Model-based Tools and Techniques for Real-Time System and Software Health Management

Sherif Abdelwahed*, Abhishek Dubey, Gabor Karsai, and Nag Mahadevan

Institute for Software Integrated Systems, Vanderbilt University

Abstract

System-level detection, diagnosis, and mitigation of faults in complex systems that include physical components as well as software are essential to achieve high dependability. The paper introduces a model, referred to as the Timed Failure Propagation Graph (TFPG) that captures the causal propagation of observable fault effects in systems. Several algorithms based on this model have been developed, including: consistency-based centralized and distributed algorithms for multiple-fault fault source isolation in real-time, algorithms to calculate diagnosability metrics, and algorithms to prognosticate impending failures. The model and the associated algorithms are applicable to physical systems but recently they have been applied to component-based software systems as well, where similar fault propagation can take place. The paper describes the modeling paradigm, the algorithms developed, and how they were applied in a system and software context.

1 Introduction

The ultimate challenge in system health management is the theory for and application of the technology to *systems*, for instance to an entire vehicle. The main problem the designer faces is complexity; simply the sheer size of the system, the number of data points, anomalies, and failure modes, can be overwhelming. Furthermore, systems are heterogeneous and one has to have a systems engineer's view to understand interactions among systems. Yet, system-level health management is crucial as faults increasingly arise from system-level effects and interactions. While individual subsystems tend to have built-in redundancy or local anomaly detection, fault management, and prognostics features, the system integrators are required to provide the same capabilities for the entire vehicle, across different engineering subsystems and areas.

The heterogeneity of subsystems necessitates the use of techniques that are domain-independent and applicable in mechanical and hydraulic systems, as well

* Sherif Abdelwahed is with Virginia Commonwealth University

as to electronics, and even in software. The most common, high-level concept one can find across these domains is the concepts of (1) observable degradations, anomalies, discrepancies caused by failure modes, (2) their propagation, and (3) their temporal evolution towards system-level faults. Such propagation and evolution effects are clearly cross-domain and cross-subsystem, and, if they are understood by the designer, they can be used in system level diagnostics and prognostics.

In this paper we present a model-based technology for system-level health management based on the concept of Timed Fault Propagation Graphs (TFPG, for short). We present the modeling language, fault diagnostics and prognostics algorithms that are applicable to complex systems, and how these techniques can be used for the health management of software systems.

We argue that the reasoning approach described here can be called as 'real-time' in two ways. First, it can operate in real-time, on-line and is capable of producing the new fault diagnostics results as soon as new evidence becomes available (or as time elapses). This will be elaborated in the section on the reasoner. Second, the operation of the reasoner is bounded in space (memory footprint) and time (response time). The bounds depend on the complexity and structure of the particular model used. Note that we use a model-based reasoner, hence its operation and behavior is ultimately determined by the model used. While we cannot at this time analytically estimate the worst-case response time of the reasoner simply from the model, we can predict it by using the specific model in the reasoner and applying it to a 'worst-case' input sequence of alarms.

The model-based approach described in this paper could be the next logical step after applying machine learning and/or data mining algorithms to data collected during operations. What the data mining techniques discover can be used in the following ways. One, the data mining can provide assistance to the modeler who constructs the TFPG models: correlations discovered in the data may serve as clues to building the causal models. Second, the data mining results can be used to build the discrete monitors (fault detectors) needed by the TFPG reasoner in operation. These monitors detect the anomalies in the system and will trigger the higher-level TFPG reasoner. Third, the data mining can play an important role in refining and improving the models. We will discuss this last point in a separate section.

The paper is organized as follows. First we review system-level fault diagnostics techniques. Next we describe the TFPG model, the reasoning algorithms used and present examples. This is followed by a section about the application of the TFPG in software health management. Then we describe the application of TFPG in system prognosis. Finally, we conclude with a discussion on how data mining techniques can be used to improve a TFPG-based system and some observations for future research. We also included a number of examples, with specific test results. The tests were conducted on an Intel Xeon (W5320) 2.67 GHz Processor with 4GB of RAM.

2 Related Work

2.1 Failure Propagation Models

Diagnostic reasoning techniques share a common process in which the system is continuously monitored and the observed behavior is compared with the expected one to detect abnormal conditions. In many industrial systems, diagnosis is limited to signal monitoring and fault identification via threshold logic, e.g., detecting if a sensor reading deviates from its nominal value. Failure propagation is modeled by capturing the qualitative association between sensor signals in the system for a number of different fault scenarios. Typically, such associations correspond to relations used by human experts in detecting and isolating faults. This approach has been effectively used for many complex engineering systems. Common industrial diagnosis methods include fault trees [23, 51, 22, 24], cause-consequence diagrams [41, 42], diagnosis dictionaries [44], and expert systems [48, 50].

Model-based diagnosis (see [17, 21, 38] and the references therein), on the other hand, compares observations from the real system with the predictions from a model. Analytical models, such as state equations [39], finite state machines [46], hidden Markov models [49], and predicate/temporal logic [43] are used to describe the nominal system behavior. In the case of a fault, discrepancies between the observed behavior and the predicted normal behavior occur. These discrepancies can then be used to detect, isolate, and identify the fault depending on the type of model and methods used. In consistency-based diagnosis the behavior of the system is predicted using a nominal system model and then compared with observations of the actual behavior of the system to obtain the minimal set of faulty component that is consistent with the observations and the nominal model. Consistency-based diagnosis was introduced in a logical framework in [43] and was later extended in [13]. The approach has been applied to develop diagnosis algorithms for causal systems [11, 12] and temporal causal systems [18, 10].

The diagnosis approach presented in this paper is conceptually related to the temporal causal network approach presented in [10]. However, we focus on incremental reasoning and diagnosis robustness with respect to sensor failures. The causal model presented in this paper is based on the timed failure propagation graph (TFPG) introduced in [29, 30]. The TFPG model is closely related to fault models presented in [37, 27, 32] and used for an integrated fault diagnosis and process control system [26]. The TFPG model was extended in [3] to include mode dependency constraints on the propagation links, which can then be used to handle failure scenarios in hybrid and switching systems. TFPG modeling and reasoning tool has been developed and used successfully in an integrated fault diagnoses and process control system [26].

The temporal aspects of the TFPG model are closely related to the domain theoretic notion of temporal dependency proposed in [6]. However, there are several major differences between the two approaches. In particular, TFPG-based diagnosis implements a real-time incremental reasoning approach that can han-

dle 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-mode systems.

2.2 Fault Recovery

The fault recovery aspect of this work has been inspired by our previous work on reflex and healing architecture. Reflex and Healing (RH) [34, ?] is a biologically inspired two stage mechanism for recovering from faults in large distributed real-time systems. The first stage is composed of primary building blocks of fault management components called reflex engines (also referred to as managers) that are arranged in a hierarchical management structure. A reflex engine integrates several fault management devices expressed as timed state machine models. While some of these state machines are used as observers that generate 'fault-events' (discussed in next section), others are used as mitigators to perform time-bounded reactive actions upon occurrence of certain fault-events.

At the system-level, fault-recovery is done using a planned reconfiguration of the system. This involves a planning step that searches over a set of candidate models obtained by using a set of pre-specified goals and a list of constraints [47]. This step is typically multi-objective in nature and is dependent on several factors such as system goals, resilience to future faults and performance. This architecture has been successfully demonstrated for a large-scale software infrastructure for a particle accelerator [16], and computing cluster infrastructure used for scientific workflows [40].

2.3 Fault Detection and Health Management of Software

Conmy et al. presented a framework for certifying Integrated Modular Avionics software applications build on ARINC-653 platforms in [9]. Their main approach was the use of 'safety contracts' to validate the system at design time. They defined the relationship between two or more components within a safety critical system. However, they did not present any details on the nature of these contracts and how they can be specified. We believe that a similar approach can be taken to formulate acceptance criteria, in terms of "correct" value-domain and temporal-domain properties that will let us detect any deviation in a software component's behavior.

Nicholson presented the concept of reconfiguration in integrated modular systems running on operating systems that provide robust spatial and temporal partitioning in [35]. He identified that health monitoring is critical for a safety-critical software system and that in the future it will be necessary to trade-off redundancy based fault tolerance for the ability of "reconfiguration on failure" while still operational. He described that a possibility for achieving this goal is to use a set 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. Furthermore, he identified that this kind of reconfiguration is more amenable to failures that happen gradually, indicated by

parameter deviations. Finally, he identified the challenge of validating systems that can reconfigure at run-time.

Goldberg and Horvath have discussed discrepancy monitoring in the context of ARINC-653 health-management architecture in [19]. They describe extensions to the application executive component, software instrumentation and a temporal logic run-time framework. Their method primarily depends on modeling the expected timed behavior of a process, a partition, or a core module - the different levels of fault-protection layers. All behavior models contain "faulty states" which represent the violation of an expected property. They associate mitigation functions using callbacks with each fault.

Sammapun et al. describe a run-time verification approach for properties written in a timed variant of LTL called MEDL in [45]. They described an architecture called RT-MaC for checking the properties of a target program during run-time. All properties are evaluated based on a sequence of observations made on a "target program". To make these observations all target programs are modified to include a "filter" that generates the interesting event and reports values to the event recognizer. The event recognizer is a module that forwards the events to a checker that can check the property. Timing properties are checked by using watchdog timers on the machines executing the target program. The main difference in this approach and the approach of Goldberg and Horvath outlined in previous paragraph is that RT-MaC supports an "until" operator that allows specification of a time bound where a given property must hold. Both of these efforts provided valuable input to our design of run-time component level health management.

3 Fault Diagnostics Using Timed Failure Propagation Graphs

3.1 The Timed Failure Propagation Graph Model

A TFPG is a labeled directed graph where nodes represent either failure modes, which are fault causes, or discrepancies, which are off-nominal conditions that are the effects of failure modes. Edges between nodes in the graph capture the effect of failure propagation over time in the underlying dynamic system. To represent failure propagation in multi-mode (switching) systems, edges in the graph model can be activated or deactivated depending on a set of possible operation modes of the system. Formally, a TFPG is represented as a tuple $(F, D, E, M, \mathsf{ET}, \mathsf{EM}, \mathsf{DC})$, where:

- F is a nonempty set of failure nodes.
- D is a nonempty set of discrepancy nodes.
- $E \subseteq V \times V$ is a set of edges connecting the set of all nodes $V = F \cup D$.
- M is a nonempty set of system modes. At each time instance t the system can be in only one mode.
- $\mathsf{ET} : E \to I$ is a map that associates every edge in E with a time interval $[t_1, t_2] \in I$.

– $\mathsf{EM} : E \to \mathcal{P}(M)$ is a map that associates every edge in E with a set of modes in M. We assume that $\mathsf{EM}(e) \neq \varnothing$ for any edge $e \in E$.
– $\mathsf{DC} : D \to \{\mathtt{AND}, \mathtt{OR}\}$ is a map defining the class of each discrepancy as either AND or an OR node.
– $\mathsf{DS} : D \to \{\mathtt{A}, \mathtt{I}\}$ is a map defining the monitoring status of the discrepancy as either A for the case when the discrepancy is active (monitored by an online alarm) or I for the case when the discrepancy is inactive (not monitored)[1].

In the above model, the map ET associates each edge $e \in E$ with the minimum and maximum time for the failure to propagate along the edge. For an edge $e \in E$, we will use the notation e.tmin and e.tmax to indicate the corresponding minimum and maximum time for failure propagation along e, respectively. That is, given that a propagation edge is enabled (active), it will take at least (most) tmin (tmax) time for the fault to propagate from the source node to the destination node. The map EM associates each edge $e \in E$ with a subset of the system modes at which the failure can propagate along the edge. Consequently, the propagation link e is enabled (active) in a mode $m \in M$ if and only if $m \in \mathsf{EM}(e)$. The map DC defines the type of a given discrepancy as either AND or OR. An OR type discrepancy node will be activated when the failure propagates to the node from any of its parents. On the other hand, an AND discrepancy node can only be activated if the failure propagates to the node from all its parents. We assume that TFPG models do not contain self-loops and that failure modes are always root nodes, i.e., they cannot be a destination of an edge. Also, a discrepancy cannot be a root node, that is, every discrepancy must be a successor of another discrepancy or a failure mode.

Figure 1 shows a graphical depiction of a failure propagation graph model. Rectangles in the graph model represent the failure modes while circles and squares represent OR and AND type discrepancies, respectively. The arrows between the nodes represent failure propagation. Propagation edges are parameterized with the corresponding interval, $[e.\text{tmin}, e.\text{tmax}]$, and the set of modes at which the edge is active. Figure 1 also shows a sequence of active discrepancies (alarm signals) identified by shaded discrepancies. The time at which the alarm is observed is shown above the corresponding discrepancy. Dashed lines are used to distinguish inactive propagation links.

The TFPG model captures observable failure propagations between discrepancies in dynamic systems. In this model, alarms capture state deviations from nominal values. The set of all observed deviations corresponds to the monitored discrepancy set in the TFPG model. Propagation edges, on the other hand, correspond to causality (for example, as defined by energy flow) in the system dynamics. Due to the dynamic nature of the system, failure effects take time to propagate between the system components. Such time in general depends on the system's time constants as well as the size and timing of underlying failure. Propagation delay intervals can be computed analytically or through simulation of an accurate physical model.

[1] In this paper we will use the terms alarms and monitored discrepancies interchangeably as they mean the same thing.

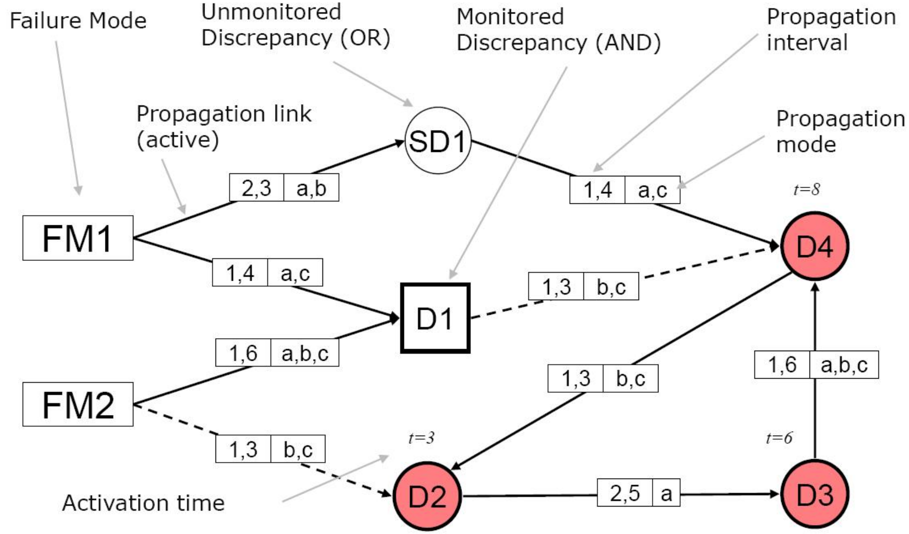

Fig. 1. TFPG model ($t = 10$, Mode=A $\forall t \in [0, 10]$.

Failure propagation in a TFPG system has the following semantics. The state of a node indicates whether the failure effects reached this node. For an `OR` type node v' and an edge $e = (v, v') \in E$, once a failure effect reaches v at time t it will reach v' at a time t' where $e.\text{tmin} \leq t' - t \leq e.\text{tmax}$. On the other hand, the activation period of an `AND` alarm v' is the composition of the activation periods for each link $(v, v') \in E$. For a failure to propagate through an edge $e = (v, v')$, the edge should be active throughout the propagation, that is, from the time the failure reaches v to the time it reaches v'. An edge e is active if and only if the current operation mode of the system, m_c is in the set of activation modes of the edge, that is, $m_c \in \mathsf{EM}(e)$. When a failure propagates to a monitored node v' ($\mathsf{DS}(v') = \mathtt{A}$) its physical state is considered to be `ON`, otherwise it is considered to be `OFF`. If the link is deactivated any time during the propagation (because of mode switching), the propagation stops. Links are assumed to be memory less with respect to failure propagation so that current failure propagation is independent of any (incomplete) previous propagation. Also, once a failure effect reaches a node, its state will change permanently and will not be affected by any future failure propagation.

3.2 Reasoning algorithm

The reasoning algorithm for TFPG model diagnosis is based on a consistency relationship defined using three state mappings for the graph nodes of the TFPG model: physical, observed, and hypothetical.

A *physical system* state corresponds to the current state of all nodes in the TFPG model. At any time t the physical state is given by a map $AS_t : V \rightarrow$

$\{\texttt{ON}, \texttt{OFF}\} \times \mathbb{R}$, where V is the set of nodes in the TFPG model. An `ON` state for a node indicates that the failure (effect) reached this node, otherwise it is set to `OFF`. The physical state at time t is denoted $AS_t(v)$.state, while $AS_t(v)$.time denote the last time at which the state of v is changed. Failure effects are assumed permanent, therefore, the state of a node once changed will remain constant after that. A similar map is used to define the state of edges based on the current mode of the system.

The *observed state* at time t is defined as a map $S_t : D \rightarrow \{\texttt{ON}, \texttt{OFF}\} \times \mathbb{R}$. Clearly, observed states are only defined for discrepancies. The observed state of the system may not be consistent with the failure propagation graph model temporal constraints, due to potential alarm failures. However, we assume that monitored discrepancy signals are permanent so that once the observed state of a discrepancy has changed, it will remain constant after that.

The aim of the diagnosis reasoning process is to find a consistent and plausible explanation of the current system state based on the observed state. Such explanation is given in the form of a valid hypothetical state. A *hypothetical state* is a map that defines node states and the interval at which each node changes its state. Formally a hypothetical state at time t is a map $H_t^{V'} : V' \rightarrow \{\texttt{ON}, \texttt{OFF}\} \times \mathbb{R} \times \mathbb{R}$ where $V' \subseteq V$. Similar to actual states, hypothetical states are defined for both discrepancies and failure modes. The estimated earliest (latest) time of the state change is denoted $H(v)$.terl ($H(v)$.tlat).

A hypothetical state is an estimation of the current state of all nodes in the system and the time period at which each node changed its states. Estimation of the current state is valid only if it is consistent with the TFPG model. State consistency in TFPG models is a node-parents relationship that can be extended pairwise to arbitrary subsets of nodes. Formally, let $\text{Pr}(v)$ denotes the set of parents of v in a TFPG model G. We can define *observable consistency* at time t as a relation $\text{OCons}_t \subset \mathcal{P}(V) \times V$ such that $(V', v) \in \text{OCons}$ if and only if $V' = \text{Pr}(v)$ and the observable state of v is consistent with that of all its parents V' based on the map S_t and the failure propagation semantics. The observable state consistency relationship can be directly extended to any set of nodes representing a subgraph of G. In this case we overload the relationship OCons so that $\text{OCons}_t \subseteq \mathcal{P}(V)$, where for each $V' \subseteq V$:

$$V' \in \text{OCons}_t \;\; \Leftrightarrow \;\; \forall v \in V' \; (\text{Pr}_{V'}(v), v) \in \text{OCons}$$

where $\text{Pr}_{V'}(v)$ is the parents of v restricted to V'. The set of maximally consistent set of nodes is denoted Φ_t where $V' \in \Phi_t$ if and only if

$$V' \in \text{OCons}_t \text{ and } \forall V'' \subseteq V \; V' \subset V'' \Rightarrow V'' \notin \text{OCons}_t$$

The set Φ_t can be efficiently computed incrementally based on Φ_{t-1} based on a new event e_t. The event e_t corresponds to either a new triggered monitored discrepancy or a time-out event generated when a sensor alarm is not observed at state `ON` while it is supposed to be based on its current hypothetical state. The underlying procedure will be denoted $\textsf{UpdateMCO}(\Phi_{t-1}, e_t)$. Note that initially $\Phi_0 = \{V\}$.

Algorithm 1 The diagnosis procedure $\mathsf{Diag}(\Phi_{t-1}, e_t)$

$\Phi_t \leftarrow \mathsf{UpdateMCO}(\Phi_{t-1}, e_t)$
$HS_t \leftarrow \varnothing$
define
$\mathsf{In}(X) := \{v \in X | (\forall v' \in X)\ (v, v') \notin E\}$
$\mathsf{PSet}(X) := \{v \in V - X | (\exists v' \in \mathsf{In}(X))\ (v, v') \in E\}$
$\mathsf{ODC}(X) := \cup_{v \in X} \mathsf{Reach}(v) - X$
$\mathsf{TSet}(X) := \{v \in V - X | \mathsf{ODC}(X) \times v \cap E = \varnothing\}$
$\mathsf{CSet}(X) := \{v \in \mathsf{TSet}(X) | (\exists v' \in X)\ (v', v) \in E\}$
end define
for all $V' \in \Phi_t$ **do**
  $H \leftarrow S_t|_{V'}$
  while $\mathsf{PSet}(V') \neq \varnothing$ **do**
    select v from $\mathsf{PSet}(V')$
    $H \leftarrow \mathsf{BProp}(H, v)$
    $V' \leftarrow V' \cup \{v\}$
  end while
  while $\mathsf{CSet}(V') \neq \varnothing$ **do**
    select v from $\mathsf{CSet}(V')$
    $H \leftarrow \mathsf{FProp}(H, v)$
    $V' \leftarrow V' \cup \{v\}$
  end while
  for all $v \in V - V'$ **do**
    $H(v)$.state $\leftarrow$ `OFF`
    $H(v)$.terl, $H(v)$.terl $\leftarrow 0$
  end for
  $HS_t \leftarrow HS_t \cup \{H\}$
end for
return Φ_t, HS_t

Based on the semantics of failure propagation it is possible to define a constructive notion of *hypothetical consistency* such that given a consistent hypothetical state $H_t^{V'}$ it is possible to extend this map forward (procedure $\mathsf{BProp}(H_t^{V'}, v)$)by assigning the maximal hypothetical state of the node v based on the hypothetical state of its parents in V', or backward (operation $\mathsf{FProp}(H_t^{V'}, v)$) by assigning the maximal hypothetical state for v' based on the state of its children in V'. The following algorithm outlines the incremental reasoning procedure.

The above diagnosis algorithm returns a set of new hypotheses that can consistently explain the current observed state of the TFPG system. A failure report is then generated from the computed set of hypotheses HS_t. The failure report enlists the set of all consistent state assignments that maximally matches the current set of observations. Any observed state that does not match the current hypothesis is considered faulty. A detailed description and analysis of the diagnosis algorithm can be found in [4].

Hypothesis Ranking The quality of the generated hypotheses is measured based on three independent factors:

- *Plausibility* is a measure of the degree to which a given hypothesis group explains the current fault signature. Plausibility is typically used as the first metric for sorting the hypotheses, focusing the search on the failure modes that explain the data that is currently being observed.
- *Robustness* is a measure of the degree to which a given hypothesis is expected to remain constant. Robustness is typically used as the second metric for sorting the hypotheses, helping to determine when to take action to repair the system.
- *Failure Rate* is a measure of how often a particular failure mode will occur.

The plausibility metric considers two independent factors, namely, alarm consistency and failure mode parsimony. The alarm consistency factor is defined as the ratio of the active consistent alarms to that of all (currently) identified alarms. The failure mode factor is defined as the ratio of identified failure modes (according to the underlying hypothesis) to the total number of failure modes in the system. This factor is a direct representation of the parsimony principle (hypothesis with fewer failures is more plausible). Hypotheses plausibility metrics are ordered lexicographically (alarm factor is more dominant).

The diagnoser selects the current set of hypothesis incrementally in an attempt to improve the current plausibility measure. In other words, the diagnoser will update a given hypothetical state map only if such an update can increase the plausibility of the underlying hypothesis. In addition, changes are restricted so that the updated hypothesis remains valid.

Reasoner Performance The reasoning algorithm described above is an online algorithm as it continually updates the active hypothesis set as new evidence (alarms and mode changes) becomes available. Furthermore, it can be triggered based on the elapse of time (with no changes in the alarms) and it will revise the hypotheses as needed. This latter capability is made possible by the propagation time interval on the edges: if propagation does not happen, it may facilitate the preference of one failure mode over another in the hypothesis. From a strictly pragmatic point of view, the two most important metrics of reasoner performance are the worst-case response time and the dynamic memory footprint used. We argue that for a particular model (with a given topology, failure modes, alarms, etc.) these two metrics are always bounded.

The dynamic memory footprint is based on the number and the size of the hypotheses the reasoner has to manage during an update. This worst-case can be induced by an alarm input sequence that activates alarms in the reverse, that is alarms farthest from the failure modes are activated first, then the alarms before them, etc. As the number of alarms is bounded this process results in a bounded use of dynamic memory. In fact, we have created a tool that predicts the dynamic memory footprint of the reasoner based on this technique and simulating its memory usage patterns.

The worst-case response time of the reasoner depends on the size and complexity of the graph and the size of the active hypothesis set. As both of them are bounded, the worst-case response time is bounded as well. Similarly to the previous case, one can predict the worst-case response time based on worst-case scenarios, on the actual reasoner implementation.

Note that the worst-case response time of the reasoner might be much worse than the average or typical response time. It is an open research question how to analytically estimate the average-case response time. In our experience, presented below, we found that for realistic models and realistic scenarios the reasoner performs well.

3.3 TFPG Examples

This section describes example of TFPG models and the results of the experiment performed using these models. Each test case involved loading the appropriate TFPG model into the centralized TFPG Reasoner, feeding the reasoner with a timed sequence of events. Each test case typically contained at least 10 events or more - depending on the nature of the test case. These test cases included cases with alarms firing in the correct sequence as per the graph (for single and multiple simultaneous failures), false alarms, intermittent alarms, mode changes. The tests were conducted on an Intel Xeon (W5320) 2.67 GHz Processor with 4MB RAM.

Figure 2 shows a trivial non-hierarchical TFPG model. It shows the root causes of the failure (Failure-Modes FM1, FM2) and the anomalies (Discrepancies RD1, D1, SD12, D12, RD2, D2) that would be triggered when one or more of these failures were to occur. All observable discrepancies have an alarm associated with them (e.g. alarm MRD1 for RD1, M1 for D1 etc.). The links capture the failure propagation starting from the Failure Modes to Discrepancies and to subsequent Discrepancies downstream. Some of these links depict additional constraints related to activation and timing for failure propagation. The activation condition (a Boolean expression over the modes) captures when failure can propagate over a link. The timing constraint expresses the time bounds within which the failure effect is expected to propagate over that link. These constraints are not specified when the failure can propagate over the link at any time or in any mode. While this model is flat, it can be made hierarchical by composing a TFPG model (at any level) from a TFPG model of components. Overall, this model has 2 Failure Modes, 6 Discrepancies (5 Observable),7 failure propagation links, 2 Modes (with 2 states each).

Next two examples describe typical TFPG models of subsystems in real applications. However, visual rendering of these models is not possible because of their complexity and the space limitations.

Example 1. Pump and Valve This example refers to a Pump and Valve (PV) subsystem. Table 1 presents a summary. It captures a hierarchical TFPG model of the faults, anomalies and failure effect propagation in a simple example consisting of a Pump, a Valve, 2 sensors, and the associated interface hardware. The

Table 1. TFPG Diagnosis Experiments

			Model Parameters											
		Example	#C	#FM	#D	#FP	#A	#M	#IP	#OP	#AG	Mem	Avg. Update Time (sec)	#TC
Centralized	1	Pump & Valve	11	36	120	174	27	3	NA	NA	29	18.5	0.0128	87
	2	Generic Fuel Transfer System	153	481	1973	3409	270	9	NA	NA	244	275	0.21	550
Distributed	3	Global	0	0	2				1	9	9	30	0.11 (0.06)	110
		AREA_1	17	34	11	43	11	0	0	9	11	7.5	0.012	
		AREA_2	50	153	62	590	62	0	1	0	87	25.4	0.08	
	4	Global	21	52	92	176	18	3	28	18	38	74.5	0.3 (0.18)	223
		AREA_1/ AREA_2	39	131	268	539	33	3	3	2	54	47.5	0.14	
		AREA_3/ AREA_4	34	87	119	266	13	0	11	7	24	24.5	0.096	
Software Diagnosis (Centralized)	5	GPS System	1	19	78	172	9	22			12	12	0.063	10

#C - Number of Components, #FM - Number of Failure Modes, #D - Number of Discrepancies, #FP - Number of Failure Propagations, #A - Number of Alarms, #M - Number of Modes, #IP - Number of Input Ports, #OP - Number of Output Ports, #AG - Number of Ambiguity Groups, #TC - Number of Test Cases, Mem - Worst case memory used (MB).

For the global reasoners the time noted in parentheses is the actual time spent in reasoning for creating and updating global hypotheses. Additional time is spent in the global reasoner for assimilating the information from local and global hypotheses to create the set of system-wide consistent failure hypotheses. See Section 3.5 for further details.

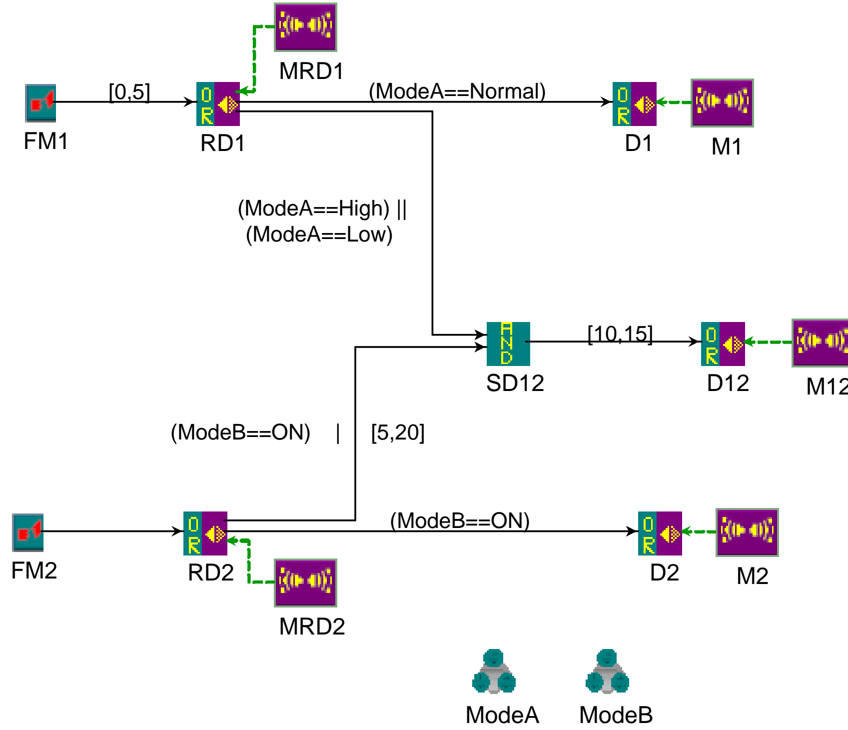

Fig. 2. An Illustrative TFPG Example

operation (and hence the failure propagation) of the PV subsystem is governed by 3 Modes - associated with Pump (On/ Off), Valve(Open/ Close) and the subsystem(Active/ Standby/ Shutdown). Apart from capturing the failure modes and their propagation in the pump, valve and their sensors, it also captures the failure effects originating from the power distribution and remote interface units associated with these devices. The failure propagation in the system depends on the operating of the system and the control actions (modes) issued to the devices (pump, valve). Further offline analysis of the model revealed that give the causal linkage expressed between the failures and the observed discrepancies in the TFPG model, the number of ambiguity groups in terms of failure modes was 29. The average processing time to update the result was 0.0128 sec. The estimate of worst case memory usage is around 18.5 MB.

Example 2. Generic Fuel Transfer System Another example, more comparable to a real-life application, is a hierarchical TFPG model for an aircraft fuel transfer system. The subsystem is symmetrically divided into left and right parts with supply tanks (Left and Right Fuselage Tanks, Left and Right Wing Tanks)

feeding to a central transfer-manifold which then feeds to engine-feed tanks (Left and Right Feed Tanks). The supply tanks and the feed tanks are full initially. The controller tries to maintain a constant supply to the engines while maintaining the center of gravity of the aircraft. The hierarchical TFPG model of this subsystem includes failure propagation across the main subsystem elements (the tanks and the manifold) as well as the power and control elements. The properties of this TFPG model are captured in Table 1 Offline analysis revealed that there were a total of 244 failure ambiguity groups. The average processing time for an update was 0.22 sec. The estimate of worst case memory usage is around 275 MB.

3.4 Distributed reasoning

A centralized reasoning approach (TFPG based or otherwise) is not well-suited for the on-line diagnosis of very large systems that are made up of many subsystems with limited failure interactions across the sub-system boundary. In such cases, a single centralized diagnosis engine might not scale to provide the desired response time. In a centralized system, all alarms and mode changes have to be routed to the central reasoner that would have to operate large models. It might be more pragmatic to use multiple reasoners to perform diagnosis of different areas / sub-systems. These area reasoners can be hosted on the same or different processing nodes and can respond faster than a single system-wide reasoner as their search spaces are much smaller and they can operate in parallel. However this approach of splitting the task among multiple independent area reasoners does not address cascading fault-effects from one sub-system to another. Additionally, it cannot provide a coherent system-wide knowledge of the fault-status of the entire system. This section discusses a TFPG-based distributed reasoning approach that handles these problems.

Overview. The TFPG-based distributed reasoning approach (discussed later in this section) employs a *global* TFPG reasoner and multiple *local* TFPG reasoners. The TFPG-model of the global reasoner captures only the failure interactions between sub-systems monitored by the local reasoner(s). The TFPG-model of the local reasoner contains a detailed TFPG model of the sub-system monitored by the local reasoner(s). The local reasoners operate in parallel and autonomously, each of them reasoning over the events in its specific sub-system. The local reasoner(s) communicate updates of any potential fault-cascade to the global reasoner. The global reasoner, on its part, ensures that this information is transferred only to relevant local reasoners whose sub-system could be potentially affected by the fault cascade. Using the knowledge obtained from the local reasoner of the potential fault-cascades and their individual diagnosis results the global reasoner builds a globally consistent system-wide diagnosis report. This approach ensures that there is a relatively quick diagnosis result from the local reasoners which, if required, could be refined or updated by the global reasoner that yields system-wide consistent result.

The figure 3 captures a model of the local reasoners and the global reasoner. Note that the global reasoner does not have a detailed view of the subsystems.

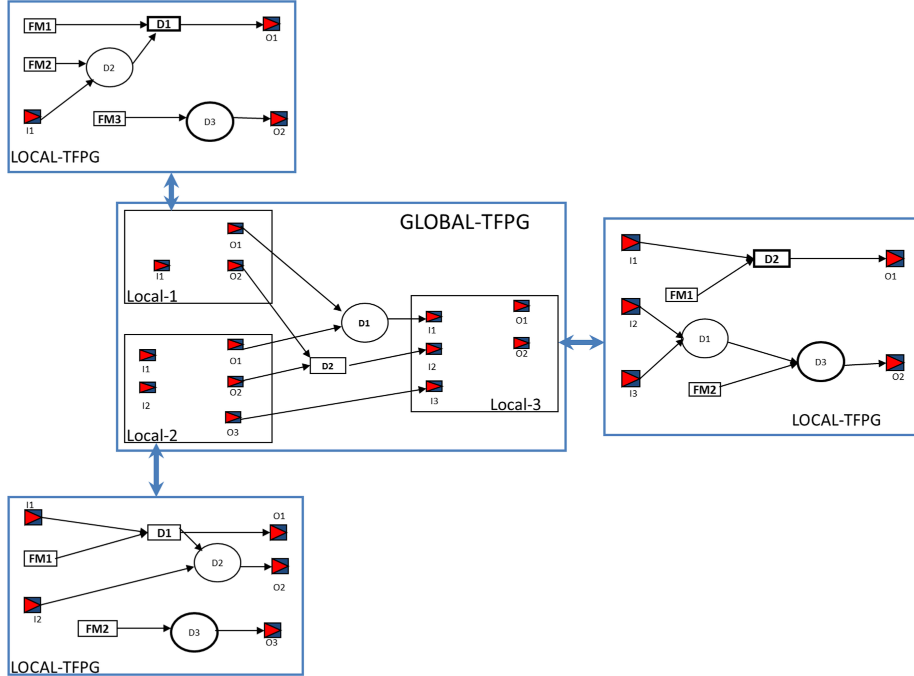

Fig. 3. Distributed TFPG model with three local TFPG models and one global TFPG model.

That is the purview of the local reasoners. The following sub-sections list extensions to the original TFPG model and algorithms. Readers are referred to [28] for further details.

Extensions to the TFPG Model: The original TFPG model - a tuple of $(F, D, E, M, \mathsf{ET}, \mathsf{EM}, \mathsf{DC})$ - needs to be extended to be able to express the global and subsystem TFPG-models.

The global and the subsystem model include two additional sets of elements

- (IP): a set of input ports,
- (OP): a set of output ports

The input and output ports are the interface points for connecting a subsystem model (monitored by a local reasoner) with the rest of the model. Any failure propagation into the subsystem model has to go through an input port. Likewise, any failure propagation out of the subsystem model has to go through an output port. In the TFPG failure propagation semantics, these input and output port extension elements are similar to un-monitored (silent) OR-discrepancies. While both the subsystem models and the global model are aware of the input and output ports associated with each subsystem model, there is a difference in their knowledge of the failure propagation interactions associated with these

ports. While each subsystem model is aware of only the failure propagation interactions within the subsystem, the global model is aware of only the failure propagation interaction outside of the subsystem models.

The global model is a TFPG model that is composed of multiple sub-system models and the failure propagation interaction between these sub-system models. As is evident from the model depicted in the figure 3 , the global model is not aware of the detailed TFPG-model inside a sub-system. It is aware of just the input and output ports of a sub-system model.

Extensions to the reasoner. The TFPG Algorithm 1 described earlier is central to the reasoning in the global and local reasoners as well. This algorithm is applied to reason upon any external physical event update - changes in the state of the alarm or modes. Additionally, in a local reasoner, if the update resulted in a change to the hypothetical state of any of the interface nodes (input and output ports), then the update needs to be communicated to the global then to the local reasoner which shares the interface element. In the reasoner receiving the update information, the update to the hypothetical state of the input/ output port, is treated like an external physical event update and the reasoning algorithm 1 is applied to reason about it. This process of applying the reasoning algorithm and communicating updates to the input/ output port from the local (global) to the global(local) reasoner continues, until the propagation stops. At this point, the metrics associated with the hypothesis in the local reasoner are re-computed and transmitted to the global reasoner which then recomputes the metrics of the hypothesis in the global reasoner.

All through the update propagation, the local and global reasoners are responsible for maintaining the hypothetical states of the nodes (including input/ output ports) in their respective TFPG models. In addition, the hypothesis in the local reasoner have a reference to a parent hypothesis in the global reasoner. Likewise, the hypothesis in the global reasoner has references to a hypothesis from each of the local reasoner. This reference tracking and maintaining of consistent hypothetical states of the shared interface elements (input / output ports) ensures that the hypothesis in the local and global reasoners are consistent with one another. This enables the hypothesis in the global reasoner to provide a system-wide consistent diagnosis result. This knowledge of a system-wide consistent result in the global, allows the global reasoner to find the best explanation, i.e. the fault source, for the sequence of alarms observed in the local reasoner(s). This information can be fed back to the local reasoner to help in prioritizing its hypotheses.

An essential aspect of the distributed diagnosis is its reliance on the communication between the reasoners. It should be noted that this algorithm permits a local reasoner to communicate only with the global reasoner. The communication messages transferred between Global and local reasoner(s) include

- (*Command*): Commands issued by the global reasoner to the local reasoner. This could include commands to initiate a diagnosis update, command to report diagnosis result or event update, etc.

– (*Response*): Message sent the local reasoner in response to a command from the global reasoner.
– (*Update*): Updates to the hypothetical state of an input/ output port sent in either direction between the global and its associated local reasoners.

The previous paragraphs provided an outline of the steps associated with processing event updates in distributed reasoning approach. The following paragraphs describe two specific strategies that were applied to handle reasoning about new events. Throughout the course of this discussion, it is assumed that the reasoner(s) and the event-generator(s)/event-reporter(s) share the same clock.

Synchronous Event Processing. In this strategy, when an external physical events (alarm/ mode updates) is received, the reasoner immediately reports it to the global reasoner, without updating the local state. The global reasoner stores all such external events (in any reasoner) in a queue, which is sorted by the event occurrence (or detection) time. This strategy assumes that the external physical events are reported to the global reasoner as soon as they are reported to the local reasoner. This ensures that the events are processed in the correct order. The global reasoner takes up the first event in the queue and commands the appropriate reasoner (where the event occurred) to process it. The reasoner applies the TFPG algorithm 1 to process the event. In case any updates on the hypothetical states of the interface elements are detected, the reasoner performs the updates and communicates them as described in the section 3.4. Once this step is finished, the global reasoner directs each of the local reasoners to report their local hypothetical results. It fuses these results to get a global consistent result and commands the local reasoners to update/synchronize their hypothesis with the global. This could involve elimination of unnecessary local hypothesis. The global reasoner takes up the next event in the queue and directs the appropriate reasoner to processes it. If the queue is empty, the global reasoner waits for a new physical event to be reported (either triggered directly in the global reasoner or reported from the local reasoner). Thus, in this strategy the global reasoner directs the execution of events across the areas.

Asynchronous Event Processing. In the asynchronous strategy, the local reasoner(s) does not wait for the global reasoners command to process / reasoner about an external physical event. As soon as an external physical event is reported to a local reasoner(s), it informs the global reasoner about this event. It also stores the event in a local sorted event queue. If the local reasoner is not processing any command from the global reasoner and its local event-queue is not empty, it starts processing these events from the queue. It stores the hypothesis updates based on these events in a separate list (the `transient-hypothesis-list`). Each local reasoner continues to maintain a primary hypothesis list that is synchronized with the global reasoner (the `stable-hypothesis-list`).

The global reasoner also maintains a sorted event queue. It processes the events in the queue in the similar manner as Synchronous Event Processing. When it is time to process the next event in the queue, it directs the appropriate reasoner to start processing it. Since the local reasoner could have already processed this event, it checks its transient hypothesis list for updates relating

to the event. If available, it does not process the event but updates the stable-hypothesis-list based on the results in transient-hypothesis-list. If not available, it processes the event and updates the stable-hypothesis-list. It then communicates the updates to the global reasoner. Next, this leads to propagation of updates across the system, which continues until the states stabilize. The global reasoner collects the updated hypothesis results from the local reasoners and creates an updated globally consistent result which is used to prune the stable-hypothesis-list in local reasoners.

Whenever the stable-hypothesis-list is updated in the local reasoner, hypotheses in its transient-hypothesis-list need to be checked if they need to be re-evaluated. It is possible that none of them need to be updated, or some or all of them need to be updated because of lack of consistency with the global stable state.

The two strategies described above have their own advantages and disadvantages. While with asynchronous updates, one can receive a quick update on a local event from a local reasoner. This is not possible in the synchronous mode of processing where the event is processed only when all events (across all reasoners) that occurred before it are processed. This is particularly inefficient if the local event was purely the result of a local fault or a cascading fault whose effect was already stored in a stable manner in the local reasoner. On the other hand, the asynchronous processing algorithm is much more complicated and involves a lot more book-keeping in the local reasoner. It could suffer from repeated re-computation if the stable-hypothesis-list involving the events are repeatedly updated by events currently being processed by the global reasoner.

The distributed reasoning approach described above has been implemented using a distributed software platform, and evaluated on numerous examples, but further refinements are subject of active research.

3.5 Distributed TFPG Examples

This section describes examples of distributed TFPG models and the results of the experiment performed using these models. Each reasoner (global and subsystem) was hosted on a separate process and the communication between the processes was realized through a CORBA-based middleware. Each test case involved starting the associated reasoner processes (global and subsystems) with the appropriate model and feeding the timed sequence of events to the appropriate reasoner (based on which subsystem the event triggered in). The total number of events triggered (across all subsystems and global) in each test case, depended on the nature of the test case. Some test cases triggered the alarms associated with one or more failure modes in the correct sequence as per the graph. Others altered these sequences by introducing false alarms, intermittent alarms, mode changes, etc.

Figure 4 shows a trivial distributed TFPG model. It shows a Distributed TFPG model with a global model and three subsystem (area) models. The global model is aware of the presence of subsystem models and the interface element (Input or Output port) in each of the subsystem models. The global

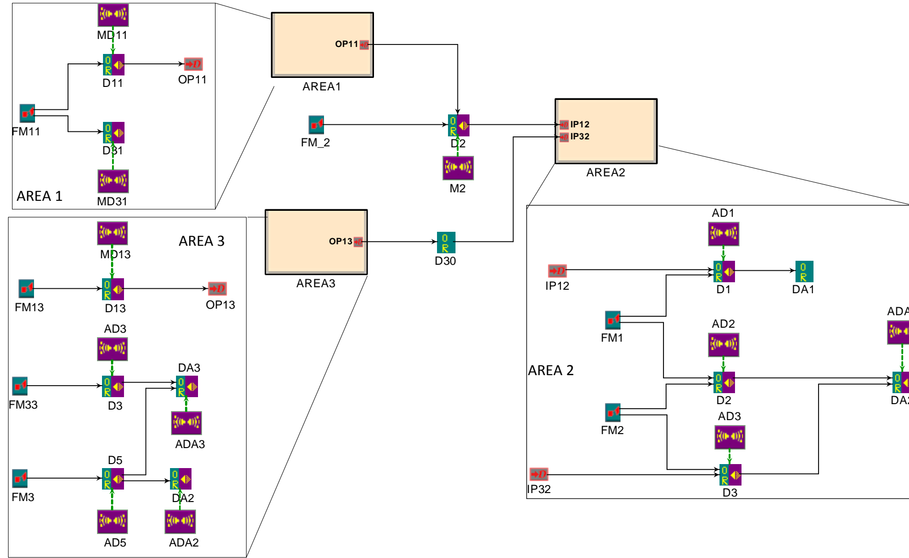

Fig. 4. An Illustrative Distributed TFPG Example

model is not aware of the internal detailed TFPG model in each subsystem. This model illustrates how failure-effects from Area1 (FailureMode - FM11) and Area3 (FailureMode-FM31) propagate out (through Output Ports OP11, OP31 respectively), enter the failure propagation graph in the global model, and finally reach into Area2 (through InputPorts IP21, IP22). It shows that there are portions of the TFPG model in some subsystems that do not participate in the cascading failure effects across subsystem boundaries.

The above example illustrates a simple Distributed TFPG model with 1 global model, 3 subsystem (Area) models, with one or two input and output ports in each of the Areas, 1-3 Failure Modes per subsystem, 2-5 Discrepancies, 3-10 failure propagation links. The next examples are distributed TFPG model developed for a small portion of real-life systems. Due to proprietary restrictions, these models can be presented only in an abstract format.

Example 3. Distributed System 1 The first example is a distributed TFPG model that includes a global (Global model) with 2 subsystems (AREA_1, AREA_2) . Here, the failure propagation across subsystem boundaries is strictly from AREA_2 to AREA_1. Table 1 describes the properties of the three TFPG models (Global, AREA_1, AREA_2). Offline analysis revealed that there were total 9 failure ambiguity groups in the global model, 11 failure ambiguity groups in AREA1 and 87 failure ambiguity groups in AREA2. This includes the ambiguities introduced by the secondary failures modes (the ports that carry in failure effects from other subsystems). The estimated worst case memory usage for global was around 30 MB, AREA1 is 7.5 MB and AREA2 is 25.4 MB.

The tests with this example were run with the 3 reasoners (2 subsystem reasoners and 1 global reasoner) hosted on a separate processes. The average time to update the hypothesis in the subsystem reasoners was 0.01 seconds in AREA1, 0.08 seconds in AREA2, while in the global reasoner it took about 0.11 seconds. This time includes the reasoning time as well as time to compute system-wide results assimilated from the set of local and global hypotheses. It was found that the latter took a significant share of the total update time. The average reasoning time was around 0.065 seconds.

Example 4. Distributed System 2 The next example of distributed TFPG model (again of a portion of a realistic system) is much richer in the failure propagation cascades across the subsystem model boundaries. Again, since the names or the exact nature of the system cannot be revealed these subsystems would be referred to as AREA_1, AREA_2,AREA_3,AREA_4. The symmetric nature of the system introduces two sets of subsystem models that are similar to one another. The global model has 4 subsystem models. The TFPG models of subsystem AREA_1 resembles that of subsystem AREA_2; AREA_3 model is similar to AREA_4.

Table 1 describes the properties of the five TFPG models (Global, AREA_1/ AREA_2, AREA_3/ AREA_4) Offline analysis revealed that there were a total of 38 failure ambiguity groups in the global model, 54 failure ambiguity groups in AREA1 (and AREA2) and 24 failure ambiguity groups in AREA3 (and AREA4), in each case including the ambiguities introduced by the secondary failures modes (the ports that carry in failure effects from other subsystems). The estimated worst case memory usage for global is around 74.5 MB, AREA_1/ AREA_2 is 47.5 MB and AREA_3/ AREA_4 is 24.5 MB. The tests were conducted in the same fashion as the previous example on the distributed reasoner. The difference is that in this example there were 2 more processes to account for the two additional subsystem reasoners. The time to update the hypothesis in the subsystem reasoners was 0.096-0.14 seconds, while in the global reasoner it took about 0.3 seconds. As stated in the previous example, this time includes the reasoning time, 0.18 seconds, as well as time taken to compute final system-wide results.

Discussion The experimental results for the centralized and distributed reasoners were presented. While it was found that in general the reasoner performed quite well, it could certainly be improved. As it can be expected, the update time was more when the event could not be explained consistently with existing hypotheses and new explanations (multiple failures) had to be found that were consistent with the current event as well as the past events.

In both distributed examples, the tests revealed that the update time was almost the same in both synchronous and asynchronous modes of processing. The difference was that in the asynchronous mode of operation, the transient result was available with the same rate as the average update time for a subsystem reasoner. Some other interesting observations were made regarding the update times in the global reasoner. The global reasoner (in the current setup) performs two tasks. The first task deals with reasoning on events and updating global hypotheses to keep a consistent state across all subsystem reasoners. The second

task involves computing the detailed results from all the local reasoners for reporting purposes, which involves the assimilation of global hypotheses and all local hypotheses. From our results, we infer that significant time is spent on the second task during each update. It is our opinion that this task is more relevant to report generation than reasoning. Hence, the table 1 shows two times for the global reasoner update, one that includes the total time of the two tasks and the other in parentheses that includes just the reasoning time (first task).

Another aspect that should be mentioned is that the total time spent in reasoning (in a global or local model) was influenced by local-events as well as the hypothetical updates received from the input and output ports. It was observed that the distributed reasoners (global and subsystem) had an average update time that was longer than a centralized reasoner operating on a model of similar size. This can be attributed to the issue that the global and subsystem reasoners have an extra load associated with the fault-cascades coming into the subsystem (from other subsystems). Whenever there is significant traffic of fault-cascades across the system boundaries, more time is spent in updating the local hypothesis relative to the boundary events. It should be noted that enough care and offline analysis should be performed while designing the boundaries of these subsystems such that they have low coupling across the subsystem boundary and heavy cohesion within the subsystem.

4 Application of TFPG for Diagnosing Software Failures

Modern cyber-physical systems, such as aircrafts, are increasingly becoming reliant on software for core functions and system integration [31, 2]. Increase in the scope of functions covered by software tends to increase the complexity of software, which often tends to increase the potential of latent software defects. These latent defects can escape the existing rigorous testing and verification techniques but manifest under exceptional circumstances. These circumstances may include faults in the hardware system, including both the computing and non-computing hardware. Often, systems are not prepared for such faults. Such problems have led to number of failure incidents in the past, including but not limited to those referred to in these reports: [5, 33, 7, 8, 20].

One way to counter these challenges is to systematically extend classical software fault tolerance techniques and apply the system health management for complex engineering systems to software systems. System health management typically includes anomaly detection, fault source identification (diagnosis), fault effect mitigation (in operation), maintenance (offline), and fault prognostics (online or offline) [36, 25].

Our research has focused on extending our diagnosis techniques described earlier in this chapter and apply it to software systems. It is essential to identify the boundaries of fault propagation in software systems in order to apply systematic diagnostic techniques such as TFPG. One approach is to ensure that software systems are built from *software components*, where each component is a reusable unit. Components, see figure 5, encapsulate (and generalize) objects

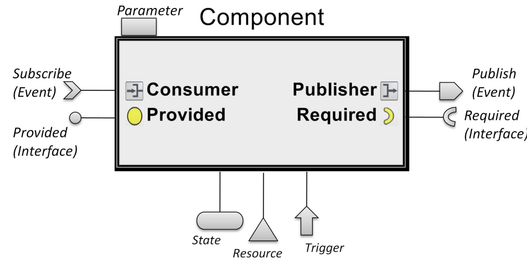

Fig. 5. The ARINC Component Model.

that provide functionality, and have well-defined ports for communication. We expect that these components are well-specified, independently developed, verified, and tested. Furthermore, all communication and synchronization among components is facilitated by a component framework that provides services for all component interactions with well-defined semantics, and no component interactions happen through 'out-of-band' channels. This component framework acts as a middleware, provides composition services, and facilitates all messaging and synchronization among components, and is used to support fault management.

Having well-specified interactions enable deduction of behavioral dependencies and failure propagation across a component assembly. Similar approaches exist in [14, 52]. The key differences between those and this work are that we apply an online diagnosis engine. Next section briefly describes an implementation of such a component framework and uses it to describe the application of online diagnosis to software systems.

4.1 ARINC Component Framework

The ARINC Component Framework (ACF) is a runtime software layer that implements the ARINC-653 component model (ACM) [15]. ACM borrows concepts from other software component frameworks, notably from the CORBA Component Model (CCM) [53], and is built upon the capabilities of ARINC-653 [1], the state-of-the-art operating system standard used in Integrated Modular Avionics. Key to ARINC-653 is the principle of spatial and temporal isolation among partitions. Discussion of abilities of ARINC-653 is out of scope of this chapter. However, interested readers are suggested to refer to [1].

In ACM, a component can have four kinds of external ports for interactions: **publishers, consumers, facets** (provided interfaces[2]) and **receptacles** (required interfaces), see fig 5. Each port has an interface type (a named collection of methods) or an event type (a structure). The component can interact with other components through **synchronous** interfaces (assigned to provided or required ports) and/or **asynchronous** event (assigned to event publisher or consumer ports). Additionally, a component can host internal methods that are periodically triggered. Each port can be periodic (i.e. time triggered) or aperiodic (i.e. event triggered). It is statically bound to a unique ARINC-653 process.

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

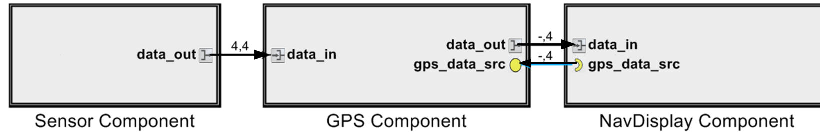

Fig. 6. Example for component interactions. Each interface is annotated with its (periodicity, deadline) in seconds.

This binding is defined and configured during initialization. Given that a provided interface can have more than one method, every method is allocated to a separate process.

Figure 6 shows a simple example assembly of components. The Sensor component contains an asynchronous publisher interface (source port) that is triggered periodically (every 4 sec). The event published by this interface is consumed by a periodically triggered asynchronous consumer/event sink port on the GPS component (every 4 sec). Note that the event sink process is periodically released, and each such invocation reads the last event published by the Sensor. The consumer process in the GPS, in turn, produces an event that is published through the GPS's event publisher port. This event triggers the aperiodic consumer / event sink port on the Navigation Display component. Upon activation, the display component uses an interface provided by the GPS to retrieve the position data via a synchronous method invocation call into the GPS component.

4.2 Health Management in ACM

With this framework, there are various levels at which health management techniques can be applied, ranging from the level of individual components to the whole system.

Component-level health management. (CLHM) provides localized and limited functionality for managing the health of one component by detecting anomalies, mitigating its effects using a reactive timed state machine, on the level of individual components. It also reports to higher-level health manager(s): the system health manager.

System-Level Health Manager. (SLHM) manages the overall health of the system i.e. assembly of all components. The CLHM processes hosted inside each of the components report their input (monitored alarms) and output (mitigation actions) to the SLHM. It is important to know the local mitigation action because it could affect how the faults cascade through the system. Thereafter, the SLHM is responsible for the identification of root failure source(s) - multiple failure mode hypotheses are allowed. Once the fault source is identified, appropriate mitigation strategy is employed.

Component Level Detection. The ACM framework allows the system designer to deploy monitors which can be configured to detect deviations from expected behavior, violations in properties, constraints, and contracts of an interaction port or component. Figure 7 summarizes various places where a component's

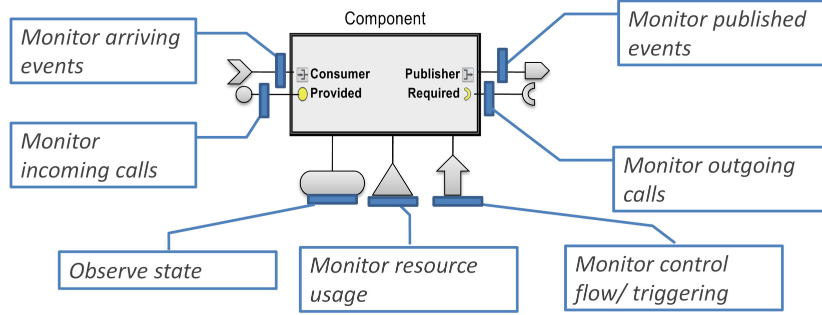

Fig. 7. Approach to component monitoring.

behavior can be monitored. These monitors check properties associated with the resource utilization (locks, deadline violation), data (validity, pre-condition and post-condition violation) and user-code (exception or user-code violation). The placement of a monitor is dependent on the property violation that it monitors. The monitors associated with resource usage, data validity, and deadline violation are implicitly handled and triggered by the framework. All other monitors have to be configured and enabled at design time. For all ports, exceptions in the user provided functional code are abstracted by the framework as an error triggered by a monitor associated with user-code. The monitors checking the pre-condition and post-condition violation on method calls are evaluated before and after the user provided functional code. These conditions are expressed over the current value or the historical change in the value, or rate of change of value of variables such as (a) the event-data of asynchronous calls, (b) function-parameters of synchronous calls, and (c) state variables of the component. Table 2 summarizes these monitors. While the monitors associated with resource usage are run in parallel by framework, other monitors are evaluated in the same thread executing the component port. When any monitor reports a violation, the status is communicated to its Component Level Health Manager (CLHM) and then possibly to the System Level Health Manager (SLHM).

Component Level Mitigation. Once a local discrepancy is detected, the framework reports the problem to the CLHM. CLHM for each component is deployed on a high-priority ARINC process that is triggered when any of the associated ARINC processes (hosting the Component ports) or the underlying ACM framework report any violations detected by monitors. In a blocking call, the reporting process waits for a response/mitigation action from the CLHM. It is important to note that a local reactive action changes the state of the system and is therefore important information that is required to ascertain the diagnosis. Table 3 summarizes all possible local mitigation actions.

Example 5. CLHM Example In this scenario, we inject a fault in the functional code written by us for the GPS example (see 6) such that between 10 and 20 seconds since its first launch, the GPS get data process sends out bad data when

<table>
<tr><td><PreCondition>::=<Condition></td></tr>
<tr><td><PostCondition>::=<Condition></td></tr>
<tr><td><Deadline>::=<double value> <i>/* from the start of the process associated with the port to the end of that method */</i></td></tr>
<tr><td><Data_Validity>::=<double value> <i>/* Max age from time of publication of data to the time when data is consumed*/</i></td></tr>
<tr><td><Lock Time Out>::=<double value> <i>/* from start of obtaining lock*/</i></td></tr>
<tr><td><Condition>::=<Primitive Clause><op><Primitive Clause>|<Condition><logical op><Condition>| !<Condition> | True| False</td></tr>
<tr><td><Primitive Clause>::=<double value>| Delta(Var)| Rate(Var)|Var
/* A Var can be either the component State Variable, or the data received by the publisher, or the argument of the method defined in the facet or the receptacle*/</td></tr>
<tr><td><op>::= < | > | <= | >= | == | !=</td></tr>
<tr><td><logical op>::=&& | ||</td></tr>
</table>

Table 2. Monitoring Specification. Comments are shown in italics.

CLHM Action	**Semantics**
IGNORE	Continue as if nothing has happened
ABORT	Discontinue current operation, but operation can run again
USE_PAST _DATA	Use most recent data (only for operations that expect fresh data)
STOP	Discontinue current operation
START	Re-enable a STOP-ped periodic operation
RESTART	STOP followed by a START for the current operation

Table 3. CLHM Mitigation Actions.

queried by the navigation display. The bad data is defined as the rate of change of GPS data being greater than a threshold. This fault simulates an error in the filtering algorithm in the GPS such that it loses track of the actual position.

Figure 8 shows a snapshot of the sequence of events in the experiment. The fault is injected approximately 18 seconds after the start of experiment. The navigation display component retrieves the current GPS data using the remote procedure call. Then, the post condition check of the remote procedure call is violated. This violation is defined by a threshold on the delta change of current GPS data compared to past data (last sample). The navigation display component raises an error, which is received by the local health manger. Consequently, it receives an ABORT response from the health manager. Notice that the execution of navigation display process is preempted till it receives a response from the health manager. The ABORT response means that the process that detected the fault should immediately stop processing and return. The effect of this action is that the navigation's GPS coordinates are not updated as the remote procedure call finished with an error.

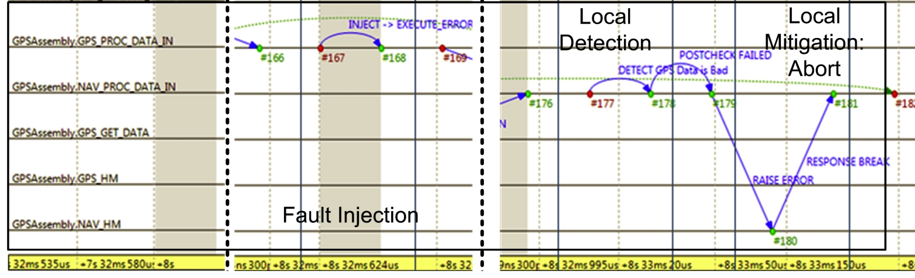

Fig. 8. Example of a component level health management action. The fault was injected in the GPS component to publish bad data. This was detected via the post condition of the Navigation component's required method. The specified reactive action is to abort the call. Note the time scale is non-linear.

Having described the local monitors and component level reactive action, we can discuss the TFPG model for a given software assembly. This model is used by the SLHM to diagnose failures. To facilitate this diagnosis, all component level managers are required to report their input (alarm/monitor) and output (response/ mitigation action) to the SLHM. As more information (alarms or alarms plus responses) becomes available, the SLHM (using the diagnosis engine) improves its fault-hypothesis, which can then be potentially used to drive the mitigation strategy at the system level.

4.3 Software Fault Propagation Model

As described in previous sections, in this framework the software assemblies are composed of components which are in turn composed of specific kinds of ports - Publisher, Consumer, Provides Interface, and Requires Interface. While these interaction ports can be customized by the event-data-types published/consumed, interfaces/methods exposed, periodicity, deadline etc., their fundamental behaviors and interaction patterns are well defined. This implies that it should be possible to identify specific faults and fault propagation pattern that are common to each kind of interaction pattern, which could result in a generic TFPG model for each interaction pattern (connection between two ports of different components). Thus, the failure propagation across the component boundaries can be captured from the assembly model. The generic TFPG model for a specific interaction port captures the following information:

1. Health Monitor Alarms and the Component Health Manager's Response to these alarms are captured as observable discrepancies
2. Failures originating from within the interaction port and the effect of their propagation as failure modes
3. Effect of failures originating from other entities as discrepancies
4. Cascading effects of failures within the interaction port as discrepancies
5. Effect of failures propagating to other entities as cascades

Additionally, a data and control flow model about the component internals (between the component processes), assists in capturing the failure propagation within the component. In principle, this approach is similar to the failure propagation and transformation calculus described by Wallace [52]. That paper showed how architectural wiring of components and failure behavior of individual components can be used to compute failure properties for the entire system.

Example 6. Generic TFPG for a Periodic Consumer

Figure 9 shows a generic TFPG model for a periodic consumer port. The list below explains the failure effects that are captured in a generic TFPG model through the example of the consumer TFPG model. The Consumer's TFPG model 9, is presented in terms of the failure propagations captured in the context of the observed alarms - LOCK_TIMEOUT, VALIDITY_FAILURE, PRE-CONDITION Violation, USER-CODE Failure, Deadline Violation. Each of these sub-graphs covers most of the points described above.

1. LOCK_TIMEOUT - This is caused by problems in obtaining the Component Lock. Being a real-time system, any attempt to obtain a lock is bounded by a maximum deadline. In case of timeout the fault is observed as a discrepancy with an anomaly of LOCK_TIMEOUT, resulting in a CLHM response of either IGNORE or REFUSE/ABORT. Based on the response, its failure effect can lead to an invalid or a missing state update and affect entities downstream.
2. VALIDITY_FAILURE - This is caused when the "age" of the data fed to the consumer is not valid. It is observed as a discrepancy with an anomaly of VALIDITY_FAILURE, resulting in a CLHM response of either USE_PAST_DATA or REFUSE/ABORT. Based on the response, its failure effect can lead to an invalid or a missing state update and effect entities downstream. The failure effect could also cascade into one or more of the anomalies described below.
3. PRE-CONDITION_FAILURE - This is caused when the pre-condition to the consumer process is not satisfied. This anomaly could also be observed as a result of a VALIDITY_FAILURE followed by a response to USE_PAST_DATA. It is observed as a discrepancy with an anomaly of PRE-CONDITION_FAILURE, resulting in a CLHM response of either IGNORE or REFUSE/ABORT. Based on the response, its failure effect can lead to an invalid or a missing state update and affect entities downstream. The failure effect could also cascade into user-code and/or deadline violation.
4. USER-CODE_FAILURE - This is caused when there is a failure in the user-code (e.g. exception). This anomaly could be observed as a result of a USER-CODE failure (captured as a failure mode) or a cascading failure effect propagation from VALIDITY_FAILURE and subsequent health management response of USE_PAST_DATA or PRE-CONDITION violation followed by an IGNORE response. Once observed, depending on the local CLHM response its failure effect can lead to an invalid or a missing state update and affect entities downstream. The failure effect could also cascade into a deadline violation.

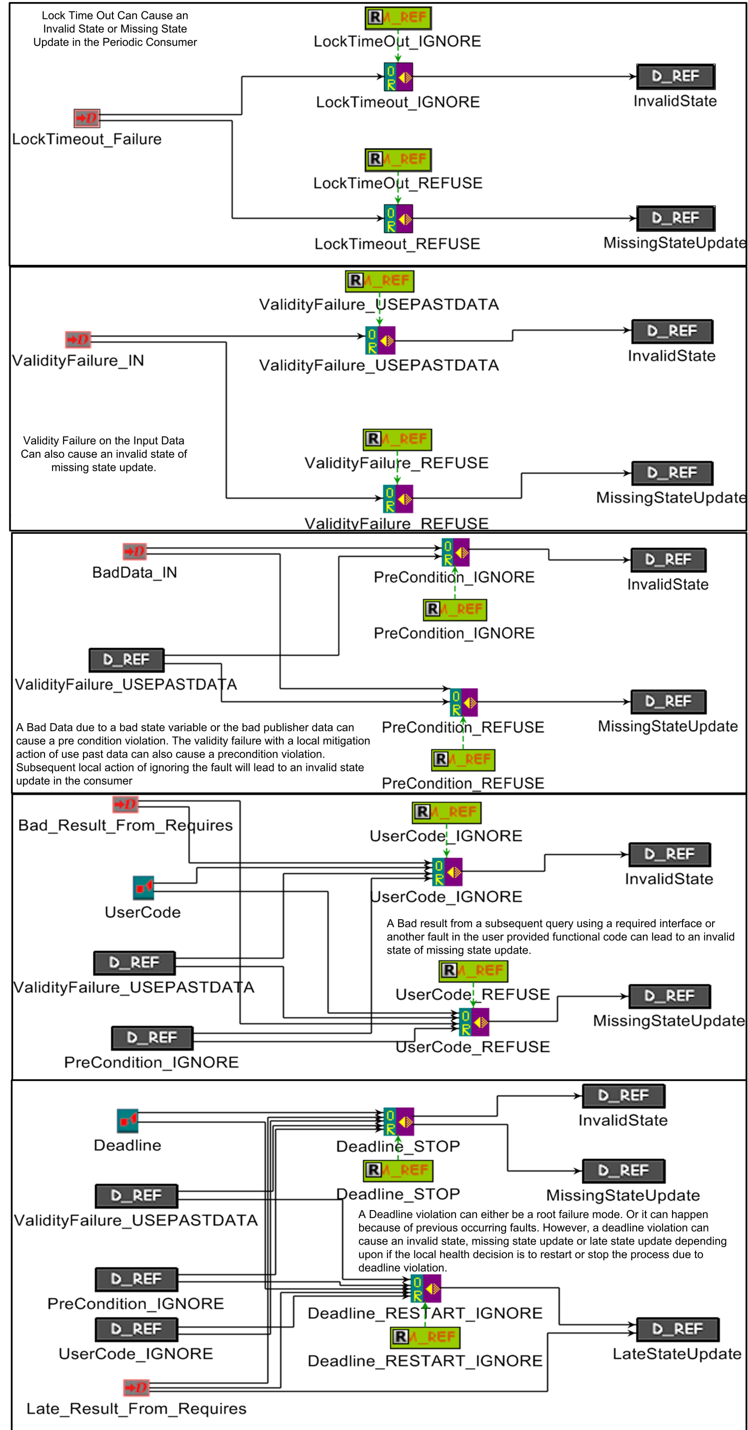

Fig. 9. TFPG template showing five (out of six possible) fault propagation patterns for a periodic consumer.

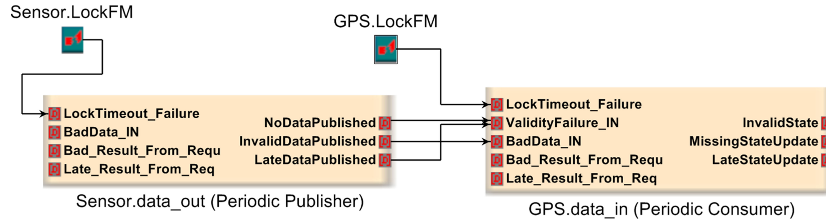

Fig. 10. TFPG model for publisher-consumer interaction.

5. DEADLINE_FAILURE - This is caused when the process deadline is violated. This anomaly could be observed as a result of a deadline failure (captured as a failure mode) or a cascading failure effect propagation from VALIDITY_FAILURE followed with a local health management response of USE_PAST_DATA or PRE-CONDITION violation with a local CLHM response of IGNORE or USER_CODE. It is observed as a discrepancy with an anomaly of DEADLINE_FAILURE, resulting in a CLHM response of either STOP (stopping the process in case of hard-deadline violation) or IGNORE (in case of soft-deadline violation) or RESTART. Based on the response, its failure effect can lead to an invalid / missing / late state update and affect entities downstream.

It should be noted that sometimes it might not be possible to monitor some of the failures / alarms mentioned above. In such cases, these observed discrepancies would turn into unobserved discrepancies and the fault effect would propagate through the discrepancy without raising any observation (alarm).

Example 7. TFPG for the GPS Assembly Figure 10 shows the failure propagation link created for interaction between a publisher and a periodic consumer. The publisher and consumer boxes encapsulate the detailed TFPG model of the publisher and the consumer entities. The failure propagation in-to or out-of the ports captures the failure effect propagation across the entity boundary. Any failure in the Publisher entity that could lead to a discrepancy of NoDataPublished / LateDataPublished could possibly cascade to the consumer entity through a VALIDITY_FAILURE in its input data. Likewise, a failure in the publisher leading to InvalidDataPublished discrepancy could produce anomalies in the consumer through the triggering of Bad-Data-Input discrepancy. The model also captures the possibility that a Failure-Mode of a problem in the Component lock could lead to discrepancies associated with Lock_Timeout in the Component's processes/ interaction ports.

Once we can create the models to capture the failure propagation across all interactions, we can essentially create TFPG model for the full assembly. In practice, the assembly level TFPG model is generated by instantiating the appropriate TFPG-model for the Component interaction ports and connecting

$$\#FM = \#Comp + \#Interactions * 3 - \#ReqPorts \quad (1)$$
$$\#Alarm = \#Interactions * 2 + \#Cons * 2 + \#Post * 2 + \#Pre * 2 \quad (2)$$

Table 4. Metrics for automatically generated TFPG from ACM component assembly. FM - Failure Modes, Comp - Components, Interactions - All Intra Component and Inter Component Interactions, ReqPorts - Required Interface Ports, Alarms - All observable TFPG discrepancies, Cons- Consumer Ports (Validity Monitors), Pre - Pre Conditions, Post - Post Conditions.

the failure propagation links between the discrepancy ports. The data and control flow within and across the component can be used to generate the failure propagation links across the instantiated TFPG models. This information can either be obtained by static analysis of the user-level code for the component or relying on the designer to provide this information. Currently, we take the latter approach.

Complexity of the generated model Table 4 describes the formulae for calculating the metrics for the size and complexity of failure propagation graphs generated from a given software assembly. These metrics are based on the templates of generic fault propagation across all possible inter and intra component interactions in the ACM framework.

Figure 11 shows the high-level TFPG model for the system/assembly described in figure 6. In this assembly, there were 3 components, 6 methods, 2 consumer ports, 1 requires port, and 2 post conditions. Hence, the number of failure modes totaled 20 and the number of alarms also totaled 20. The detailed TFPG-model specific to each interaction pattern is contained inside the respective TFPG component model (brown box). The figure shows failure propagation between the Sensor publisher[3] and GPS consumer[4], the GPS publisher [5] and NavDisplay consumer [6], the 'requires' method in NavDisplay[7] and the 'provides' method in GPS [8], the effect of the bad updates on state variables and the entities updating or reading the state-variables. Table 1 summarizes the worst case memory consumed by the reasoner and average time taken to update the hypothesis.

It should be noted that while the interaction pattern between a publisher port and a consumer port produces a fault propagation in the direction of data and event flow i.e. from the publisher to the consumer, the interaction pattern between a Requires interface and its corresponding Provides interface involves

[3] (Sensor_data_out)
[4] (GPS_data_in)
[5] (GPS_data_out)
[6] (NavDisplay_data_in)
[7] (NavDisplay_gps_data_src_getGPSData)
[8] (GPS_gps_data_src_getGPSData)

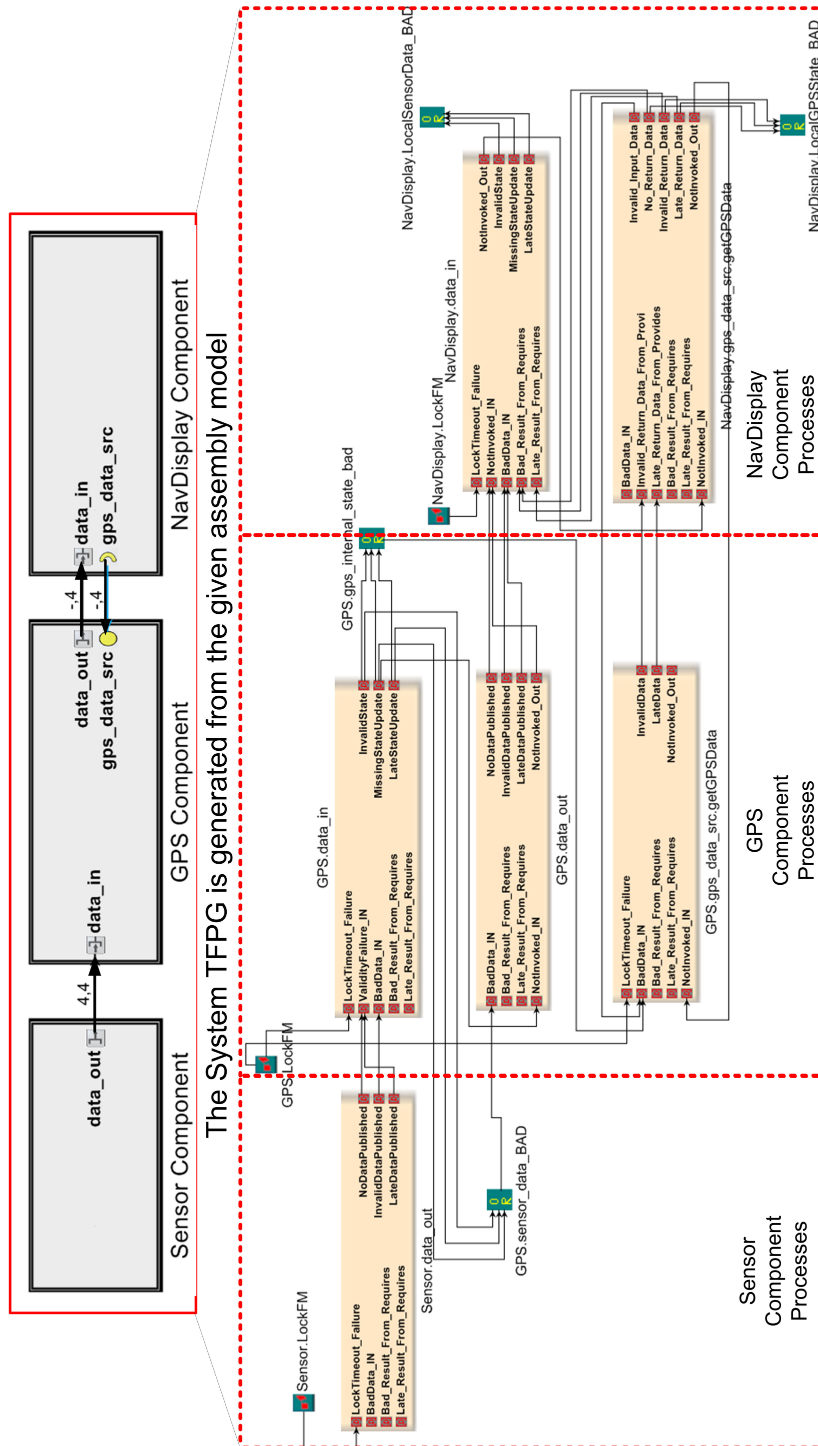

Fig. 11. TFPG model for the component assembly.

fault propagation in both directions. The fault propagation within a component is captured through the propagations across the bad updates on the state variables within the component. Currently, in the framework there is no instrumentation to deploy a monitor to specifically observe violations in state variable updates. These could be captured indirectly as pre-condition or post-condition monitors on the interfaces/interactions ports that update or read from these state variables.

4.4 The Diagnosis Process

The TFPG diagnosis engine is hosted inside a separate component.An alarm aggregator component is responsible for aggregating all the alarms and passing it to the diagnosis engine. These two components are hosted in a separated ARINC-653 partition. When the diagnosis engine receives the first alarm from a fault scenario, it generates all hypotheses that could have possibly triggered the alarm. Each hypothesis lists its possible failure modes and their possible timing interval, the triggered-alarms that are supportive of the hypothesis, the triggered alarms that are inconsistent with the hypothesis, the missing alarms that should have triggered and the alarms that are expected to trigger in future. Additionally, the reasoner computes hypothesis metrics such as Plausibility and Robustness that provide a means of comparison. The higher the metrics the more reasonable it is to expect the hypothesis to be the real cause of the problem. As more alarms are produced, the hypothesis are further refined. If the new alarms are supportive of existing set of hypotheses, they are updated to reflect the refinement in their metrics and alarm list. If the new alarms are not supportive of any of the existing hypotheses with the highest plausibility, then the reasoner refines these hypotheses such that hypotheses can explain these alarms.

Figure 12 shows the diagnosis results for a specific scenario wherein the Sensor (figure 6) stops publishing data. This results in failure effect that cascades across component boundaries. The initial alarm is generated because of data validity violations in the consumer of the GPS component. When this alarm was reported to the local Component Health manager, it issued a response directing the GPS component to use past data (USE_PAST_DATA). While the issue was resolved local to the GPS component, the combined effect of the failure and mitigation action propagated to the Navigation Display component. In the Navigation Display component, a monitor observing the post-condition violation on a Required interface was triggered because the GPS data validated its constraints. These two alarms where sent to the System Level Health Manager and processed by the TFPG diagnoser.

As can be seen from the results, the system correctly generated two hypotheses. The first hypothesis blamed the sensor component lock to be the root problem. The second hypothesis blamed the user level code in the sensor publisher process to be the root failure mode. In this situation the second hypothesis was the true cause. However, because we current treat the lock time out monitors as unmonitored discrepancies the diagnoser was not able to reasonably disambiguate between the two possibilities.

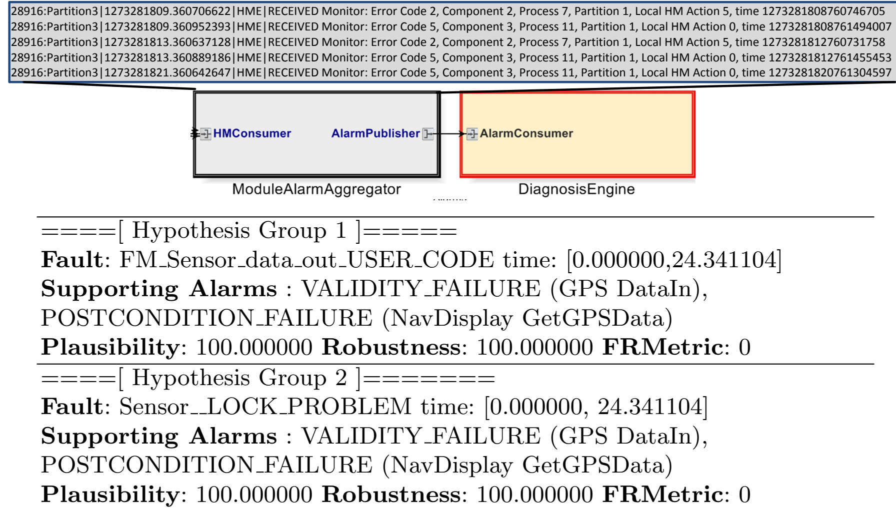

Fig. 12. Diagnosis results from the TFPG reasoner.

5 Application of TFPG For Prognostics of Impending Faults

In general, the aim of failure prognosis is to estimate the system reliability, given a set of conditions and observations, by assessing how close the system is to a critical manifestation of current failures. The reliability estimation can then be used to reconfigure the system, change the operating settings, or schedule specific maintenance procedures targeting the faulty components. In the TFPG modeling and reasoning settings, the prognosis problem and the associated reliability measure can be defined based on three main factors; failure criticality levels, diagnosis or hypothesis plausibility, and the time distance from the current state to the critical failure.

The first factor addresses the common situation in which different sections of the system may correspond to different levels of criticality with respect to system functionality. These sections can be identified using a measure of criticality assigned to all discrepancies in the systems. The second factor addresses the current estimated (diagnosed) condition of the system and the plausibility of the corresponding hypothesis. The third factor addresses the timing proximity of the current estimated state relative to a given critical region of the system. All these factors directly contribute to the reliability of the system at a given time. We will discuss them in details in the rest of this section.

5.1 Failure Criticality

In typical practical situations, failure progresses starting from the initial failure modes into several stages with increasing level of criticality. To capture this

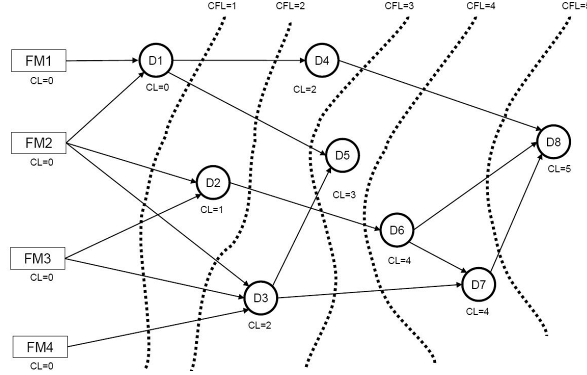

Fig. 13. TFPG model with assigned criticality levels and the corresponding criticality fronts.

situation, we define the map $\mathsf{CL} : D \to \mathbb{N}$ that assign to each discrepancy, $d \in D$, a criticality level $\mathsf{CL}(d)$. The lowest criticality level, 0, is reserved for non-critical discrepancies and all failure modes. To capture the increasing criticality with respect to propagation depth, we assume that

$$(\forall d', d \in D) \quad (d', d) \in E \longrightarrow \mathsf{CL}(d') \leq \mathsf{CL}(d)$$

The above condition states that if d' is a parent of d in a TFPG model then the criticality level of d' should be less than or equal to that of d. As a consequence, the criticality levels along any given path in a TFPG model form a monotonically increasing sequence. Note that we only assign a criticality level to all monitored and non-monitored discrepancies D. Failure modes are assigned a criticality level of 0 by default.

Based on the definition of criticality levels, we can define *criticality fronts* associated with a given TFPG model by the map, $\mathsf{CF} : \mathbb{N} \to \mathcal{P}(D)$, where,

$$(\forall d \in D) \quad d \in \mathsf{CF}(n) \quad \longleftrightarrow \quad \mathsf{CL}(d) \geq n \text{ and } (\forall (d', d) \in E)\ \mathsf{CL}(d') \leq n$$

The set of criticality fronts are essentially the codomain of the above map, and the set of criticality front levels CFL are the set $\{n \in \mathbb{N} \mid \mathsf{CF}(n) \neq \varnothing\}$. It can be shown that CFL corresponds bijectively to the codomain of CL. Based on the above definitions, a criticality front level, $n \in \mathbb{N}$, corresponds to a graph cut of the TFPG model in which the nodes on one side of the cut have criticality levels less than n and the remaining nodes have criticality level greater than or equal to n. Figure 13 shows an example TFPG model with assigned criticality levels and the corresponding criticality fronts.

Criticality levels are typically assessed based on the requirements for system operation and functionality. In particular, the criticality value for a given discrepancy depends on the operation cost associated with the fault reaching and progressing from this discrepancy. This will include the cost of maintenance, reconfiguration, and recovery when applicable. However, in some situations, it

is not possible to have a precise value for the criticality of a sensor. In such situations, an enumeration of criticality levels (ex. low, medium, and high) can be used to distinguish between sensors with respect to fault severity. Such enumeration can be assigned an approximate integer value, that reflects its relative importance, which can be used later in this section to compute a reliability measure for the system, in terms of the remaining useful life (RUL) or the time to criticality.

5.2 State Estimation Plausibility

As discussed in the previous section, the TFPG reasoning algorithms relies on sensor signals (alarms) and the TFPG model structure to identify the most plausible estimates of the current system condition as a set of state hypotheses. The plausibility of each hypothesis is defined based on the number of supporting sensor signals (alarms) versus the inconsistent and missing ones. We will write $A(H)$ for the set of discrepancies that are presumed active (ON) according to H and $I(H)$ for the set of discrepancies that are presumed inactive (OFF) according to H. That is,

$$A(H) = \{d \in D \mid H(d).\text{state} = \texttt{OFF}$$

The state front of a hypothesis H is denoted SF_H and is defined as a set of discrepancies $D' \subseteq D$ such that $(\forall d \in \mathsf{SF}_H)$

$$d \in A(H) \text{ and } (\exists (d, d') \in E)\ d' \in I(H)$$

Accordingly, the state front SF_H is the set of discrepancies that are currently active as estimated by H but some of their children discrepancies are not active according to H. Given that any discrepancy in D can either be in $A(H)$ or $I(A)$ but not both. The set SF_H is well-defined and the boundary line between D' and $D - D'$ forms a graph cut for the underlying TFPG model.

The intuitive meaning of the state front for a hypothesis, is that all the discrepancies on this front have the same likelihood of being active at the current time and they are forming the front of fault propagation in the sense that they are the discrepancies that could become active based on the next set of alarms as the fault propagation continues to progress.

The plausibility of a state front is equal to the plausibility of the underlying hypothesis. It is possible that several hypotheses can have the same plausibility level and therefore several state fronts may have the same plausibility level.

5.3 Time Proximity

The time proximity factor measures how close the current state of the system is to a future failure. As discussed earlier, future failures are identified by their criticality level front as defined by the map CF. Each front is defined as a set of discrepancies at the boundary of a graph cut for the TFPG model. Similarly, a state estimation is defined by a hypothesis (with its plausibility level) and

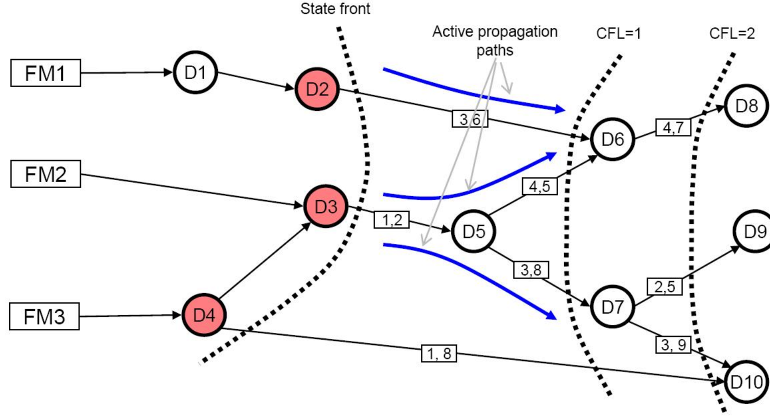

Fig. 14. Example for time to criticality.

is identified by the discrepancies at the boundaries of the cut formed by the underlying hypothesis level state front. Accordingly, the time proximity factor is a measure for the temporal distance between two fronts (graph cuts) each corresponding to a set of discrepancies in the TFPG model.

To define such distance, consider two sets of discrepancies $D_1, D_2 \subseteq D$ such that all discrepancies in D_1 are either ancestors of some discrepancies in D_2 or not connected to any discrepancy in D_2. In this case, we write $D_1 \prec D_2$. We define the propagation time between D_1 and D_2 with respect to a hypothesis H, denoted $t_H(D_1, D_2)$, as the minimum time to trigger discrepancy in D_2 as a result of failure propagation from discrepancies in D_1.

To compute $\hat{t}_H(D_1, D_2)$, we first consider the set of all discrepancies that are children of D_1. We then compute the earlier propagation time to these discrepancies based on the activation times of their parent nodes according to H. The computation of the earliest propagation time for all subsequent nodes is done based on the earliest propagation times available for their parents. This incremental computation continues until the earliest propagation time is computed for all the nodes in D_2. The minimum time is selected as the output. Algorithm 2 outlines the computation procedure.

In algorithm 2, for a given subset of nodes X in the TFPG model, $\mathsf{RSet}(X)$ is the set of discrepancies outside of X where all their parents belong to X. The function $\mathrm{terl}(\mathtt{TNodes}, d)$ computes the earliest time for d to become activated based on the earliest time the parents of d are activated. This function can be directly computed based on the semantics of failure propagation in TFPG models.

5.4 Time to Criticality

Given a set of criticality levels, the associated criticality fronts are computed directly from the earlier definition. The state front for a given hypothesis can be

Algorithm 2 The propagation time procedure $\hat{t}_H(D_1, D_2)$

assumption: $D_1 \prec D_2$
if $D_1 \cap D_2 \neq \varnothing$ **then**
  return 0
end if
define $\mathsf{RSet}(X) := \{d \in D - X | (\forall d' \in D)\ (d', d) \in E \to d' \in X\}$
$\texttt{TNodes} = \{(d, H(d).\text{terl}) \mid d \in A(H)\}$
$t_{\min} = \infty$
while $D_2 \not\subset \texttt{TNodes}.\text{nodes}$ **do**
  select d from $\mathsf{RSet}(D_1)$
  compute $\text{terl}(\texttt{TNodes}, d)$
  if $d \in D_2$ **then**
    $t_{\min} = \min(t_{\min}, \text{terl}(d))$
  end if
end while
return $t_{\min}$

directly computed based on the given definition. Let Y be the set of hypothesis with the highest plausibility value at a time t. We define the time to criticality level n at a given time t, denoted $\mathsf{TTC}(Y, n)$, as follows

$$\mathsf{TTC}(Y, n) = \min\{\hat{t}_H(\mathsf{SF}_H, \mathsf{CF}(n)) \mid H \in Y\}$$

That is, the time to criticality level n is the minimum of all (earliest) propagation times for all hypotheses with the maximum plausibility. In practice, there are typically few enumerated criticality levels. The time to criticality, therefore, follows the increasing order of the criticality. That is, the time to reach a high criticality level is usually longer than the time to reach a lower criticality level, as expected. However, this is not always the case as shown the in Figure 14. In this example, there are three different paths from the state (estimation) front SF_H to the criticality front level 1 (CFL=1), where H is the most plausible hypothesis in which D2, D3, D4 are assumed active and D1 is assumed faulty. In this example, the time to criticality to the first level is 3 (time units) while the time to criticality for the next higher level is 1.

6 Relation to Machine Learning and Data Mining

The TFPG model for a specific physical system, as described above, is the result of an engineering process that is based on the engineer's knowledge of the system, first principles, and system connectivity. For large systems this model construction can be quite complex, and will rarely be perfect on the first try. Hence, a model maturation process is needed where the TFPG model is refined over time, based on operational experience. Arguably, machine learning and/or data mining techniques can be used to assist in this maturation, as discussed below.

Imperfections in TFPG models come in two main forms: (1) a failure propagation edge is missing, and (2) a failure propagation edge is present where it should not exist. Assume that we operate the reasoner using such degraded model. If the input and the output of the reasoner are logged, one can apply data mining techniques to the input, with the goal of discovering the weak points in the model. For validation, we will also need the result of the maintenance activity that repaired the system, i.e. the knowledge of the real, physical failure mode.

If a failure propagation edge is missing from the model then the reasoner will likely produce degraded results and mark the downstream alarms as false alarms, indicating that they are not on a valid propagation path. However, if the data mining algorithm tells that there is a strong correlation between the two alarms, and such correlation can be supported because of a physical connection in the system, then it is likely that an edge is missing, and adding it would improve the performance of the reasoner.

If a failure propagation edge is present but it should not be, then the reasoner will again produce degraded results because it will misidentify failure modes as fault sources. Again, if the data mining indicates that there is very little correlation between the two alarms, and this can be corroborated by the lack (or the weakness) of physical connections between the components, then it is likely that the edge is superfluous and its removal would improve the diagnostic reasoning.

In either case, the change on the model shall be validated with the real physical failure mode. This can be done by supplying the same input sequence to the reasoner but now with the changed model, and observing if it is able to correctly produce the real failure mode. Once this validation is performed, the model can be fielded and used.

7 Summary

We described a discrete-event based, graph-oriented approach to fault source isolation that is applicable in the context of system level health management. We presented the modeling approach, the algorithms for the centralized reasoner, the algorithms for a distributed, master-slaves reasoner architecture. We have also illustrated how the same approach can be used for software health management, and potentially integrating reasoners about the physical and the software components of a system. The modeling approach and the reasoner have been applied in various example systems, and were found to provide satisfactory performance. The software health management application has been recently developed and tested only on small examples. The full, system-level integration and application of the reasoner approach are the subject of active research and development.

Acknowledgments

This paper is based upon work supported in part by NASA, the Boeing Company and the DARPA Software Enabled Control program. 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 Boeing Company or DARPA. The authors would like to thank Paul Miner, Eric Cooper, and Suzette Person of NASA Langley Research Center and Stan Ofsthun, Tim Wilmering, Erik Fries, and John Wilson of the Boeing Company for their help and guidance.

References

1. ARINC Specification 653-2: Avionics Application Software Standard Interface Part 1 - Required Services. Tech. rep.
2. Joint advanced strike technology program, avionics architecture definition – appendix B. Tech. rep., August 1994.
3. Abdelwahed, S., Karsai, G., and Biswas, G. System diagnosis using hybrid failure propagation graphs. In *The 15th International Workshop on Principles of Diagnosis* (Carcassonne, France, 2004).
4. Abdelwahed, S., Karsai, G., and Biswas, G. A consistency-based robust diagnosis approach for temporal causal systems. In *The 16th International Workshop on Principles of Diagnosis* (Pacific Grove, CA, 2005).
5. Ariane 5 inquiry board report. Tech. rep., Available at http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf, June 1996.
6. Brusoni, V., Console, L., Terenziani, P., and Dupre, D. T. A spectrum of definitions for temporal model-based diagnosis. *Artificial Intelligence 102*, 1 (1998), 39–79.
7. Bureau, A. T. S. In-flight upset; 240km nw perth, wa; boeing co 777-200, 9m-mrg. Tech. rep., August 2005.
8. Bureau, A. T. S. Ao-2008-070: In-flight upset, 154 km west of learmonth, wa, 7 october 2008, vh-qpa, airbus a330-303. Tech. rep., October 2008.
9. Conmy, P., McDermid, J., and Nicholson, M. Safety analysis and certification of open distributed systems. In *International System Safety Conference,* (Denver, 2002).
10. Console, L., and Torasso, P. On the co-operation between abductive and temporal reasoning in medical diagnosis. *Artificial Intelligence in Medicine 3*, 6 (1991), 291–311.
11. Darwiche, A. Model-based diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research 8* (1998), 165–222.
12. Darwiche, A., and Provan, G. Exploiting system structure in model-based diagnosis of discrete-event systems. In *Proc. of the Seventh International Workshop on Principles of Diagnosis* (1996), pp. 95–105.
13. de Kleer, J., Mackworth, A., and Reiter, R. Characterizing diagnoses and systems. *Artificial Intelligence 56* (1992).
14. de Lemos, R. Analysing failure behaviours in component interaction. *Journal of Systems and Software 71*, 1-2 (2004), 97 – 115.

15. Dubey, A., Karsai, G., Kereskenyi, R., and Mahadevan, N. A Real-Time Component Framework: Experience with CCM and ARINC-653. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on* (2010), 143–150.
16. et al., S. A. RTES demo system 2004. *SIGBED Rev. 2*, 3 (2005), 1–6.
17. Frank, P. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy - a survey and some new results. *Automatica 26*, 3 (1990), 459–474.
18. Gamper, J. *A Temporal Reasoning and Abstraction Framework for Model-Based Diagnosis Systems.* PhD thesis, RWTH, Aachen, Germany, 1996.
19. Goldberg, A., and Horvath, G. Software fault protection with ARINC 653. In *Proc. IEEE Aerospace Conference* (March 2007), pp. 1–11.
20. Greenwell, W. S., Knight, J., and Knight, J. C. What should aviation safety incidents teach us? In *SAFECOMP 2003, The 22nd International Conference on Computer Safety, Reliability and Security* (2003).
21. Hamscher, W., Console, L., and de Kleer, J. *Readings in model-based diagnosis.* Morgan Kaufmann Publishers Inc., San Francisco, CA, 1992.
22. Hessian, R., Salter, B., and Goodwin, E. Fault-tree analysis for system design, development, modification, and verification. *IEEE Transactions on Reliability 39*, 1 (1990), 87–91.
23. Himmelblau, D. M. Fault detection and diagnosis in chemical and petrochemical processes. *Chemical Eng. Mon. 8* (1978).
24. Ishida, Y., Adachi, N., and Tokumaru, H. Topological approach to failure diagnosis of large-scale systems. *IEEE Trans. Syst., Man and Cybernetics 15*, 3 (1985), 327–333.
25. Johnson, S., Ed. *System Health Management: With Aerospace Applications.* John Wiley & Sons, Inc, Based on papers from First International Forum on Integrated System Health Engineering and Management in Aerospace, 2005. To Appear in 2011.
26. Karsai, G., Biswas, G., and Abdelwahed, S. Towards fault-adaptive control of complex dynamic systems. In *Software-Enabled Control: Information Technology for Dynamical Systems*, T. Samad and G. Balas, Eds. IEEE publication, 2003, ch. 17.
27. Karsai, G., Sztipanovits, J., Padalkar, S., and Biegl, C. Model based intelligent process control for cogenerator plants. *Journal of Parallel and Distributed Systems 15* (1992), 90–103.
28. Mahadevan, N., Abdelwahed, S., Dubey, A., and Karsai, G. Distributed diagnosis of complex causal systems using timed failure propagation graph models. In *IEEE Systems Readiness Technology Conference, AUTOTESTCON* (2010).
29. Misra, A. *Sensor-Based Diagnosis of Dynamical Systems.* PhD thesis, Vanderbilt University, 1994.
30. Misra, A., Sztipanovits, J., and Carnes, J. Robust diagnostics: Structural redundancy approach. In *SPIE's Symposium on Intelligent Systems* (1994).
31. Morgan, D. Pave pace: system avionics for the 21st century. *Aerospace and Electronic Systems Magazine, IEEE 4*, 1 (Jan. 1989), 12 –22.
32. Mosterman, P. J., and Biswas, G. Diagnosis of continuous valued systems in transient operating regions. *IEEE Trans. on Systems, Man and Cybernetics 29*, 6 (1999), 554–565.
33. NASA. Report on the loss of the mars polar lander and deep space 2 missions. Tech. rep., NASA, 2000.

34. Neema, S., Bapty, T., S.Shetty, and S.Nordstrom. Developing autonomic fault mitigation systems. *Journal of Engineering Applications of Artificial Intelligence Special Issue on Autonomic Computing and Grids* (2004).
35. Nicholson, M. Health monitoring for reconfigurable integrated control systems. *Constituents of Modern System safety Thinking. Proceedings of the Thirteenth Safety-critical Systems Symposium. 5* (2007), 149–162.
36. Ofsthun, S. Integrated vehicle health management for aerospace platforms. *Instrumentation Measurement Magazine, IEEE 5*, 3 (Sept. 2002), 21 – 24.
37. Padalkar, S., Sztipanovits, J., Karsai, G., Miyasaka, N., and Okuda, K. C. Real-time fault diagnostics. *IEEE Expert 6*, 3 (1991), 75–85.
38. Patton, R. Robust model-based fault diagnosis:the state of the art. In *IFAC Fault Detection, Supervision and Safety for Technical Processes* (Espoo, Finland, 1994), pp. 1–24.
39. Patton, R., Frank, P., and Clark, R. *Fault Diagnosis in Dynamic Systems: Theory and Application*. Prentice Hall International, UK, 1989.
40. Piccoli, L., Dubey, A., Simone, J. N., and Kowalkowlski, J. B. Lqcd workflow execution framework: Models, provenance and fault-tolerance. *Journal of Physics: Conference Series 219*, 7 (2010), 072047.
41. Rao, S. V. N., and Viswanadham, N. Fault diagnosis in dynamical systems: A graph theoretic approach. *Int. J. Systems Sci. 18*, 4 (1987), 687–695.
42. Rao, S. V. N., and Viswanadham, N. A methodology for knowledge acquisition and reasoning in failure analysis of systems. *IEEE Trans. Syst., Man and Cybernetics 17*, 2 (1987), 274–288.
43. Reiter, R. A theory of diagnosis from first principles. *Artificial Intelligence 32*, 1 (1987), 57–95.
44. Richman, J., and Bowden, K. R. The modern fault dictionary. In *International Test Conference* (1985), pp. 696–702.
45. Sammapun, U., Lee, I., and Sokolsky, O. Rt-MaC: runtime monitoring and checking of quantitative and probabilistic properties. In *Proc. 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (17–19 Aug. 2005), pp. 147–153.
46. Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., and Teneketzis, D. Failure diagnosis using discrete event models. *IEEE Trans. Contr. Syst. Technol. 4* (1996), 105–124.
47. Saxena, T., Dubey, A., Balasubramanian, D., and Karsai, G. Enabling self-management by using model-based design space exploration. *Engineering of Autonomic and Autonomous Systems, IEEE International Workshop on 0* (2010), 137–144.
48. Scherer, W. T., and White, C. C. A survey of expert systems for equipment maintenace and diagnostics. In *Knowledge-based system diagnosis, supervision and control*, S. G. Tzafestas, Ed. Plenum, New York, 1989, pp. 285–300.
49. Srivastava, A. N. Discovering system health anomalies using data mining techniques. In *Proceedings of the Joint Army Navy NASA Air Force Conference on Propulsion* (2005).
50. Tzafestas, S., and Watanabe, K. Modern approaches to system/sensor fault detection and diagnosis. *J. A. IRCU Lab. 31*, 4 (1990), 42–57.
51. Viswanadham, N., and Johnson, T. L. Fault detection and diagnosis of automated manufacturing systems. In *27th IEEE Conference on Decision and Control* (1988).

52. Wallace, M. Modular architectural representation and analysis of fault propagation and transformation. *Electron. Notes Theor. Comput. Sci. 141*, 3 (2005), 53–71.
53. Wang, N., Schmidt, D. C., and O'Ryan, C. Overview of the corba component model. *Component-based software engineering: putting the pieces together* (2001), 557–571.