the other readings. These files were abnormal data files of much larger size and are treated as if they were data falsification attempts.

E. Reliability Considerations

It is important to address how the addition of the real-time weather control application affects the reliability of the target application, i.e., the traffic control system. The reliability of the traffic control system is not affected by the addition of the embedded system since none of the components of the original traffic signal control system are modified. The embedded system implements only *added value*, but not basic functionality. In fact, as mentioned before, the traffic controller is NTCIP compliant. Thus action movie scenarios like an "all-green intersections" or "split second yellow timing" causing accidents are not possible (with or without the embedded system). Any attempt to assign parameters that violated NTCP compliance are simply ignored by the traffic controller, which we verified during testing of the embedded system. The only physical connection with the existing infrastructure is via the connection to the switch, as indicated in Figure 1.

VI. Conclusion

This paper described the architecture of a resilient control application operating in a critical infrastructure. The theoretical basis for effective run-time monitoring was given and the system has been observed over time. The experience gained so far indicate that the three monitoring approaches allow for adaptation of any changes we have observed so far. As the fault and attack vector that the system is exposed to is unknown, time and testing using fault injection will ultimately be the judge for its effectiveness. Further operation in the field will allow us to study the sensitivity of the certification parameters that implement the thresholds of nominal executions. However, it should be noted that this application, due to its NTCIP compliance cannot compromise safety. If it enters fail-safe mode due to off-nominal executions due to unknown origin, it only ceases to supply the added value. If, against all expectations, it were to completely fail and behave pathological it could not overwrite settings to violate safety margins.

References

[1] The Clarus System: http://www.clarus-system.com/

[2] A. Krings, *Survivable Systems*, in Information Assurance: Dependability and Security in Networked Systems. Morgan Kaufmann Publishers, 2008, ch. 5, pp. 113-146.

[3] A. Krings et al., *A Measurement-based Design and Evaluation Methodology for Embedded Control Systems*, in Proc. of the 7th Annu. Workshop on Cyber Security and Information Intelligence Research (CSIIRW'11), Oak Ridge National Laboratory, October 12 - 14, 2011.

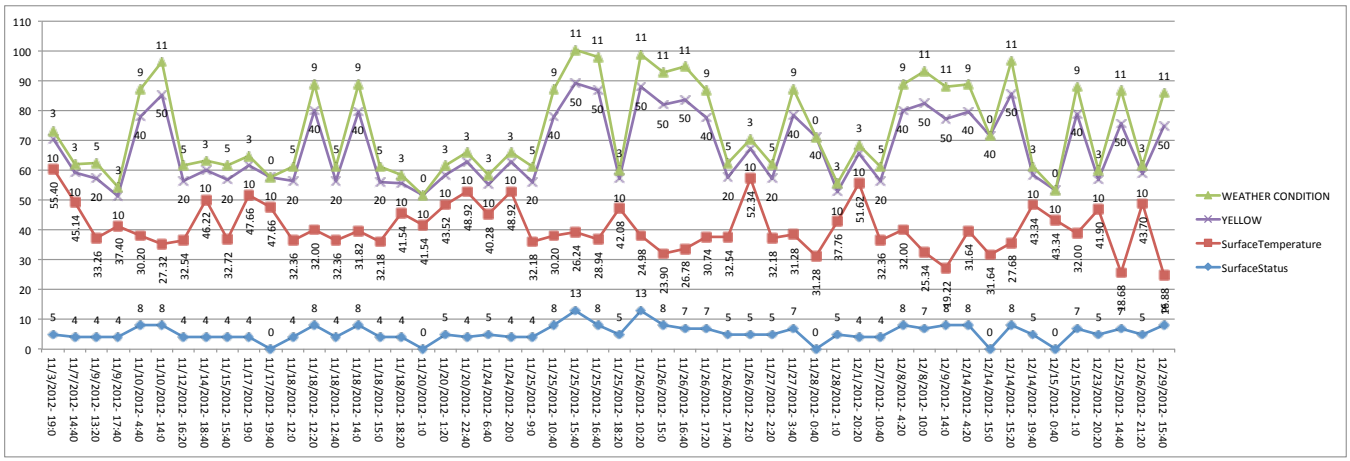

Fig. 6. Yellow adjustments during winter months related to weather conditions

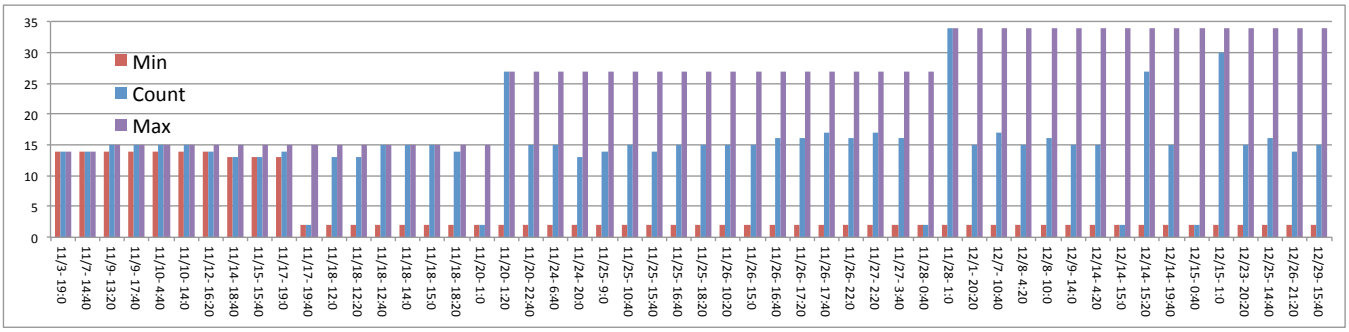

Fig. 8. Profiles of module m_{23} with behavior set size equal to 1 over operational epochs.

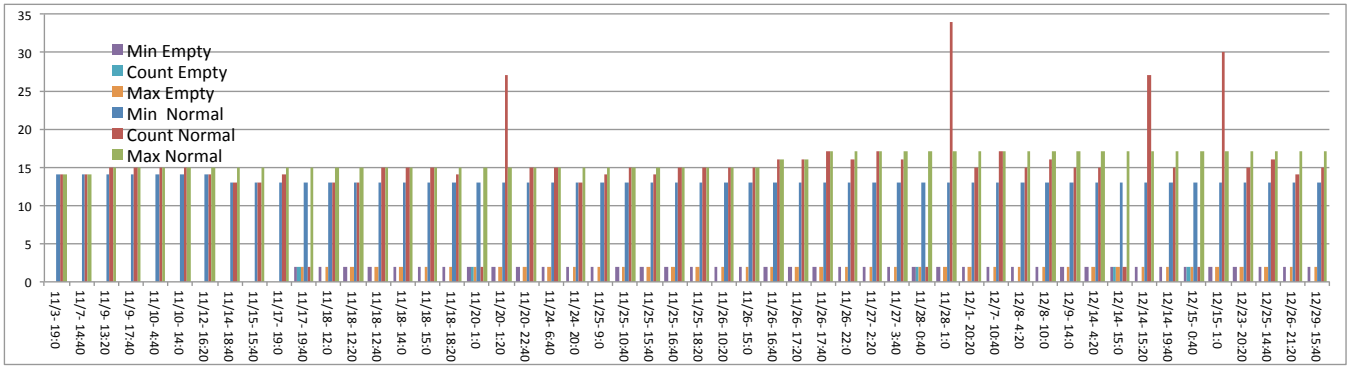

Fig. 9. Profiles of module m_{23} with behavior set size equal to 2 over operational epochs.

[4] J. Munson, A. Krings and R. Hiromoto, *The Architecture of a Reliable Software Monitoring System for Embedded Software Systems*, in Proc. 5th Intl. Topical Meeting on Nuclear Plant Instrumentation Controls, and Human Machine Interface Technology, 12-16 Nov 2006, pp. 765-774.

[5] A. Krings, A. Serageldin and A. Abdel-Rahim, *A Prototype for a Real-Time Weather Responsive System*, in the 15th Int. Conf. on Intelligent Transportation Systems (ITSC), Anchorage, Alaska, September 16-19, 2012, pp. 1465-1470.

[6] A. Serageldin, A. Krings, and A. Abdel-Rahim, *A Survivable Critical Infrastructure Control Application*, in Proc. of the 8th Annu. Cyber Security and Information Intelligence Research Workshop (CSIIRW'13), Oak Ridge National Laboratory, January 8 - 10, 2013.

[7] V. Chandola, A. Banerjee, and V. Kumar, *Anomaly Detection: A Survey*, in ACM Computing Surveys (CSUR), July 2009.

[8] J. Allen et.al., *State of the Practice of Intrusion Detection Technologies*, Software Engineering Institute,Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU/SEI-99-TR-028, 2000.

[9] S. Elbaum and J. Munson, *Intrusion Detection Through Dynamic Software Measurement*, in Proc. of the 1st conf. on Intrusion Detection and Network Monitoring, Santa Clara, CA, April 9-12, 1999, pp. 41-50.

[10] A. Krings et.al., *A Two-Layer Approach to Survivability of Networked Computing Systems*, in Proc. of the Int. Conf. on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet, (SSGRR'2001), L'Aquila, Italy, Aug. 06-12 2001, pp. 1-12.

[11] M. Limber, S. Drobot, and T. Fowler, *Clarus Quality Checking Algorithm Documentation Report*, Final Report, RITA Intelligent Transportation Systems Joint Program Office, Technical Report FHWA-JPO-11-075, December 21, 2010.