A Prototype for a Real-Time Weather Responsive System*

Axel Krings, Ahmed Serageldin[1] and Ahmed Abdel-Rahim[2]

Abstract— **This paper presents a prototype of a secure, dependable, real-time weather-responsive system. The prototype performs two operations: 1) it accesses weather information that provides near-real-time atmospheric and pavement observations and 2) it adapts signal timing in response to inclement weather. Since this real-time control system operates in a critical infrastructure, it must be designed with built-in security and survivability mechanisms. For this purpose a software architecture is presented that uses real-time monitoring to detect precedence violations and off-nominal system execution in order to provide effective contingency management. The focus of these systems is the autonomous recognition and reaction to execution patterns that have not been previously observed and thus fall outside of the known behavior. Because the described system has very similar requirements to other traffic control applications, it serves as a milestone in the development of secure and dependable real-time traffic control systems.**

I. Introduction

This paper presents the prototype of a real-time weather-responsive system. The prototype system design incorporates state-of-the-art secure and dependable software design concepts to ensure accurate execution of two operations. For the first operation, the system accesses weather information from the Federal Highway Administration (FHWA) Clarus data system [2] that provides near-real-time atmospheric and pavement data from participating states' environmental sensor stations (ESS). The second operation adapts signal timing in response to inclement weather. Since these operations involve communication over the Internet, the application is subject to the full pallet of Cyber security issues with generally unknown overall attack vectors. Given that the real-time system controls part of a critical infrastructure, special focus on security and survivability is imperative.

The proposed system employs two revolutionary software design approaches: *Design for Survivability* [5], [6] and a *Measurement-Based Methodology* [4], [8]. The latter 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. Furthermore, the software design incorporates self-monitoring techniques for fault detection and recovery to maximize the survivability and the security of the system. Minimal hardware is required for full implementation of the system as it operates and achieves its potential using current traffic controller and cabinet standards and technologies. As a result, it is compatible with future applications within the FHWAs connected-vehicle (formally IntelliDrive) initiative.

In order to understand the basic philosophy of this research it is important to understand the concept of *survivability*. There is no single agreed-upon definition for system survivability. Instead, one may use as a starting point the vague notion that a system has to be able to tolerate diverse faults. This includes those faults typically considered in the area of fault-tolerant system design, such as faults resulting from component failure as a consequence of aging, fatigue or break-down of materials. These faults may exhibit very predictable behavior and frequency. Software faults are more difficult to describe, however, they essentially cause the system to enter a state that deviates from the specification. In the last two decades there has been much attention on (humanly induced) malicious faults, e.g. hacking, denial of service, virus, Trojan horses, spoofing etc. These kinds of faults may affect the software and even the hardware and can be totally unpredictable. They are the main target of security and survivability considerations.

The many definitions of survivability can be loosely partitioned into *qualitative* and *quantitative* definitions. Qualitative definitions mainly focus on guaranteeing that essential functionalities of the system are maintained, in a timely manner, even in the presence of faults and attacks; "the mission must survive" [3]. Quantitative definitions imply that survivability can be quantified, e.g., measured, and assume a formal model. For a closer look at survivability, its definitions and implications, the interested reader is referred to [6]. The work presented here incorporates both qualitative and quantitative aspects of survivability as will be shown below.

As indicated before, the source of addressing survivability is the capability of dealing with faults. There are too many fault sources to list them individually and exhaustively. Therefore the notion of *fault models* is used, capturing the behavior of a fault, i.e., a fault can produce an error that then can lead to a failure. The diversity of faults and their consequences on a system have been the primary motivation for the definition of fault models. A fault model addresses the behavior of the faults and specifies the redundancy levels required to tolerate a single fault type or a mix of fault types. Many different fault models have been proposed over the years ranging from the simple models that make no assumptions about the fault behavior [7], to hybrid fault models considering multiple fault behavior. The latter considers a mix of faults ranging from benign, symmetric to asymmetric faults [Thambidurai 1988], with potential transmissive and omissive behaviors [1]. The latter fife-fault

*This research was supported by grant DTFH61-10-P- 00123 from the Federal Highway Administration - US DoT

[1]Axel Krings and Ahmed Serageldin are with the Computer Science Department of the University of Idaho, USA

[2]Ahmed Abdel-Rahim is with the Civil Department of the University of Idaho, Moscow, ID, USA

model of [1] constitutes the basis for the faults addressed in the described system. Omission faults will be emphasized, because communication may be interrupted. Furthermore, value faults (symmetric and asymmetric) such as infeasible or incorrect input or output data were also deemed important, since any of those faults has the potential to decrease safety.

II. Real-time Weather Response System

An overview of the real-time weather response system is shown in Fig. 1. Weather data is collected by the Clarus system from a network of environmental sensor stations, ESS, of participating states. This data is accessible via the Internet, and it which undergoes quality and consistency checks based on their quality checking algorithm. Thus, survivability considerations are not questioning the quality of the original Clarus data. A microcontroller (Processing Unit) located in the traffic signal system in the intersection retrieves the Clarus System data, analyses the relevant data, and computes changes of signal timing. Upon approval, the signal timing changes are made in the Traffic Controllers by the Processing Unit. 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.

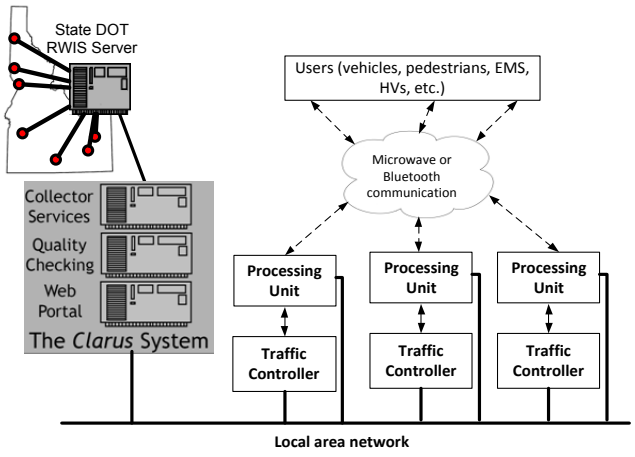

Fig. 1. Overview of the real-time weather response system

A. CLARUS Real-time Weather Data Support

The Clarus System shown in Fig. 1 maintains the location of the ESS. The ESS most suitable for the specific traffic signal system, e.g., the one with closest proximity 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 protected from general access by a password, publicly readable. Specifically, a list of Observations, i.e., the actual CSV files, is given in regular intervals, which depend on the subscription and typically range from 5 to 15 minutes. These 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 on the traffic controller. Since a subscription is not limited to contain data from only one ESS, but can be specified to contain data from multiple sensors, e.g., to include neighboring ESS, the control algorithms of the weather responsive system can take advantage of data fusion, thus taking advantage of a "larger view".

B. General System Description

A Rabbit 5700 microprocessor running Dynamic C[1] is the core hardware in the system that communicates with the traffic controller through Ethernet. To facilitate communications, the controller and microprocessor must follow the National Transportation Communications for ITS Protocol (NTCIP) communication standard (AASHTO 2005), a family of standards for transmitting data and messages between different devices used in Intelligent Transportation Systems (ITS). The Dynamic Object STMP/UDP/IP Ethernet protocol stack is used to facilitate the NTCIP-based communication between the microprocessor and the traffic controller. A computer, connected to the microprocessor through the cabinet serial connection, is used to setup and add the control logic to the microprocessor. Because the microprocessor is directly connected to the traffic controller through the Ethernet port, the connection is not sensitive to the cabinet configuration. However, the microprocessor requires an additional 110 volt power connection. This connection method should be possible in any NTCIP compliant controller.

The Rabbit system, which we will refer to simply as "Rabbit", executes the software that implements the real-time weather responsive system, which consists of the Operational Software System and an Operation Monitoring and Contingency Management System. An overview of the software system is shown in Fig. 2, where shaded areas refer to external hardware interfaces. The 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 the regular intervals specified in the subscription the Clarus data is read and converted by the Rabbit, the desired sensor data is extracted, and specific algorithms compute for example the yellow timing from the critical extracted sensor parameters. The traffic controller is then updated. All this is monitored by the Rabbit at run-time.

As the application software is executing on the Rabbit, it monitors the execution of its software via sensor points injected by instrumentation in real-time. The principle is shown

[1]Dynamic C and Rabbit are registered trademarks of Digi International Inc. See documentation at www.rabbit.com

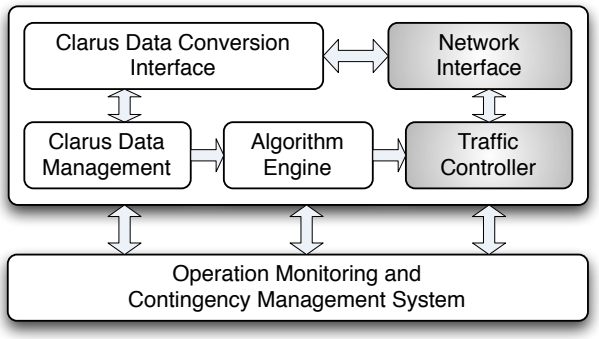

Fig. 2. Overview of the software system architecture

in Fig. 3, where the executing program is monitored via the instrumentation telemetry by the *Operation Monitoring and Contingency Management System*. Due to its complexity the exact function of this system will be described in a separate publication. The instrumentation data is analyzed at run-time and the outcome of the analysis is used to adapt to observed behavior, or during design time to alter design parameters, as described in [8]. The general methodology is the basic mechanisms for "design for survivability" and allows for real-time reaction, e.g., reconfiguration, state changes, or entering a fail-save mode.

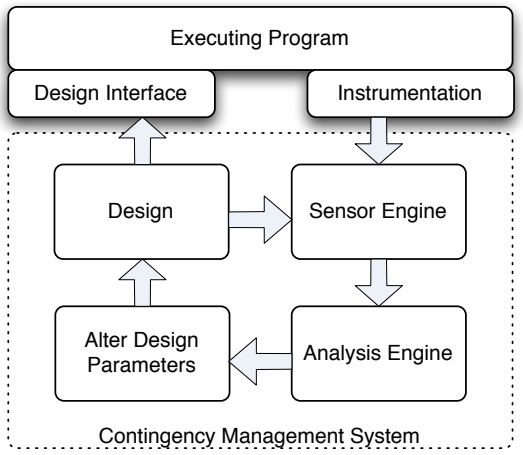

Fig. 3. Overview monitoring and contingency management system

III. Survivability Architecture

We now present the survivability architecture of the system, which is based on the general principles shown in [4], [8]. In the following we subscribe to the general notation of these papers and partially restate important concepts or definitions.

A. Formal System Model and Dependencies

During operation of the system, and with proper instrumentation of the software, one can get a "life" picture of how the system is performing in real time, e.g., what a typical execution looks like, how often functionalities are invoked, what mix of software modules is instantiated over a certain window of observation, or how often certain modules get called during a time interval. All information related to counting of events or invocations is captured in profiles, which may also be represented as frequency spectra. Calling behavior like specific execution sequences, which represent "who is calling (or invoking) whom", is captured in precedence graphs. These graphs are static in an executing system. However, they may change as the result of design alterations associated with the design cycle shown in Fig. 3. An example of such graph is the call graph of software modules as it is generated at compile time.

Just as in [8] the system executes a set of operations O, e.g., "Get Clarus Data" or "Update Controller", with cardinality $|O|$. These operations constitute the *operational machine*. The transition from one operation to another marks an *operational epoch*. Each operation o_i in O uses one or more functionalities f_j from a set F of functionalities with cardinality $|F|$. Similar to the operational epoch the functional epoch is defined by transitions from one functionality to another. Functionalities are implemented by code modules written in Dynamic C, which is a C-like language with a unique multitasking environment as will be described in subsection III-C. The set of modules M of cardinality $|M|$ is thus the implementation of the functionalities in code. 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.

Operations, functionalities and modules are related and that relationship can be defined in a graph $\mathcal{G}^{OFM}$, where the superscript simply indicates that the graph maps from sets O to F to M. The vertex set of $\mathcal{G}^{OFM}$ is thus $O \cup F \cup M$ and the edges define the *utilization relation* from O to F to M, i.e., in graph $\mathcal{G}^{OFM}$ each operation $o_i \in O$ is mapped to the functionalities f_j it utilizes, and each $f_j \in F$ is mapped to the modules $m_k \in M$ it utilizes, which are of course the models that implement f_j. In Fig. 4 these dependencies are shown as dotted edges. Mappings can take on many shapes. For example, a functionality f_j may be part of one or more operations and it may be implemented by one or many modules. Similarly, a module m_k may be used in several functionalities. Graph $\mathcal{G}^{OFM}$ thus allows to validate at runtime if for example a module is supposed to be executed in the context of a specific functionality or if observed calls violate the valid O to F to M dependencies. A violation could be the result of a buffer-overflow attack where suddenly the known mappings are violated.

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 the dependencies within operations, functionalities, or modules. Those relationships can be defined by precedence graphs within the shaded areas of Fig. 4. Specifically, dependencies between operations are defined by graph $\mathcal{G}^O = (O, \prec)$, where $\prec^O$ defines a

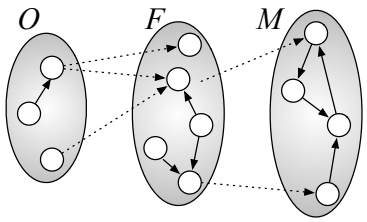

Fig. 4. Dependencies between operations, functionalities, and modules

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 of the program. 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.

B. Profiles and Profiling

Leaning on the notation of [8] 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_l denote the probability that the system is executing module m_l. 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.

During execution of the system we are interested in observing the module profile over n epochs. This observed profile is $\hat{\mathbf{p}} = (\hat{p}_1, \hat{p}_2, ..., \hat{p}_{|M|})$, 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 of an "h-day moving average" in financial stock movements, i.e., we will derive a centroid that will serve as a reference for observed profiles. For that, just as in [8], 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.

One can compute the distance of an observed profile $\hat{\mathbf{p}}^k$ from centroid $\overline{\mathbf{p}}$ to get a distance scalar d_k

$$d_k = \sum_{i=1}^{n} (\overline{p}_i - \hat{p}_i^k)^2 \tag{2}$$

Given the computational realities of the Rabbit, we actually use $d_k = \sum_{i=1}^{n} |(\overline{p}_i - \hat{p}_i^k)|$, rather than the square. The goal is to analyze the effectiveness of using the distance of observed profiles from the centroid to detect off-nominal executions.

C. Run-time Model

One of the challenges in monitoring a system is dealing with the effects of nondeterminism of the executions. Typical sources of nondeterminism are interrupts, or even more importantly, context switching in multiprocessing based on time slicing. The Rabbit system uses a single processor in which multitasking is not achieved using time slicing; rather it is implemented using a model defined by *costatements*. A costatement is defined as a task in a nonpreemptive multitasking model. In practice, the main program of a control applications runs costatements (the tasks) in an endless loop, cycling from one costatement to the next. Each costatement has a statement counter, i.e., a program counter, which indicates which instruction of the costatement will execute when it gets a chance to run. Execution is switched from one costatement (of the infinite loop) to the next in a round-robin fashion when the currently executing costatement "yields" to the next costatement using explicit commands such as *yield*, *abort* or *waitfor(event)*. Note that these yielding mechanisms represent a model that is based on good behavior. The state of a costatement is called a *costate*. We will use the terms costatement and costate interchangeably.

An execution model in which there are no externally initiated task switches executes with a low level of non-deterministism, i.e., a task switch is explicitly demanded by the currently executing task: the *active costatement*. On the other hand this means however that it is possible for a costatement to cause starvation by not yielding. However, a special mechanism called watchdog can be used to force timer interrupts. In this case the system deviates from its otherwise nonpreemptive execution model.

As operations, functionalities, and modules are called from within exactly one costatement at a time, it is possible to precisely determine the functionality and module that are being executed on behalf of a specific operation. Thus, the dispatching model results in executions with a low degree of nondeterminism, which is very desirable when working with profiles.

With the introduction of costates we can now extend the definitions of profiles presented in Subsection III-B to profile on a costate-basis. Thus, the observed profile $\hat{\mathbf{p}} = (\hat{p}_1, \hat{p}_2, ..., \hat{p}_{|M|})$, the k^{th} module profile $\hat{\mathbf{p}}^k$, the centroid $\overline{\mathbf{p}} = (\overline{p}_1, \overline{p}_2, ..., \overline{p}_{|M|})$ and d_k, i.e., the distance from $\hat{\mathbf{p}}^k$ from centroid $\overline{\mathbf{p}}$, can now be defined on a costate-basis. For

a costate α this leads to notation $\hat{\mathbf{p}}[\alpha]$, $\hat{\mathbf{p}}^k[\alpha]$, $\overline{\mathbf{p}}[\alpha]$ and $d_k[\alpha]$ respectively. Now it is possible for each costate α to have its own profiling, which is not affected by any non-determinism due to costate (task) switching, i.e., profiles of costates do not interfere.

D. Run-time Monitoring and Certified Executions

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 specified 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, or 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.

Our run-time monitoring employs two approaches:

1) *Validation of Dependencies*: Given the two types of dependencies shown in Fig. 4 the system can detect any violation of mappings from operations to functionalities to modules in $\mathcal{G}^{OFM}$, and any violations of precedence in each $\mathcal{G}^O$, $\mathcal{G}^F$ and $\mathcal{G}^M$.
2) *Detection of off-nominal executions*: Here observed profiles are checked to establish if they meet an expected certified behavior, as will be described in the rest of this subsection in the context of certifying nominal profiles.

Detection of off-nominal executions is less precise of a science than checking for precedence violations. Our approach to the first does not attempt to mimic anomaly detection, but it utilizes 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. Nonimal behavior can be refined to a costate level. Thus, 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, and thus more accurate than that of a system lacking that refinement.

The specifics of the instrumentation and how simple data structures can be used to achieve costate-based profiling is described in [4]. Using the data from the instrumentation, i.e., the profiles, one can detect off-nominal executions. However, rather than identifying off-nominal behavior, we "certify" nominal executions. Here we describe a new dual-bound approach to execution certification by extending the certification of [4]. The result is a more stringent view of nominal executions.

Certifying behavior per costate is now possible and will again be described using 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 $\epsilon_i^{max}[\alpha]$ 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 this could be the minimal and maximal observed values of each $\hat{p}_i^k[\alpha]$. Alternatively, one could introduce weight functions w, defined per costate, to be multiplied with the threshold vectors. Then a nominal profile would satisfy $w\epsilon_i^{min}[\alpha] \leq \hat{p}^k[\alpha] \leq w'\epsilon_i^{max}[\alpha]$. For completeness sake it should be noted that the "threat vector" of a system is of course unknown, but it appears realistic to make the assumption that in a controlled environment no unobserved faults or malicious act can sneak in.

E. Experimental Results

A prototype has been built based on a Rabbit 5700 running Dynamic C version 10.5.4, which has been instrumented to allow operation, function and module profiling. Initially all Dynamic C modules were instrumented, including library modules. However, not having the Dynamic C library source code we did not instrument the library assembler routines. That in itself would have been fine, had it not been for the fact that some of the non-instrumented assembler routines called C modules, thereby breaking out precedence violation detection capability. We therefore had to eliminate the instrumentation to those modules. A total of 71 modules furnished the data for the observed profiles, the centroid and the dual-bound threshold vectors. For the figures shown below a subset of only 56 relevant modules was used.

Fig. 5 shows an actual observed profile which encapsulates the data transmission from the Clarus systems. The x-axis indicates the module ID and the y-axis shows the frequencies in logarithmic scale. The figure depicts profile $\hat{\mathbf{p}}^k[\alpha]$ for four costates: 1) System Configuration, 2) Application Control, 3)

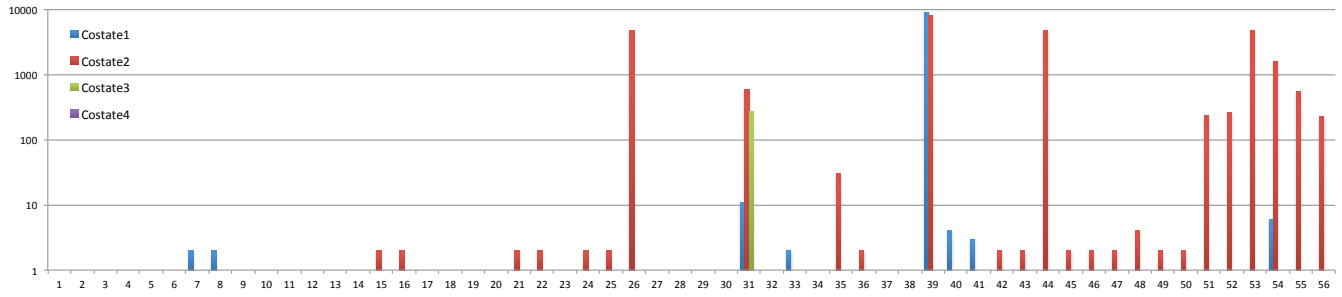

Fig. 5. Typical observed profile of 4 costates (module IDs and frequencies on the axis)

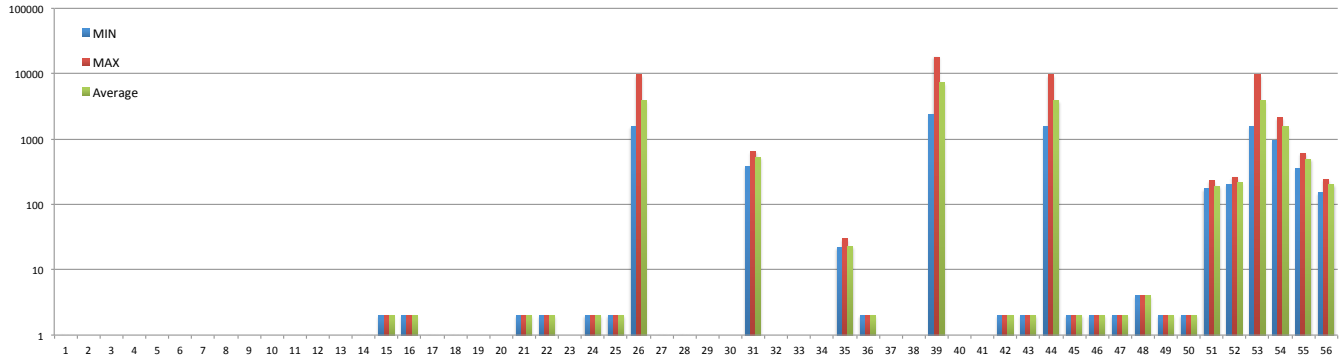

Fig. 6. Centroid, and dual-bound threshold vectors (module IDs and frequencies on the axis)

Monitoring and 4) Utilities. As expected, costate 2 with its application control dominates the spectrum, i.e., $\hat{\mathbf{p}}^k[2]$. The magnitude of the frequencies is of course highly dependent on the size of the Clarus subscription.

An example of the average, minimum and maximum frequencies are shown in Fig. 6. The average constitutes centroid $\hat{\mathbf{p}}^k[\alpha]$, in this case $\hat{\mathbf{p}}^k[2]$ for costate 2. For this specific Clarus subscription the certification space between $\epsilon^{min}[2]$ and $\epsilon^{max}[2]$ appears rather tight. However, the frequency count is shown as log-scale. For example, if we look at module m_{25}, which is *filter_clarus_data*, the actual value for the centroid $\overline{\mathbf{p}}[\alpha]$ is $\overline{p}_{25}[2] = 3873$. The minimum and maximum number of invocations, which we used for the dual-bound threshold functions, were $\epsilon^{min}_{25}[2] = 1564$ and $\epsilon^{max}_{25}[2] = 9665$ respectively.

As indicated before, threshold vectors are generated by observing the system in a learning mode over time, thereby letting observed executions affect what is considered a nominal execution. In our example, over the time of the observations the specific values indicated arose. For m_{25} this meant that as long as $\hat{p}^k_{25}[2]$ was in the interval $[1564, 9665]$ the execution was considered nominal.

IV. Conclusions

A prototype of a real-time weather responsive system has been described. This system can take real-time weather data and modify traffic signal timing within safety standard. The design of the system employs a state-of-the-art design methodology that incorporates design for survivability. It allows to monitor its executions to detect 1) violations of dependencies of operations, functionalities, and modules, and 2) the detection of off-nominal observed execution profiles. The latter is established by checking if the observed profiles are within a dual-bound threshold space that defines nominal profiles. Current efforts are to test the effectiveness of the detection of off-nominal executions in preparation for field tests.

References

[1] M.H. Azadmanesh, and R.M. Kieckhafer, *Exploiting Omissive Faults in Synchronous Approximate Agreement*, IEEE Trans. Computers, 49(10), pp. 1031-1042, Oct. 2000.

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

[3] R. J. Ellison, D. A. Fisher, R. C. Linger, H. F. Lipson, T. Longstaff and N. R. Mead, *Survivable Network Systems: An Emerging Discipline*, Technical Report CMU/SEI-97-TR-013, November 1997, Revised: May 1999.

[4] A. Krings, V. Balogun, S. Alshomrani, A. Abdel-Rahim, and M. Dixon, *A Measurement-based Design and Evaluation Methodology for Embedded Control Systems*, 7th Annual Cyber Security and Information Intelligence Research Workshop, CSIIRW'11, Oak Ridge National Laboratory, October 12 - 14, 2011.

[5] Axel Krings, *Design for Survivability: A Tradeoff Space*, Proc. 4th Cyber Security and Information Intelligence Research Workshop, Oak Ridge National Laboratory, May 12-14, 2008.

[6] Axel Krings, *Survivable Systems*, Chapter 5 in: Information Assurance: Dependability and Security in Networked Systems. Morgan Kaufmann Publishers, Yi Qian, James Joshi, David Tipper, and Prashant Krishnamurthy Editors), in press, 2008.

[7] L. Lamport, et.al., *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp. 382-401, July 1982.

[8] John Munson, Axel Krings and Robert Hiromoto, *The Architecture of a Reliable Software Monitoring System for Embedded Software Systems*, ANS 2006 Winter Meeting and Nuclear Technology Expo, 2006.