# Architecting Health Management into Software Component Assemblies: Lessons Learned from the ARINC-653 Component Model

Nagabhushan Mahadevan Abhishek Dubey Gabor Karsai

*Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37212, USA*

***Abstract*—Complex real-time software systems require an active fault management capability. While testing, verification and validation schemes and their constant evolution help improve the dependability of these systems, an active fault management strategy is essential to potentially mitigate the unacceptable behaviors at run-time. In our work we have applied the experience gained from the field of Systems Health Management towards component-based software systems. The software components interact via well-defined concurrency patterns and are executed on a real-time component framework built upon ARINC-653 platform services. In this paper, we present the lessons learned in architecting and applying a two-level health management strategy to assemblies of software components.**

## I. Introduction

Software is used heavily in modern systems - both to implement the core functionality as well as to integrate functions across various subsystems [22]. It is well known that software can contain latent defects that escape the existing rigorous testing and verification regimes and manifest only under exceptional circumstances. These circumstances may comprise faults in the hardware system, including both the computing and non-computing hardware. Often, systems are not prepared for such faults, which have led to a number of incidents in the past, including but not limited to those referred to in these reports: [19], [5], [6], [14].

Software Health Management (SHM) is an extension of software fault tolerance using techniques borrowed from System Health Management of complex engineering systems [28]. The goal of SHM is to make systems self-managing such that they exhibit resilience to faults by adaptively mitigating the effects of those faults. Recent work in this area includes [21], [27], [18], [4].

We have developed an architecture and approach for implementing software health management functions for component-based software systems [12]. Foundation of the architecture is a real-time component framework (built upon an ARINC-653 platform) that defines a specific model of computation for software components [11]. This framework brings the concept of temporal isolation, spatial isolation, and strict deadlines from ARINC-653 and merges these with the well-defined interaction patterns described by the CORBA Component Model [30]. The health management function in the framework is performed at two levels, see figure 2. The Component-level Health Manager (CLHM) provides localized and limited service for managing the health of individual software components. A System-Level Health Manager (SLHM) manages the health of the overall system.

SLHM includes a diagnosis engine that uses a Timed Failure Propagation (TFPG) [2], [3] model that is automatically synthesized from the component assembly. Note the distinction between diagnosis and detection. Diagnosis is the process of identifying and isolating the root cause(s) of a detected anomaly. In our work, the diagnosis engine reasons about fault effect cascades in the system, and isolates the fault source components. This is possible because the data and behavioral dependencies and hence the fault propagation across the assembly of software components can be deduced from the well-defined and restricted set of interaction patterns supported by the framework. Once the fault source is isolated, the necessary system level mitigation action is taken. Similar approaches can be found in [10], [29]. The key difference between these and our work is that we apply an online diagnosis engine coupled with a two-level mitigation scheme.

This paper summarizes our current work along these lines - component model & design tools (section I-A), component & system level health managers (section II), diagnosis scheme (section III). Finally section IV discusses the lessons learned in adapting and applying a system-level diagnosis approach to software health management. The lessons learned are discussed from the perspective of improving the quality and correctness of diagnosis and we do not delve into assessing the performance of the current strategy (CLHM/SLHM ) through quantitative measures.

### A. Overview of the ARINC-653 Component Model (ACM)

In our approach to Software Health Management we assume that software is built as an assembly of components, where the individual components comply with a specific component model. The component model defines the component ports that facilitate interactions among components.

The ARINC-653 Component Framework is the runtime code that implements the ARINC-653 Component Model (ACM), and it was introduced in [11]. ACM is built upon the services of the ARINC-653 platform; an avionics standard

for safety critical operating systems [1]. ARINC-653 systems group *processes* into spatially and temporally separated *partitions*, with one or more partitions assigned to each *module* (i.e. a processor), and one or more modules forming a *system*. Spatial partitioning ensures the exclusive use of a (virtual) memory region by a partition. Temporal partitioning ensures the exclusive use of the processing resources by a partition. Partitioning also ensures that rogue and faulty processes do not corrupt the memory or hog the CPU resource in other partitions. In a multi-processor system, the runtime framework provides facilities to synchronize the start and end of the hyper periods of all processors.

The ARINC-653 component model allows the developers to group a number of ARINC-653 processes into a reusable component. At the component level, the role of a process is based on the type of the component port it is attached to. The component model defines the following component port types: **publishers, consumers, facets** (a.k.a. provided interfaces[1]), **receptacles** (a.k.a. required interfaces), and **methods** (methods are internal to the component). Each component port is mapped to one active ARINC-653 process that is used to execute some business logic.

Inter-component interactions are based on well-defined interaction patterns borrowed from other software component frameworks, notably from the CORBA Component Model (CCM) [30]. Components can interact with other components through **synchronous** call/return interfaces (associated with facets or receptacles), and/or via **asynchronous** publish/subscribe event connections (assigned to publisher and consumer ports), facilitated by ports. While the facet and receptacle ports are associated with an interface type (a named collection of methods), the publisher and consumer ports are associated with an event type (a data structure).

### B. Design and Generation of the Software Component Assembly

Supporting tools accompanying the ARINC-653 Component Framework enable the specification of components, their ports with types. Additional real-time properties of the port (corresponding to the ARINC-653 process) such as periodicity, deadline, worst case execution time etc. can also be specified. Further data and control flow dependency between the ports of a component can be modeled using a graphical tool.

Once a library of components has been created, these can be used to put together the design of a sub-system or a system assembly. The integrator can create the assembly model by instantiating and connecting components or subsystem models, thereby capturing the interaction across the assembly. The design constraints enforced by the modeling tool ensure that all ports are properly connected e.g. the type of publisher matches the subscriber.

[1]An interface is a collection of related methods.

**Deployment** details can be specified by modeling the platform (i.e. the ARINC-653 modules), the partitions associated with each module, and then specifying the partitions where each component in the assembly is to be deployed. Note that even though ACM assemblies are multi module, ACM does not specify any particular networking implementation to be used.

Thereafter the code generators included in the modeling tool suite[2] generate the glue code that creates partitions, ARINC-653 processes, bindings between the component ports and the ARINC-653 processes, code to map the developer provided business logic to the ARINC-653 APIs, configuration files for the schedule information for each module, and additional details to facilitate the inter-partition communication.

## II. Two-level Software Health Management

A nominal assembly without any fault management support can be modeled, deployed, and executed to achieve functional goals using a design- and run-time framework that follow the concepts discussed in the previous sections. This section describes the additional support in the design, generated code and runtime framework to support a two-level health management strategy as discussed in the introduction. However, we must first discuss the execution states and various faults that can occur in each component.

### A. Component Execution States, Faults, and Anomalies

Any component, once deployed in the system can be in one of the following three states: **active** (all ports are operational), **inactive** (no port is operational) and **semi-active** (only consumer and requires port are operational). Typically, a component is in active state under nominal operation, semi-active when it serving as a passive replica, inactive when it is faulty or not required currently. Note that components are equipped with a lock that ensures that at most one thread (process) can be active in a component at any time.

**Failure Sources** While the component is executing, i.e. it is in active or semi-active state, anomalies detected in component ports can indicate faults in the larger system. We consider two cases : (a) faults related to the environment of the component that manifests itself as an anomaly with acquiring the component lock (this can be related to scheduling problems, etc.), and (b)a latent defect in the developer-supplied, functional code implementing the business logic of the component port.

Both these fault sources can lead to anomalies in either the same component or in a connected component. In our framework, the design tools allow the system designer to deploy monitors which can be configured to detect deviations from expected behavior, violations of specifications:

[2]These tools can be downloaded from https://wiki.isis.vanderbilt.edu/mbshm/

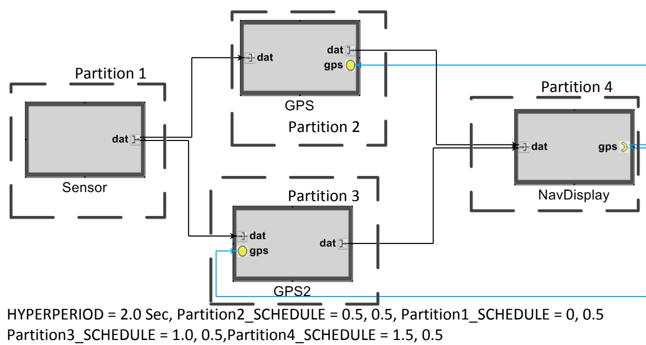


Figure 1. GPS Software Assembly - Unit of time is seconds.

conditions and constraints associated with an interaction port or component. Based on these monitors, following anomalies can be detected:

- *Lock timeout*: The framework implicitly generates monitors to check for process starvation. Each component has a lock (to avoid interference among callers), and if a caller does not get access to the lock within a specified time, an anomaly is generated. The value for the timeout is either set to a default value equal to the deadline of the process associated with component port or can be specified by the system designer.
- *Data validity violation* (only applicable to consumer ports): Any event data token consumed by a consumer port has an associated expiration age. This is known as the validity period in ARINC-653 sampling ports. We have extended this to be applicable to all types of component consumer ports, both periodic and aperiodic.
- *Pre-condition violation*: Developers can specify conditions that are evaluated before executing the functional code. These conditions can be expressed over the current value or the historical change in the value, or rate of change of values of function call parameters of the and the state variables of the component.
- *User-code failure*: Any error or exception raised in the user code can be treated by the software developer as an error condition which can then be reported to the framework. Any unreported error is recognized as a potential unobservable anomaly.
- *Post-condition violation*: These are similar to pre-condition violations, but the conditions here are checked after the execution of the functional code associated with the component port.
- *Deadline violation*: Detected when a process does not finish its execution within a specified deadline.

Figure 1 shows an assembly model with redundant GPS systems. This model shows the connection between the components and their deployment on four different partitions. Partition 1 contains the Sensor Component. Partition 2 contains the GPS, Partition 3 contains the redundant GPS (GPS2) component and Partition 4 contains the Navigation Display component. The Sensor component publishes an event every 4 sec. The GPS component consumes the event published by sensor at a periodic rate of 4 sec. Afterwards it publishes an event, which is sporadically consumed by the Navigation Display (abbreviated as NavDisplay or Display). Thereafter, the display component updates its location by using getGPSData facet of the GPS Component.

### B. Component Level Health Manager

In ACM, each component can be equipped with a Component Level Health Manager (CLHM). During component design, the CLHM is modeled as a hierarchical timed state machine. It captures the reactions or mitigation actions for the component anomalies (discussed above), given the current state of the CLHM. The health manager model can also include one or more observer automata that are parallel state machines that track the state-evolution and /or the sequence of operations executed in the component and report violations to the health manager. Basic component level mitigation commands enable a component developer to ignore the anomaly, abort the current operation, use previoues data, or completely stop the execution of the component. In all cases, the default action is to report the anomaly and the local mitigation action to the System Level Health Manager (SLHM), discussed next.

### C. System Level Health Manger

In ACM the infrastructure to support system level health management is created through automated code synthesis involving additional dedicated components and architectural extensions to integrate the new components with the existing functional component assembly. The customized mitigation strategy that is hosted in the SLHM is auto-generated from state machine models designed by the system integrator. These hierarchical state machine models capture the reactive mitigation action(s) in response to component failure(s), and aim to restore functionality by cold/warm reset of components, activating redundant component(s), de-activating faulty component(s), rewiring(i.e. instructing components to use alternate facet providers) etc. Due to space restrictions, a full list of system mitigation actions is not included here.

Runtime instrumentation of the SLHM strategy captured in the hierarchical state machines requires additional services than those offered by the ACM runtime framework. These additional services include:

1) Instrumentation to communicate the anomalies observed in the components and the local mitigation action by the CLHM.
2) Aggregation of these component level anomalies/ mitigation action to support a system level analysis.
3) System level diagnosis, i.e. identification of the faulty component that is the root cause of the observed anomalies.

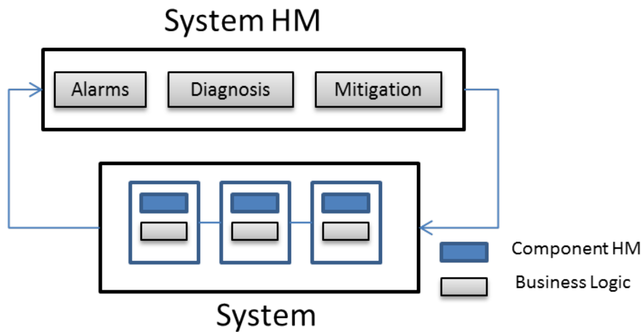


Figure 2. Hierarchical Layout of Component-Level and System-Level Health Managers

4) Execution platform for the SLHM strategy based on the identification of the faulty component.
5) Instrumentation to communicate the mitigation commands to components in the assembly.
6) Instrumenting the components to receive the system level commands and execute them as needed.

Three special, dedicated components (described below) are automatically added to the assembly to implement the System Level Health Manager shown in figure 2. These components are:

- the *Alarm Aggregator* : Responsible for collecting and aggregating anomalous events and the corresponding mitigation actions from the components and reporting these events to the Diagnosis Engine component.
- the *Diagnosis Engine*: Hosts an instance of a diagnosis/ reasoning engine that can isolate the most plausible fault-source *component* based on the information obtained by the Alarm Aggregator.
- the *SystemHM Mitigation Engine* : Receives the diagnosis results: the set of faulty components and responds with an appropriate system-level command(s) to mitigate the fault. It executes the code corresponding to the SLHM strategy captured in the hierarchical state-machine model.

Interconnections of these components to support data-flow (information/ command) in support of SLHM, are automatically synthesized by a code generator that operates on models. The only input required from system integrator is the fault mitigation specification as a hierarchical timed state-machine model.

The following sections delve more into the internals of the SLHM runtime mechanism, the current diagnosis approach adopted in SLHM and the lessons learned in adapting it to a software system.

## III. Diagnosis in SLHM

The diagnosis engine in SLHM is a model-based reasoner that relies on a Timed Failure Propagation Graph (TFPG) [2], [3] model of the entire component assembly. The TFPG-based diagnosis engine implements a real-time incremental reasoning approach that can handle multiple failures including sensor/alarm faults. In addition, the underlying TFPG model can represent a general form of temporal and logical dependency that directly incorporates the dynamics of multi-modal systems.

A **TFPG** is a labeled directed graph where nodes represent either failure modes (i.e. the fault causes) or discrepancies (i.e. the off-nominal conditions that are the effects of failure modes). Edges between nodes in the graph capture the failure propagation effect. While the failure modes are always the root nodes in the graph, the discrepancy nodes always have one or more parent nodes which could be failure mode(s) or other discrepancies. A discrepancy could be of type OR or AND. The discrepancy type determines the conditions that need to be satisfied for the discrepancy (anomaly) to occur. An AND discrepancy (anomaly) could occur only if the failure effect propagated from all of its parent nodes, while an OR discrepancy (anomaly) could occur if the failure propagated from at least one parent node. Further, some discrepancies are observable as the associated anomalies can be detected through a monitor, others are unobservable. To represent failure propagation in multi-modal (switching) systems, edges in the graph can be activated or deactivated based on the current operation mode of the system. The temporal constraints of failure propagation is captured in the edges as a time interval, where the lower and upper bound of the time interval represent the minimum and maximum time for the failure effect to propagate along the edge when it is active.

### A. Creating TFPG Model from a Component Assembly

The TFPG model of the entire system is automatically synthesized from the ACM assembly model. The synthesis follows the component hierarchy, starting with the TFPG model of the component ports, using these port TFPG model to build the component TFPG models which are then used to build the TFPG model of the entire assembly. The well-defined sequence of operations in every component port implementation provides a default failure propagation path across the anomalies that could potentially be observed within a port's operation. This is useful in building a template TFPG model for each component port type. An earlier version of these templates and how they are used to build component and then system level TFPG models was discussed in [12]. Figure 3 shows a portion of the TFPG model of the GPS Assembly captured in figure 1. It shows parts of the TFPG of the Sensor component (and its publisher port: data_out), the GPS component (and its consumer port: data_in) and the failure propagation across these components and their ports.

During the construction of the TFPG model new failure propagation links are added at each stagem, within a

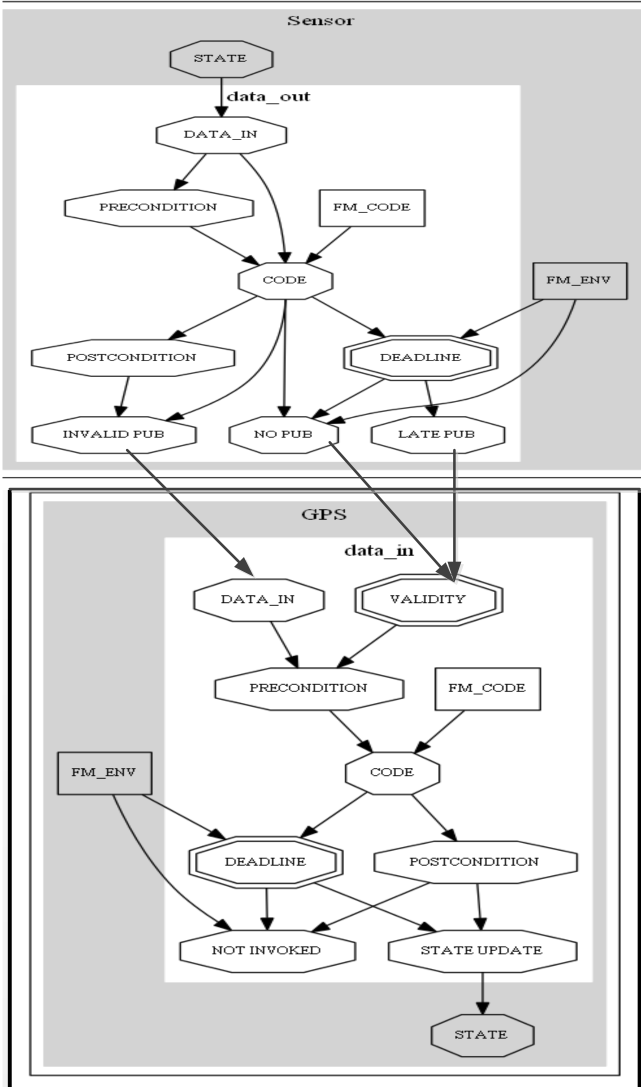


Figure 3. TFPG model for Sensor-Publisher and GPS-Consumer

component TFPG model and within the assembly TFPG model. These additional failure propagation links are based on the failure cascades within a component and across the component boundaries. The properties governing these new failure propagation links are discussed in the next section.

### *B. Failure cascades in ACM Component Assemblies*

The anomalies observed within a component port can be caused by a local component failure (i.e. a latent defect in the component code) or problems from the component's environment (e.g., related to resource sharing) or can be the result of failure effects cascading from other components in the assembly. The discussion below summarizes the contexts that have been considered for failure cascades captured in the TFPG model of the component assembly.

**Correctness contracts and dataflow dependency**: For each component (or component port), the pre- and post-conditions capture the required guarantees on the input data and the provided guarantees on the output data. This relationship across the dataflow in the assembly model leads to an understanding that for the nominal operation of the software assembly, the output/contract guarantees (i.e. post-condition ) of the supplier component port must satisfy the input contract guarantees (i.e. pre-condition) of the receiver component. Stated otherwise, when a system integrator is building a component assembly, care should be taken that the pre-conditions evaluated on the data should be able to accommodate the post-conditions verified on the data. This type of reasoning is critical in achieving modular certification of software components [26].

This implies that if the rules were followed correctly, when a pre-condition violation is detected on the receiver, a post-condition violation in the sender ensures that the fault is propagating along the direction of data flow. In case of the TFPG model presented in Figure 3, a post-condition violation in Sensor component's publisher port (data_out), can result in a failure propagation that ultimately leads to a pre-condition violation in the GPS component's consumer port (data_in).

**Timing Constraint dependency**: Timing constraints are enforced in the model through the real-time properties of the component ports (i.e., periodicity, deadline, and WCET). Timing constraints to detect staleness in data are captured through the data validity properties on the consumer ports [11]. During runtime, it is possible that a deadline violation in one process (component port) can lead to deadline violations in other ports of the same component or in ports sharing the partition. More importantly, if a process (i.e. component port) depends on the completion of a method call to another component, the designer should have taken into account these dependencies. In other words, the TFPG model is required to have a path for timing constraint violations in a direction opposite to the direction of invocation. In the TFPG model presented in Figure 3, a problem with the timing constraint of the Sensor's publisher can manifest as a Deadline Violation anomaly. This can lead to a delayed or omitted publication of data by the publisher, leading to Validity constraint violation in the GPS's consumer.

**Combination of constraints on data and timing**: Violations of the constraints on data and timing can affect each other. For example, a violation on the timing constraint can lead to a poor or lack of update on the data which can then affect the constraints on the data flow. Likewise, a violation on the data constraints, can lead to computational problems that affect the timing properties associated with component ports. For example, in Figure 3, violation of the Validity constraint in the GPS's consumer port can lead to problems in consumer port's code which can lead to a post-condition violation on the consumer port, resulting in a bad state-update of the GPS's state variables by the consumer port.

It should be noted that the problems associated with

timing and data constraint violation manifest because of (i) latent bugs (FM_Code in GPS and Sensor components of the TFPG model in figure 3), (ii) or problems associated with operating environment of the component (FM_Env in the GPS and Sensor components of the TFPG model in figure 3). Propagation of these failures that cause anomalies in other parts of the assembly model is dependent on the kind of interaction pattern being considered, as follows.

**Synchronous Interactions:** These interactions between a required (receptacle) port and a provided (facet) port are affected by failure propagations associated with constraint violations on timing and data. A requires port can supply bad data to a provider port, there by affecting the state in the provider component. Similarly, a bad state in a provider component can propagate to the component hosting the requires interface via the returned data value. In case of a violation of timing constraints, a deadline violation of the provider can lead to a deadline violation in the requires port.

**Asynchronous Interactions:** Unlike synchronous interactions, failure propagation in asynchronous interactions proceeds in only one direction - from the publisher to the consumer. For example, the TFPG model in figure 3 which shows the failure propagation interaction between the Sensor's publisher and the GPS's consumer . While violation of constraints associated with data propagates directly from the publisher to the consumer, the problems associated with timing do not have a direct relationship like the synchronous interaction. However, it is possible that a periodic consumer can be affected with data-validity violations of a stale data if the publisher violates its deadline or fails to publish the data.

**Invocation Interactions**: Ports (or processes) within the same component can be affected by a fault propagation associated with timing constraint violations when the business logic associated with a port invokes another aperiodic publisher port or a required port. In this case, deadline violation propagates backwards along the invocation chain.

## IV. Discussion

In this section, we identify and discuss several issues pertinent to effective diagnosis of distributed software component assembly. We do not focus on the performance or quantitative aspects of the diagnosis problem or the associated health management architecture. Rather we focus on some interesting aspects that crop up while adapting system health management approaches to support software health management, especially as observed in the context of ACM software framework. We also discuss potential strategies (some of which have already been implemented) to account for these problems, thereby improving the software health management architecture.

**A. Effects of Local Mitigation:** The support for local mitigation actions provides a quick local response to an anomaly. However, this can have the effect of creating a modified failure cascade. For example, consider the case in which CLHM receives a pre-condition violation on a publisher, and decides to abort the publisher operation. While it prevented the publication of bad data (that could have potentially violated the contracts), the lack of data published can lead to a problem on the consumer side. Now, if the downstream consumer is periodic, it will get a validity violation because the sampled data has not been updated by the publisher. To account for the modified fault cascades, we added modes based on the CLHM action (in the generated TFPG model) to activate or deactivate certain failure propagation paths. Also, the CLHM actions (along with the anomalies) were reported to SLHM in order to aid in proper diagnosis.

**B. Alarm Timing Issues:** In our architecture all anomalies detected are time stamped using the local module clock. However, unless a reliable and deterministic network such as Time-Triggered Ethernet [15] is being used, it is possible that alarms do not arrive at the SLHM modules in the order of their detection. This can be either due to the varying network latency or task preemption. To ensure the consistency of the diagnosis , the received alarms are aggregated and sorted by detection time in a moving window and supplied to the diagnosis engine (by AlarmAggregator component). The window size is set based on the higher of the two values: the system hyper period, and the worst-case network latency. In our implementation, this window-size or delay has been set to the system hyper period as it was much bigger than the network latency. Further, this also assumes that the schedule generation ensures that a partition associated with each active component port is triggered at least once in every system hyper period. An alternative approach to this problem involves re-computing hypothesis(in the diagnosis engine) to tolerate the delayed alarm reporting. However, this is not yet implemented.

**C. Masking of Fault Effects:** While building a diagnoser that considers the fault cascades (discussed earlier), we must consider the effect of component or component groups (such as voters) that are designed to mask the effect of certain faults, thereby preventing their cascade to downstream components. Since the generic TFPG model automatically synthesized from the assembly model is not aware of this fault-masking behavior of the component, the diagnosis process related to these faults is affected. The diagnoser (on the basis of the incomplete TFPG model) can expect certain downstream alarms to fire. The masking effect will ensure that the alarms do not fire and hence lead to less robust hypothesis and possibly large number of ambiguities. In such cases, it is important to update the generic TFPG model to ignore the alarms associated with these faults whose effects are being masked. Our earlier work presented in [18] shows an assembly that was setup to tolerate up to two failures among a class of components.

**D**. **Intermittent Faults/Alarms:** It is possible that the failure source or the alarms associated with the anomalies are intermittent, i.e. they are observed in one period but not observed in another. This intermittent behavior can be caused by a partial masking effect, or intermittent behavior in the original fault source, or it can be due to the mitigation actions. TFPG as a diagnosis engine has handled this problem in the system health management domain [3]. However, this problem has not been handled in the software health management framework yet.

**E**. **System Hysteresis:** It is possible that the despite the mitigation action taken at the system-level to remove the fault source, the fault cascade remains in the system for a few cycles. Such hysteresis will result in intermittent alarms during this period and should be ignored by the diagnosis engine. Furthermore, it is important that the CLHMs report not only the activation of alarms, but also their deactivation to the SLHM, thereby improving the quality of future diagnosis.

**F**. **Alarms 'near' the Fault Source:** It is possible that certain anomalies in the component assembly are not observable as they do not have an associated alarm. For example, the developer can choose not to specify a pre-condition for a port. In such cases, when a failure propagates through this port, it is not detected. Further, if an anomaly is detected downstream, the hypothesis ambiguity set could be much larger than if the pre-condition had been specified. This is because then the firing or lack of firing of the pre-condition would eliminate potentially many fault causes. Also, when the ambiguity set grows the mitigation action would probably need to be applied to all components or repetitively to each component until normal functionality is restored. Since this is not an efficient approach, system integrators should ensure that all possible monitors that could be specified are accounted for. This will ensure that the alarms are mostly close to the fault source and that the diagnosis process is less ambiguous resulting in faster and effective mitigation.

**G**. **Distributed SLHM:** In case of very large systems, with a large number of components, it will be useful to identify component assemblies that have limited or no interaction and diagnose each independent regions with a different diagnosis engines. This will allow the diagnosis engine to focus on a smaller region and provide a real-time response to the observed fault-effects. In previous work [17] we have presented a distributed TFPG model. This model can be applied to a large system wherein the local reasoners deal with the diagnosis of their almost independent regions while the global reasoner deals with providing an integrated view for the entire assembly.

## V. Related Research

Our approach focuses on latent faults in software systems, follows a component-based architecture, with a model-based development process, and implements all steps in the Collect/Analyze/Decide/Act loop [7].

Conmy et al. presented a framework for certifying Integrated Modular Avionics software applications built on ARINC-653 platforms in [8]. Their main approach was the use of 'safety contracts' to validate the system at design time. Nicholson presented the concept of reconfiguration in integrated modular systems running on operating systems that provide robust spatial and temporal partitioning in [20]. He suggested use of lookup tables, similar to the health monitoring tables used in ARINC-653 system specification, that maps trigger event to a set of system blue-prints providing the mapping functions.

Rohr et al. advocate the use of architectural models for self-management [25]. They suggest the use of a runtime model to reflect the system state and provide reconfiguration functionality. From a development model they generate a causal graph over various possible states of its architectural entities. Garlan et al. [13] and Dashofy et al. [9] have proposed an approach which bases system adaptation on architectural models representing the system as a composition of several components, their interconnections, and properties of interest. They make reconfiguration decisions using rule-based strategies.

While these works have tended to the structural part of the self-managing computing components, some have emphasized the need for behavioral modeling of the components. For example, Zhang et al. described an approach to specify the behavior of adaptable programs in [34]. Their approach is based on separating the adaptation behavior specification from the non-adaptive behavior specification in autonomic computing software. They model the source and target models for the program using state charts and then specify an adaptation model, i.e., the model for the adaptation set connecting the source model to the target model using a variant of Linear Temporal Logic [33].

Williams' research [24] concentrates on model-based autonomy. The paper suggests that emphasis should be on developing techniques to enable the software to recognize that it has failed and to recover from the failure. Their technique lies in the use of a Reactive Model-based Programming Language (RMPL)[31] for specifying both correct and faulty behavior of the software components. They also use high-level control programs [32] for guiding the system to the desirable behaviors.

The work described here is closely related to the larger field of software fault tolerance: principles, methods, techniques, and tools that ensure that a system can survive software defects that manifest themselves at run-time [16], [23]. To the best of our knowledge, this work and similar

work done by our peers [21], [27], [18], [4] comes closest to applying formal system health management techniques, i.e. detection, diagnosis, mitigation for dynamic software fault removal, performed at run-time.

## VI. Conclusion

This paper summarizes the approach we adopted towards augmenting a software component assembly with support for real-time health management. We adapted a diagnosis scheme, used for Systems Health Management for electromechanical systems, and applied it towards diagnosing problems in software assembly, thereby enabling a two-level software health management scheme. The paper documents the lessons learned (from a diagnosis perspective) to improve the quality of software health management.

**Acknowledgments:** This paper is based upon work supported by NASA under award NNX08AY49A. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Aeronautics and Space Administration. The authors would like to thank Paul Miner, Eric Cooper, and Suzette Person of NASA Langley Research Center for their help and guidance on the project.

## References

[1] Arinc specification 653-2: Avionics application software standard interface part 1 - required services. Technical report.

[2] S. Abdelwahed and G. Karsai. Notions of diagnosability for timed failure propagation graphs. In *Proc. IEEE Systems Readiness Technology Conference*, pages 643–648, 18–21 Sept. 2006.

[3] S. Abdelwahed, G. Karsai, N. Mahadevan, and S. C. Ofsthun. Practical considerations in systems diagnosis using timed failure propagation graph models. *Instrumentation and Measurement, IEEE Transactions on*, 58(2):240–247, February 2009.

[4] M. Barry. http://www.kestreltechnology.com/downloads/FailsafeOverview.pdf, 2008.

[5] A. T. S. Bureau. In-flight upset; 240km NW Perth, WA; Boeing Co 777-200, 9M-MRG. Technical report, August 2005.

[6] A. T. S. Bureau. AO-2008-070: In-flight upset, 154 km west of Learmonth, WA, 7 October 2008, VH-QPA, Airbus A330-303. Technical report, October 2008.

[7] e. Cheng, Betty H. Software engineering for self-adaptive systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[8] P. Conmy, J. McDermid, and M. Nicholson. Safety analysis and certification of open distributed systems. In *International System Safety Conference,*, Denver, 2002.

[9] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 21–26, New York, NY, USA, 2002. ACM Press.

[10] R. de Lemos. Analysing failure behaviours in component interaction. *Journal of Systems and Software*, 71(1-2):97 – 115, 2004.

[11] A. Dubey, G. Karsai, and N. Mahadevan. A component model for hard real-time systems: Ccm with arinc-653. *Software: Practice and Experience*, 41(12):1517–1550, 2011.

[12] A. Dubey, G. Karsai, and N. Mahadevan. Model-based Software Health Management for Real-Time Systems. In *Aerospace Conference, 2011 IEEE*, pages 1–18. IEEE, 2011.

[13] D. Garlan, S. W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. *Architecting Dependable Systems*, 2003.

[14] W. S. Greenwell, J. Knight, and J. C. Knight. What should aviation safety incidents teach us? In *SAFECOMP 2003, The 22nd International Conference on Computer Safety, Reliability and Security*, 2003.

[15] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[16] M. R. Lyu. Software reliability engineering: A roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 153–170, Washington, DC, USA, 2007. IEEE Computer Society.

[17] N. Mahadevan, S. Abdelwahed, A. Dubey, and G. Karsai. Distributed diagnosis of complex causal systems using timed failure propagation graph models. In *IEEE Systems Readiness Technology Conference, AUTOTESTCON*, 2010.

[18] N. Mahadevan, A. Dubey, and G. Karsai. Application of software health management techniques. In *Proceedings of the 2011 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, New York, NY, USA, 2011. ACM, ACM.

[19] NASA. Report on the loss of the mars polar lander and deep space 2 missions. Technical report, NASA, 2000.

[20] M. Nicholson. Health monitoring for reconfigurable integrated control systems. *Constituents of Modern System safety Thinking. Proceedings of the Thirteenth Safety-critical Systems Symposium.*, 5:149–162, 2007.

[21] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A hard real-time runtime monitor. In *Runtime Verification*, pages 345–359. Springer, 2010.

[22] J. Potocti de Montalk. Computer software in civil aircraft. In *Digital Avionics Systems Conference, 1991. Proceedings., IEEE/AIAA 10th*, pages 324 –330, oct 1991.

[23] L. L. Pullum. *Software fault tolerance techniques and implementation*. Artech House, Inc., Norwood, MA, USA, 2001.

[24] P. Robertson and B. Williams. Automatic recovery from software failure. *Commun. ACM*, 49(3):41–47, 2006.

[25] M. Rohr, M. Boskovic, S. Giesecke, and W. Hasselbring. Model-driven development of self-managing software systems. In *Proceedings of the Workshop "Models@run.time" at the 9th International Conference on model Driven Engineering Languages and Systems (MoDELS/UML'06)*, 2006.

[26] J. Rushby. Modular certification. Technical report, Sept. 2001.

[27] J. Schumann, A. Srivastava, and O. Mengshoel. Who guards the guardians?toward v&v of health management software. In *Runtime Verification*, pages 399–404. Springer, 2010.

[28] A. Srivastava and J. Schumann. The Case for Software Health Management. In *Fourth IEEE International Conference on Space Mission Challenges for Information Technology, 2011. SMC-IT 2011.*, pages 3–9, August 2011.

[29] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electron. Notes Theor. Comput. Sci.*, 141(3):53–71, 2005.

[30] N. Wang, D. C. Schmidt, and C. O'Ryan. Overview of the CORBA component model. *Component-based software engineering: putting the pieces together*, pages 557–571, 2001.

[31] B. Williams, B. Williams, M. Ingham, S. Chung, and P. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[32] B. C. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. T. Sullivan. Model-based programming of fault-aware systems. *AI Magazine*, 24(4):61–75, 2004.

[33] J. Zhang and B. H. C. Cheng. Specifying adaptation semantics. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM.

[34] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.